ACER: MANIPULATION PRINCIPLES APPLIED TO LANGUAGE DESIGN

by

Eduardus Antonius Theodorus Merks

B.Sc. Simon Fraser University, 1986

M.Sc. Simon Fraser University, 1987

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
April 1992

APPROVAL

Name:	Eduardus Antonius Theodorus Merks
Degree:	Doctor of Philosophy
Title of thesis:	Acer: Manipulation Principles Applied To Language Design
Examining Committee:	Dr. J.J. Weinkam, Professor Chair
	Dr. R.D. Cameron, Associate Professor Senior Supervisor
	Dr. F.W. Burton, Professor Committee Member
	Dr. R.F. Hobson, Associate Professor Committee Member
	Dr. A.H. Dixon, Senior Lecturer SFU Examiner
	Dr. G.V. Cormack, Associate Professor External Examiner, University of Waterloo
Date Approved:	April 14, 1992

Abstract

Programming language design is explored from the viewpoint that support for program manipulation is a fundamental guiding concern. Three general areas of language design are identified as being of particular significance in terms of support for manipulation, namely the mapping between concrete syntax and abstract syntax, the mapping between static semantics (context-dependent syntax) and abstract syntax, and the mapping between equivalent language constructs. Abstract syntax-tree nodes and their context-dependent relations are the unifying concept in this realm.

A particular programming language, Acer, based on the typeful programming language Quest, is designed and implemented to illustrate how support for manipulation is enhanced. Acer is a general-purpose, imperative language with a full accompaniment of modern language features, as well as a number of novel features (e.g., persistent storage). Its concrete syntax is designed to meet strict requirements, e.g., every node gives rise to a token so that it is visible for selection or annotation. Its abstract syntax is similarly strict and provides node representations for all semantic objects. Hence, semantic relations are simply relations on nodes and semantics-preserving transformations, such as folding and unfolding, are supported as simple transformations of node structure.

Acer's support for manipulation demonstrates the benefits of designing abstract syntax first and treating concrete syntax as a particular way of viewing abstract syntax. It also demonstrates that a concrete syntax can be designed which is both natural in appearance and yet highly constrained. And, perhaps most importantly, it demonstrates that imperative languages can support the same kinds of powerful transformations supported by functional languages, e.g., all expressions can be folded.

Acknowledgements

A thesis cannot be written without the support of others, and this thesis is no exception. I gratefully acknowledge the contributions of the following people: Rob Cameron for his guidance and financial support over many years; Michael Dyck for his attention to detail and his constructive commentary; my family for their encouragement; Larry Thiele for being there; Howard Hamilton and Andrew Kurn for their helpful advice; the examining committee for their time, and the many other people at Simon Fraser University with whom I have had the pleasure of sharing the past ten years.

For the financial contribution toward research and research facilities, and toward my graduate work in particular, Simon Fraser University, the British Columbia Advanced Systems Institue, and the Natural Sciences and Engineering Research Council of Canada also deserve recognition.

Contents

Chapter 1

The representation of language

1.1 Programming language design

This thesis explores programming language design from the viewpoint that support for program manipulation is a fundamental guiding concern. A program, after all, is a structured object which throughout its lifetime is the target of numerous manipulation activities, including everything from initial construction to subsequent debugging and maintenance. As such, the design of a program representation that facilitates manipulation is of primary importance.

Consider, for instance, the design of a representation of natural numbers. Roman numerals provide an adequate representation but Arabic numerals provide a manipulable representation: they facilitate meaningful semantic manipulation (e.g., addition and multiplication of numbers) via simple syntactic manipulation (e.g., combining the digits of numerals). For a language of programs, as for a language of numbers, such a manipulable representation is best.

Illuminating the path towards the design of more manipulable languages is the well-recognized observation that a language exists at three levels of abstraction:

- meaningful semantic objects
- abstract syntactic objects
- concrete lexical objects

Accordingly, language design too takes places at three levels:

- Designing the set of possible semantic objects.

- Designing the abstract syntax to represent the semantic objects.
- Designing the concrete syntax to encode the abstract syntax.

And since the decisions made at each level depend on decisions made at the previous level, language design logically proceeds in a top-down manner.

The primary goal of a particular language design, then, is to represent a particular semantics in the simplest and most flexible way. To further this goal:

- The complexity of the semantic objects should be kept to a minimum.
- The abstract syntax should mirror the structure of the semantic objects.
- The concrete syntax should mirror the structure of the abstract syntax.

Now the semantic objects of a programming language vary considerably depending on whether the language is functional, imperative, relational, object-oriented, and so on. Despite that, there are techniques applicable to any programming language for representing the semantic objects as syntactic objects and for encoding the syntactic objects as sequences of symbols [AU72,AU73,AU77]. And the manipulation characteristics of the language are strongly influenced by the nature of these techniques.

After all, the semantics of a program is derived from its syntactic structure, so meaningful manipulation is best supported by directly manipulating this structure. And indeed, existing high-level manipulation tools invariably represent programs as abstract syntax trees and carry out manipulation in terms of these syntax trees. Examples of such systems are Mentor [DGKLM84], PSG [BS86], and the Cornell Program Synthesizer [TR81,TRH81,Rep84,RT84, RT89]. To advance these high-level manipulation tools, a well-designed language should specify an abstract syntax that enhances a programmer's ability to synthesize, analyze, and transform programs.

The design of a lexical encoding of abstract syntax, although a secondary concern, is not immaterial however, for abstract syntax can only be viewed in terms of a concrete encoding. Nevertheless, an encoding is simply one possible view; many alternative views could be defined for various different purposes. Only when underlying abstract syntactic structure is emphasized is it possible to consider the provision of such alternative views.

Moreover, the advent of high-resolution bitmapped displays and high-quality typeset printing provides exciting possibilities for producing more readable views. The notion of literate programming [Knu84], which advocates an integrated approach to program construction and documentation, benefits greatly from these technological advances: various font cues, such as bold-face keywords and italic identifiers, can dramatically improve a program's readability.

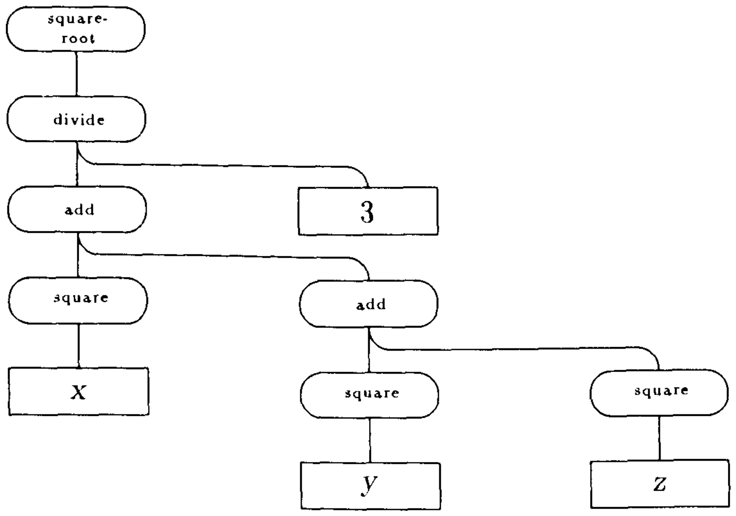
In this light, a language designer, ignoring the inherent limitations of ASCII, should first design the abstract syntax of a language and should only then design the portable ASCII view of that syntax. The designer should then consider the possibility of defining more readable views that make better use of improving display technologies. For example, a mathematical expression encoded in ASCII as

```
sqrt((sqr(x) + sqr(y) + sqr(z))/3)
```

could be more readably viewed using mathematical notation as

$$\sqrt{\frac{x^2 + y^2 + z^2}{3}}$$

and could even be viewed as a syntax tree:



A program, then, is a syntactic object, not merely a sequence of ASCII symbols. This perspective echoes the first sentence of [TRH81], which states: "Programs are not text; they are hierarchical compositions of computational structures ..."

It is well-established that programs should be viewed abstractly as syntax trees consisting of nodes. What is not so well-established is that this abstract view should be specified during language design. Doing so ensures that different tools, and even different language implementations, support the same abstract view and that tools can be integrated into a unified supporting environment [DMS84]. Since high-level support tools invariably implement manipulation in terms of a syntactic representation, a standard abstract view should clearly be specified as part of a language definition.

The definition of DIANA [GWEB83], an intermediate representation for Ada [Ada83], demonstrates recognition of the fact that a standard abstract representation for programs as data objects is an important goal. An example of its use is presented in [Ros85]. Unfortunately, DIANA was designed almost as an afterthought; its design did not influence the design of Ada. As a result, DIANA is complex because it must handle all the details of Ada's concrete syntax.

The emphasis on abstract syntax as the basis for manipulation can even be taken one step further by insisting that all aspects of a programming language's static semantics be defined in terms of nodes and relations on nodes. As such, *all* static semantic objects are represented as nodes and hence a program is just a set of related nodes. For instance, each identifier node in a program will have a related node that represents its defining-occurrence and each expression node will have a related node that represents its type. Dyck referred this use of nodes for representing both programs and their derived semantic attributes as the *double-duty* strategy [Dyc90]; he applies the strategy to define the complete static semantics of ISO Pascal [ISO83], thereby demonstrating its feasibility. I used the strategy in earlier work [Mer87] involving the source-to-source compilation of Modula-2 [Wir85].

To clarify any misconceptions, the term *static* semantics draws a distinction between semantic aspects that are static in nature, such as scoping and typing, and semantic aspects that are dynamic in nature, such as activations, continuations, allocations, and so on. Although the distinction between static and dynamic semantics is clear, the distinction between static semantics and syntax is blurred. Both deal with constraints on node structure: syntax deals with context-free constraints and static semantics deals with context-dependent constraints. For example, in the Pascal if-statement

IF *condition* **THEN** *action*

syntax constrains *condition* to be an expression, while static semantics constrains *condition* to be of type Boolean. This blurring of static semantics and syntax is desirable because it allows both notions to be handled uniformly through the representation of nodes.

In this thesis, "context-free" syntax is used to describe the tree relations specified by a context-free grammar and "context-dependent" syntax is used to describe the static-semantic relations derived from the tree relations. "Syntax" thus includes static semantics.

1.2 The design of Acer

To bring substance to this exploration of language design, I will demonstrate the design of a programming language called Acer. Like all language designs, the design of Acer begins with requirements. In this case, Acer is to be a more manipulable version of Quest [Car89]. I chose Quest, a complex imperative language with an elaborate type system, so that I would encounter, with the design of Acer, all the technical problems involved in designing a full-scale language. Furthermore, the richness of Quest's semantic structure provides many inherent manipulation opportunities and issues not present in other contexts. As well, Quest conforms to many of the design principles discussed in Chapter 4. Realize, however, that the requirements for a programming language are often beyond the control of the language designer. In this sense, the choice of Quest is an arbitrary one.

To demonstrate the manipulation advantages of Acer, I have implemented an environment for Acer, called PCAcer, consisting of a language-based editor and a compiler. This environment, described in detail in Appendix B, was implemented using a GRAMPS-style metaprogramming system [CI84], a tool that enables programmers to write high-level tools and transformations in terms of an abstract representation of programs. In many ways, designing a language that supports manipulation is synonymous with designing a language that has a simple metaprogramming system. Details are presented in Chapter 5.

The remainder of the thesis is organized as follows. Chapter 2 presents an overview of Acer, outlining its features and how they support manipulation. Chapter 3 explores techniques for specifying syntax, including a modified GRAMPS-style approach for specifying context-free syntax as well as a relational approach for specifying context-dependent syntax. Chapter 4 discusses manipulation principles for guiding language design, and explains how they are applied to the design of Acer—these principles were also presented in [MDC92]. Chapter 5 describes Acer's metaprogramming system and shows how it is used to facilitate manipulation. Finally, Chapter 6 summarizes the results of the thesis, relating it to the work of others and suggesting directions for further research.

Additional information is provided in the form of appendices. Appendix A gives a precise specification of Acer, providing more detail than the overview of Chapter 2. Appendix B is the user's manual for PCAcer, Acer's language-based environment. Appendix C is the implementor's manual for Acer's metaprogramming system and its environment. And Appendix D provides quick reference tables for Acer's manipulation primitives and Acer's type system.

Much effort has gone into the design and implementation of PCAcer in order to demonstrate the efficacy and feasibility of the ideas presented. However, PCAcer is not intended to be innovative and so is not described in the main body of the thesis. Readers are nevertheless urged to read the user's manual of Appendix B to get a feel for how Acer's semantic and syntactic framework enhances the manipulation support provided by a programming environment.

PCAcer is a simple, yet powerful, programming environment. It is a language-based environment, according to the Dart taxonomy [DEFH87], and an instance of an individual model programming environment, in the terminology of [PK91]. It uses the hybrid approach to language-based editing described in [BS86], providing both textual and structural views with easy conversion between the two.

Chapter 2

An introduction to Acer

This chapter presents an informal overview of Acer. It is intended as a more intuitive introduction than the detailed definition in Appendix A. However, the reader is strongly urged to consult Appendix A to clarify questions concerning syntax that might arise, particularly section A.34, which summarizes Acer's grammar. The grammar formalism itself is described in Chapter 3. Appendix D provides quick reference tables.

Throughout this thesis, hyphenated terms, such as binding-list, defined-identifier, and defining-occurrence, are used to refer to concepts with a particular Acer meaning, usually as a syntactic element or attribute. It would be more precise to use instead the terms binding-list construct, defined-identifier component, and defining-occurrence attribute. However, to avoid being overly pedantic, this more precise terminology is used only if confusion would arise in its absence.

2.1 Conceptual foundations

Before outlining Acer, consider the general categories of semantic object that are common to programming languages and the implications of these categories on support for program manipulation. Ideally, a language should support a simple and uniform model of semantic objects and their corresponding syntactic representations, for this simplicity and uniformity are key in supporting straightforward formal manipulation without a myriad of special cases.

2.1.1 Value

In general, programs manipulate abstract entities, i.e., *values*, according to concrete representations, i.e., as bits in memory. The details of the concrete representation are unimportant however, only abstract behavior matters. Therefore, the semantics of a programming language is concerned primarily with values, not with particular representations of values.

However, when manipulating a program, it is often necessary to manipulate the values to which the program refers. Hence, to support manipulation, it is desirable that there exist for every value, a corresponding expression (i.e., a syntactic literal) to denote it. In other words, each value should have a syntactic representation, so it can be expressed and manipulated as part of a source program.

As a counter-example, consider the way records are supported in Pascal. A variable of a record-type can be declared and the fields of that variable can be initialized via assignment, but there is no concept of a record-literal, i.e., a syntactic expression that denotes a record with particular values for its fields. Hence, although Pascal supports manipulation of records as run-time values, it does not support manipulation of records as syntactic objects. Yet supporting environments must often present run-time values in a human readable form, such as when debugging a program, so the need for a syntactic representation for each class of value inevitably arises.

With a well-designed expression syntax for literals, there is a blurred distinction between a semantic value and a syntactic expression that denotes it. After all, a semantic value must be represented in some particular way; when that representation closely mirrors the structure of the semantic value, the semantic value and its representation are easily confused. For example, drawing a distinction between the number ten and the numeral 10 may seem like semantic nit-picking. But it is this distinction which allows semantic manipulation (e.g., the sum of two numbers) to be achieved by syntactic manipulation (e.g., by a formal process involving adding the digits of the two numerals). It is also what allows the number ten to be concretely represented as the base two numeral 1010 in memory.

To generalize, then, for any particular programming language, *value* is the ground-level category of semantic object, and to support program manipulation, a corresponding category of syntactic object should exist. The uniform treatment of abstract entities as values and the provision of syntactic representations for those values is the best way to support the syntactic manipulation of programs and the values they use. It is encouraging, therefore, that the trend toward more expressive programming languages, which include a richer variety of first-class values (e.g., Quest provides first-class functions and first-class modules), is also a trend toward enhanced support for program manipulation, provided syntax is appropriately designed.

2.1.2 Type

Programming languages generally organize their value space according to *type*, giving rise to the type-level category of semantic object. Since manipulating a program involves manipulating the types to which the program refers, it is desirable if there exists for every type, just as for every value, a corresponding expression (i.e., syntactic literal) to denote it.

When striving to achieve an expressive programming language, there may be a temptation to treat types as values, as does Poly [Har84], but this temptation should be avoided. Program manipulation is best supported for programs with strong static constraints. This should not hinder expressiveness though because dynamic typing can be supported within the framework of a statically typed language [ACPP91,CF91].

To generalize, then, for any particular programming language, values, the ground-level category of semantic object, are organized according to type, the type-level category of semantic object. And, just as for values, a corresponding type-level category of syntactic object should exist.

2.1.3 Kind

Just as programming languages organize their value space according to *type*, so too a programming language can organize its type space according to *kind*, giving rise to a kind-level category of semantic object and a corresponding kind-level category of syntactic object. Quest has such a type system.

This thesis will demonstrate, however, that the inclusion of a syntactic representation for kinds is unnecessary. Acer has a type system analogous to that of Quest, but Acer provides only syntactic representations for values and types. The notion that types are typed is retained, but the type of a type, in Acer, is a type rather than a kind. Manipulation is enhanced by an economy of syntax.

2.1.4 Additional syntactic categories

In an *ideal* programming language, then, each category of semantic object gives rise to a corresponding category of syntactic object. And to enhance manipulation of the semantic objects, there should be a simple direct mapping between semantics and syntax. Thus, in an ideal programming language, a program's semantic objects are mirrored by the structure of its syntactic objects.

Now, because a programming language is used to specify values in terms of computations, the syntactic category for value (expression) will include additional syntactic classes corresponding to various value-yielding semantic operations, e.g., function-calls, conditionals, and so on. As well, because not all semantic operations are naturally value-yielding, syntactic categories disjoint from value may also occur, such as the category *statement* in Pascal. Nevertheless, in an expression-oriented language [vWMP+76,KR78], semantic operations that are not naturally value-yielding may be interpreted as yielding instead a special type of value indicating a void result. Thus all semantic operations can be treated uniformly as expressions, although some are evaluated purely for side-effect. This uniform treatment enhances support for manipulation.

Another notion common to programming languages is the notion of scope: an identifier is associated with a semantic object and is visible over some region in which it is used to stand in place of that semantic object. Static scoping with nested block structure is widely accepted as the preferred approach for dealing with scope.

According to the terminology used in this thesis, an identifier denoting a semantic object (i.e., a value or type) can be introduced by either a *binding* or a *declaration*. The distinction between the two is that a binding construct associates with its identifier a particular semantic object, but a declaration specifies only the object's type—the actual semantic object is then associated with the identifier dynamically, either via parameter binding or via assignment. To reiterate then, a binding construct initializes the identifier it introduces but a declaration specifies only the identifier's type.

2.1.5 Summary

To summarize the above, the semantic objects of a programming languages can be categorized according to the ground-level category, which includes values; the type-level category, which includes types; and the semantic-operation category, which includes computational and scoping structures.

Accordingly, the syntactic objects of a programming language can be categorized according to the value category, which includes value literals and value-yielding semantic operations; the type category, which includes type literals and type-yielding semantic operations; the binding category, which includes type- and value-bindings; the declaration category, which includes type- and value-declarations; and the miscellaneous category, which includes all those syntactic objects that serve merely as components of semantically complete syntactic objects (e.g., just as an argument-list serves merely as a component of a function-call and a signature serves merely as a component of a function-type).

This conceptual framework is the basis for Acer's design and mirrors the underlying framework of many programming languages.

2.2 Onward

Acer's design is based on that of Quest [Car89], an imperative, general-purpose, expression-oriented language with an elaborate type system that supports type quantification and structural subtyping [CW85]. Like Quest, Acer features abstract types, polymorphism, single and multiple "inheritance," and garbage collection. Acer differs from Quest in significant ways, however.

Most importantly, Quest has three levels of semantic objects (values, types, and kinds), and Acer has only two (values and types) because kinds are represented as types. To understand how this is possible one must realize that Quest's subtyping rules induce a lattice on types (i.e., a type hierarchy), and that a kind is simply a sublattice of this lattice. Intuitively, kinds classify types just as types classify values. Thus, in Quest one says, "The type T is an element of kind K."

Now, a kind, being a sublattice, can always be denoted by its root type, i.e., the sublattice's maximum type. Therefore, in Acer, each of Quest's three ways of specifying a kind is replaced by a type as follows:

- Instead of providing the kind **TYPE**, which Quest uses to denote the whole type lattice, Acer provides the any-type **Any**, which it uses to denote the root of the type lattice.

- Instead of providing the power-kind **POWER**(T), which Quest uses to denote the sublattice rooted at type T, Acer provides just T itself.

- And finally, instead of providing an operator-kind **OPER**() K, which Quest uses to denote the sublattice rooted at a type-operator[1] **Oper**() T, where the type T is an element of kind K, Acer provides just the type-operator itself (see 2.9).

[1]A type-operator is a function from types to types.

Therefore, in Acer, type expressions are used to denote kinds. Also, since the kind of a type expression is a type, it is as natural to refer to the type of a type expression as it is to refer to the type of a value expression. Thus type expressions and value expressions are jointly referred to as simply expressions. And we can speak of the type of an expression, regardless of whether it denotes a type or a value. So, in Acer, when we say that a type T is of type K we mean that T is a subtype of K.

Acer also includes a number of features not present in Quest. In particular, features for supporting high-level iteration and accumulation are provided (see 2.16), as are features for supporting high-level notations applicable to abstract-types (see 2.7).

Eliminating kinds from the syntax and introducing additional constructs significantly alters Quest. Quest's syntax is further altered in accordance with the GRAMPS approach and the relational approach, as described in Chapter 3, and to conform with the design principles described in Chapter 4. For example, in Acer, type and value expressions are to be syntactically distinguishable. Therefore, a type-identifier must start with an upper-case letter and a value-identifier must not. (An underscore is considered an upper-case letter.) Besides letters and digits, a value-identifier may also be spelled using a combination of the following:

```
! # $ & * + - / < = > ? \ ^ | ~
```

Here are some examples of type-identifiers:

Type   *T0*   *Base_Type*   _

and here are some examples of value-identifiers:

value   *v_0*   ++   \/   <=

As an additional font cue for distinguishing type-identifiers from value-identifiers, type-identifiers are shown *Italic* and value-identifiers are shown *slanted.* Also, keywords are shown **bold** to distinguish them from identifiers; comments, which start with a '%' and end at the end of the line, are shown in Roman; and character- and string-literals (see 2.8) are shown in `typewriter`.

Acer's syntax includes a number of unusual features. First of all, several constructs contain unbalanced brackets (e.g., '(', '{', and '[') that are balanced by the matching bracket of a different construct. For example, a binding-list begins with a '{' followed by a series of binding constructs but no terminating '}.' However, a binding-list in context can appear only as the initial component of a block, and a block provides the terminating '}.' Thus, although a construct out of context may have unbalanced brackets, brackets are always balanced for constructs in context. The justification for this approach stems from the notion of avoiding phrase ambiguity as discussed in section 4.3.

Another unusual aspect of Acer's syntax, which closely mirrors that of Quest, is the fact that all commas and semicolons are optional. Acer's syntax is designed so that it is always clear where a given construct ends and a new one begins; hence separators and terminators are redundant. Throughout this thesis, commas and semicolons are generally included only to separate or terminate constructs appearing on the same line; or when the example is intended to indicate appropriate punctuation.

Acer's scope rules follow the traditional style of nested block structure, but forward-reference among declarations and bindings is generally permitted. This provides direct support for recursion. Restrictions on forward-references to values prevents access to uninitialized values (see A.7.2). To begin this overview of Acer, consider the way value- and type-identifiers are introduced at the top level, that is, with global scope.

2.3 Global identifier

The most basic Acer program consists simply of a fixed-value-binding, for example,

let *pi* **be** 3.14159

introduces a location named *pi* of type *Real* containing an approximate representation of the mathematical value π. Compiling[2] this binding produces the fixed-value-declaration

pi : *Real*

which is then globally visible to subsequent compilations.

An arbitrary number of global value-identifiers can be simultaneously introduced in the form of a binding-list. For example, compiling the binding-list

{**let** *x* **be** *y*;
 let *y* **be** 10;

produces two declarations:

x : *Integer*

[2]Of course, compiling a binding produces object code in addition to producing a declaration.

and

y : *Integer*

A binding-list begins with a left brace and the bindings are terminated by optional semicolons; there is no terminating right brace. Top-level binding-lists typically are used only for introducing mutually dependent values, a situation that arises only infrequently. The above values are not mutually dependent since *y* could be compiled separately before x.

A global type-identifier is introduced by a type-binding, for example,

let *Color* **be Enumeration** *red*, *blue*, *green* **end**

introduces an enumeration-type named *Color*. Unlike a fixed-value-binding, which must be compiled to produce the declaration that becomes visible, a type-binding is directly visible without prior compilation.

To understand the reason for the different handling of global types and values, realize that Acer supports information hiding by introducing a global value-identifier with a declaration, which indicates the identifier's type but hides its implementation. Such a top-level declaration, as we shall see in section 2.7, is analogous to a Modula-2 definition module [Wir85]; the binding from which it is derived is analogous to a Modula-2 implementation module. Thus, in Modula-2 terms, one could think of it as if compiling an implementation module (a binding) automatically produces a definition module (a declaration).

Of course, in above example, there is little reason to hide the value associated with *pi*, except maybe to hide the number of significant digits supported by the implementation of *Real*. But then, the example is intended to be a simple one, not an example that shows the need for information hiding. Had the binding introduced a function, the effect would have been to declare a value-identifier with a function-type, thereby hiding the function's implementation.

Hiding of type implementations is supported in similar manner using type-declarations, for example,

Color :: **Any**

A type-declaration indicates the kind of type bound to its identifier but hides the actual implementation, which may be any subtype of the indicated type. However, hiding of a type implementation occurs only in the context of a quantifier, which provides the operations that apply to values of the hidden type (see 2.7 and A.20.3). Hence, top-level type-declarations are not supported. Type hiding will be discussed further with respect to type-declarations in function signatures (see 2.6) and with respect to type-declarations in tuple-types (see 2.7).

To summarize then, each top-level value-identifier is introduced by a fixed-value-declaration, for which there presumably exists a fixed-value-binding that provides the value's definition, i.e., its hidden implementation; and each top-level type-identifier is introduced by a type-binding that provides the type's definition. In this way, any value or type can be introduced at the top level to make it globally visible.

Acer economizes on notation by avoiding special top-level constructs, such as modules, interfaces, packages, and so on. Explicit imports and exports are also avoided. Acer simply makes do with nested block structure and existing constructs, such as bindings and declarations, which serve other roles in the language as well. Such economy simplifies manipulation.

2.4 Block

Binding-lists are more frequently used in conjunction with blocks so that bindings can be introduced local to an expression. A block consists of a binding-list and a body and is either a type-block, e.g.,

{**let** *T* **be** *Integer*; *T*}

or value-block, e.g.,

{**let** *x* **be** 10; **let** *y* **be** 20; {*x* + *y*}}

depending on whether the body is a type or a value. Remember, type and value expressions are to be syntactically distinguishable. Notice the use of a dyadic-method-call {*x* + *y*}, which is similar in appearance to a block, to invoke the *Integer* addition operation. Method-calls are discussed in detail in section 2.7.

A binding in a type-block's binding-list must be a type-binding, but a binding in a value-block's binding-list may be a type-, fixed-value-, or variable-value-binding. A variable-value-binding is directly analogous to a fixed-value-binding except for the additional **var** indicator, e.g.,

let var *x* **be** 0

A variable-value-binding introduces an updatable location, initialized as indicated, which may subsequently be modified by assignment, e.g.,

{x **becomes** 10}

The syntax of an assignment expression mirrors that of a dyadic-method-call. The type of an assignment expression is *Void*, the type of a value expression evaluated for side-effect; the void-literal {} denotes the one value of type *Void*, and this is the value yielded by an assignment.

When a value-block is evaluated, the bindings are evaluated first and the body is evaluated in the context of those bindings to yield the result of the block. The bindings themselves can be evaluated in any order that does not access uninitialized values. For example, in the block

{**let** x **be** f(); **let** y **be** g(); {$x + y$}}

either f or g can be evaluated first. Nevertheless, a left-to-right evaluation must be a correct evaluation order as determined by *dependency analysis* (see A.7.2), and bindings may well be evaluated in that order. For example, the block

{**let** x **be** $h(y)$; **let** y **be** 42; {$x + y$}}

is incorrect because it cannot be evaluated left-to-right. So to put it another way, the dependencies between bindings must be reflected in their left-to-right order, but when bindings do not depend on one another, they can be evaluated in any order. This provides a measure of freedom to the compiler implementor.

So that side-effects can be sequenced, Acer provides a compound-value, which is a list of value expressions bracketed by **begin end** and separated by optional semicolons, for example,

begin f(); g(); h() **end**

A compound-value is evaluated left-to-right yielding the value yielded by the last expression. If any evaluation raises an exception (see 2.12), the compound-value also raises that exception. The empty compound-value is equivalent to the void-literal.

Acer supports local program transformations by providing blocks that can be introduced local to any expression context. Hence, new objects can be made visible over precisely the region in which they are required. See section 4.9 for further discussion of this issue.

2.5 Enumeration

Acer provides the enumeration-type construct for defining enumerations. For example, the binding

let *Color* **be Enumeration** *red*, *blue*, *green* **end**

introduces an enumeration-type named *Color*. The values of type *Color* can be denoted using literal-selection, e.g., *Color.red*. The commas separating the identifiers of an enumeration-type are optional.

Enumeration-values are ordered and hence an ord-call construct is provided to determine the ordinal position of an enumeration-value, e.g., **ord** (*Color.red*) yields the *Integer* value 0. In addition, given an ordinal position, the corresponding enumeration-value can be obtained with a val-call, e.g., **val** (*Color*, 0) yields the value *Color.red*. Enumeration-values can also be used in Acer's variant-inspection as described in section 2.13.

Notice that the identifiers of an enumeration-type are not visible without the qualification of a literal-selection. This enhances manipulation by keeping separate name spaces disjoint. See section 4.8 for a discussion concerning the impact of scope rules on manipulation.

2.6 Function and function-call

A function in Acer is defined using a function-literal. For instance, a function named *f* can be introduced by a binding containing a function-literal as

let *f* **be function** (*Type* :: **Any**; *value*: *Type*) *value* **end**

where

(*Type* :: **Any**; *value*: *Type*)

is called the signature. The function denoted by *f*, which is the polymorphic identity function, can then be called as

f (*Integer*, 10)

or equivalently as

f (let *Type* be *Integer*, let *value* be 10) to yield the *Integer* value 10. In general, an argument-list is similar to a binding-list except that an argument-list comprises a list of arguments (i.e., expressions and bindings) that begins with a left parenthesis and is separated by optional commas. Just as the left brace of a binding-list is balanced by the right brace of a block, so too the left parenthesis of an argument-list is balanced by the right parenthesis of a function-call.

When a function-call is evaluated, the arguments are bound to the declarations of the function's signature and then the function's body is evaluated in the context of those bindings to yield the result of the call.

As a bit of an aside, Acer provides **is** and **isnot** to test for value-identity. Therefore, because *f* is the polymorphic identity function, the is-test

```
{x is f (TYPE(x), x)}
```

always yields *true*, regardless of the type of x. Hence, even

```
{f is f (TYPE(f), f)}
```

is valid and yields *true*. Notice how the type-designation **TYPE**(x) is used to denote the type of x. In general, a type-designation can be applied to any expression to denote the type of that expression.

The type of a function-literal is a function-type. For example, the type of *f* (i.e., the type denoted by **TYPE**(*f*)) is the function-type

```
Function (Type :: Any; value : Type) Type end
```

The result-type indicated by a function-type can be explicitly indicated in a function-literal. For example, *f* above could be equivalently given as

```
let f be function (Type :: Any; value: Type): Type value end
```

Notice how the initial lower-case or upper-case letter in **function** and **Function** distinguishes a function-literal from a function-type, just as it distinguishes a value-identifier from a type-identifier. This style is used often in Acer.

The use of type-declarations in signatures supports polymorphism. For example, the function *max* in

```
let max be
  function (BaseType :: Any
              <: Function (a: BaseType; b: BaseType) Boolean end
              x : BaseType
              y : BaseType)
    if {x < y} then y else x end
  end
```

defines a polymorphic function that takes four parameters, a base-type of type **Any**, a less-than function that applies to values of the base-type, and two values of the base-type. The function *max* yields the maximum of the two values of the base-type as determined by the less-than function. Notice the use of Acer's conditional expression (see A.23.3), which can in general include **elsif** clauses, and also the use of a dyadic-method-call to call the less-than function. Method-calls are described in the section 2.7.

The type of *max* is

```
Function (BaseType :: Any
          < : Function (a :BaseType; b : BaseType) Boolean end
          x : BaseType
          y : BaseType)
  BaseType
 end
```

and the appearance of *BaseType* as the result-type demonstrates that the type of a function-call can depend on its type parameters. For example, the function *max* could be called as

```
max (Integer, integer.<, 10, 20)
```

or equivalently as

```
max (let BaseType be Integer
     let < be integer.<
     let x be 10
     let y be 20)
```

to yield the *Integer* value 20. Notice how *integer.<* is used to select the less-than function from the *integer* module. Section 2.7 explains how such modules are defined in terms of tuples.

The function *max* could be applied to a different type of argument as well, e.g.,

```
max (Real, real.<, 1.2, 10.2)
```

yields the *Real* value 10.2.

A function cannot ascertain the actual types bound in a function-call to the type-identifiers declared in its signature so each of these identifiers is considered to be an *abstract-type*. In other words, a type is an abstract-type if it is introduced by a declaration. (Note that the type bound to a type-parameter cannot be determined even at run-time since typing in Acer is static, except for dynamics which are described in section 2.15.) The values of each abstract-type in a function signature, and the operations that apply to those values, must be declared in the same signature; hence the signature quantifies the abstract-type (see A.20.3). For example, in the function *max* the abstract-type *BaseType* is declared along with two values of that type and a function applying to those values.

Recall that the introduction explained that in Acer each type expression has a type just as each value expression has a type. Normally, for a concrete-type, such as a function-type or an enumeration-type, a type expression is its own type. But for an abstract-type, such as *BaseType*, its type is given by its type-declaration. (Here we see the double-duty strategy in action.) At run-time, all that is known about a given abstract-type is that it is a subtype of the indicated type. In the function *max*, the type of *BaseType* is **Any** and hence *BaseType* is unrestricted. In other cases, the type of an abstract-type is something other than **Any** such as a function-type or an enumeration-type and hence is partially restricted. Specifying such restrictions involves Acer's subtyping rules, which will be described in section 2.14.

When a function-literal is evaluated, as opposed to when a function-call is evaluated, a *closure*, i.e., the set of all values used by the literal, is computed. At run-time, a function-value therefore contains both a reference to the instructions for evaluating the literal's body and a reference to the closure. For example,

```
let h be function (x : Integer)
            {let var y be x
             function ()
               begin {y becomes {y + 1}}; y end
              end}
            end
```

defines a function *h* that returns a locally defined function with the updatable location *y* in its closure. Note that *y* is allocated on the heap. The type of *h* is

Function (*x* : *Integer*) **Function** () *Integer* **end end**

and hence the type of *h*(0) is

Function () *Integer* **end**

Successive calls to the function yielded by *h*(0) yield succesive *Integer* values starting with 1.

Acer's functions support manipulation by allowing any value expression to be abstracted over (see 4.14). In addition, functions are first-class values and can thus be used just like any other value: they can be passed as parameters, stored in data structures, and returned from functions.

Like Quest, Acer supports polymorphism through the use of explicit type-parameters, an approach that is substantially different from approaches that infer the type-parameters from the value-parameters. Such a type-inference approach is the basis for supporting polymorphism in ML [Mil78] (or see [Wik87, pages 378–420]). A more extensive approach, which involves inferring not only the type-parameters but also the semantic operations associated with the type-parameters, is described in [CW90]. Explicit polymorphism has the advantage of being easier to define and support, but there are notational advantages to the type-inference approach, terseness of notation being one of the primary advantages. A limited form of type-inference is supported in Acer through the use of method-calls (see 2.7).

2.7 Tuple, method, and selection

The tuple is Acer's primary data structuring mechanism and subsumes the conventional notion of modules. A tuple-literal, like an argument-list, is a list of arguments separated by optional commas. But in this case, the list is bracketed by **tuple end**. The type of a tuple-literal is a tuple-type, which is analogous to a signature except that the declarations are bracketed by **Tuple end**. For example,

tuple 10, 10.0 **end**

has type

Tuple : *Integer*; : *Real* **end**

Notice that the defined-identifier of a declaration is optional so that a declaration can be derived from an expression as well as from a binding. This supports the declaration of anonymous parameters and anonymous data-structure components. Notice too that type- and fixed-value-declarations are distinguished by the fact that a type-declaration uses two colons rather than one. Thus the type-declaration :: **Any** is distinguishable from the fixed-value-declaration : **Any**.

Tuples can introduce types as well as values, e.g.,

```
let t be
  tuple
    let Type be Integer
    let value: Type be 0
    let operation be function (x : Type): Type {x + 1} end
  end
```

which has type

```
Tuple
  Type :: Any
  value : Type
  operation : Function (x : Type) Type end
 end
```

Note how a binding explicitly indicates a type, as in the binding for *value*, to specify the type of the derived declaration. The components of *t* are accessed using either type-selection (e.g., *t.Type*) or value-selection (e.g., *t.value* and *t.operation*).

The type-selection *t.Type* is an abstract-type because the actual type used in the definition of *t* is inaccessible. For instance, we might have defined *t* as

```
let t be
  tuple
    let Type be Real
    let value : Type be -10000.0
    let operation be function (x : Type) : Type {x + 10.0} end
   end
```

In either case, *t* introduces an abstract-type *t.Type* of type **Any**, a single value *t.value* of that type, and a single function *t.operation* that applies to values of that type. Thus the tuple *t* acts as a module implementing an abstract-type and to support information hiding, the abstract-type's definition is inaccessible.

The reader may be unsettled that *t.value* might contain any type of value whatsoever. This is of no concern, however, because each abstract-type is unique so *t.value* can only be used as a value of *t.Type*. Hence, it can only be passed to the function *t.operation* by the call *t.operation*(*t.value*) to yield another value of type *t.Type*.

Consider now an extended example in which we define an abstract-type for simple three-dimensional vectors. Compiling the binding

```
let vector be
  tuple
    let Type :: Any be Tuple x : Real; y : Real; z : Real end
    let construct be
      function (x : Real; y : Real; z : Real) : Type
        tuple x, y, z end
       end
    let x be function (v : Type) : Real v.x end
    let y be function (v : Type) : Real v.y end
    let z be function (v : Type) : Real v.z end
    let + be
      function (v1 : Type; v2 : Type): Type
        tuple {v1.x + v2.x}, {v1.y + v2.y}, {v1.z + v2.z} end
       end
    let - be
      function (v1 : Type; v2 : Type): Type
        tuple {v1.x - v2.x}, {v1.y - v2.y}, {v1.z - v2.z} end
       end
    let ~ be
      function (v : Type): Type
        tuple {~ v.x}, {~ v.y}, {~ v.z} end
       end
    end
```

produces the declaration

```
vector : Tuple
           Type :: Any
           construct : Function (x : Real; y : Real; z : Real) Type end
           x : Function (v : Type) Real end
           y : Function (v : Type) Real end
           z : Function (v : Type) Real end
           + : Function (v1 : Type; v2 : Type) Type end
           - : Function (v1 : Type; v2 : Type) Type end
           ~ : Function (v : Type) Type end
          end
```

The abstract-type *vector.Type* allows us to illustrate several significant aspects of Acer.

First of all, notice that Acer cannot support overloading because the identifiers defined by a tuple must be distinct. Since '-' is used for subtraction of *vector.Type* values, a different symbol, namely '~', must be used for negation. Acer's *Integer* and *Real* are also defined in terms of abstract-types and so they too use '~' for negation, which can be called using a unary-method-call, e.g., {~ 10}. However, integer- and real-literals can still use the conventional notation (e.g., -10 and -10.0) since in this case '-' is considered part of the literal.

Given the values *v1* and *v2* of type *vector.Type*, which could be created as

> **let** *v1* **be** *vector.construct* (-10, 20, -30)

and

> **let** *v2* **be** *vector.construct* (10, -20, 30)

we could add the two values using *vector.+* (*v1*, *v2*). We could negate *v1* using *vector.˜* (*v1*). Also, we could access the *x* coordinate of *v1* using *vector.x* (*v1*). These notations are rather verbose so Acer provides method-calls as a short-hand notation for function-calls (see A.21).

A prefix-method-call is the most general form of method-call. It looks like a function-call except that the function must be a value-identifier and the argument-list is preceded by a dot, e.g., +.(*v1*, *v2*). The function-call equivalent of a given prefix-method-call is determined by searching the operations associated with each successive argument that has an abstract-type. Hence, +.(*v1*, *v2*) is equivalent to *vector.+* (*v1*, *v2*) because the type of *v1* is *vector.Type* and *vector* provides a '+' component. A dot is used in the syntax for prefix-method-calls so that a prefix-method-call is distinct from a function-call and so that one is reminded that a prefix-method-call involves value-selection.

Acer defines both unary- and dyadic-method-calls in terms of prefix-method-calls, for example, {˜ *v1*} is equivalent to ˜.(*v1*) and {*v1* - *v2*} is equivalent to -.(*v1*, *v2*). Also, when value-selection is applied to a value with an abstract-type, it is treated as equivalent to a prefix-method-call. By doing so, Acer allows values with abstract-types to support the familiar dot-notation for selecting components. For example, the *x* coordinate of the *vector.Type* value *v1* could be accessed using the (abstract) value-selection *v1.x* which is equivalent by rewriting to the prefix-method-call *x*.(*v1*) which in turn is equivalent after method-lookup to the function-call *vector.x* (*v1*).

To complete the discussion of method-calls, consider one last Acer construct that is defined in terms of a prefix-method-call, namely an index expression. An index expression such as a[*i*, *j*] is defined to be equivalent to *index2*.(a, *i*, *j*); hence any value with an abstract-type can support the notation for subscripting. Notice that the method-name used in the prefix-method-call is of the form *indexn*, where *n* indicates the number of indexing values, which may be zero or arbitrarily many.

Abstract-types can also support assignable value-selection and indexing by using the rule that if the prefix-method-call equivalent *method*.(*arguments*) of a given value-selection or index has a *Pointer* or *Reference* type then the prefix-method-call equivalent is instead *method*.(*arguments*)@. In other words, an abstract value-selection or index expression that yields a reference or pointer is automatically dereferenced and hence the resulting expression may occur as the destination of an assignment. We shall see examples of this in the discussion of pointer- and reference-types in section 2.10.

Acer's tuples support manipulation by providing modules as first-class values: they can be passed as parameters, stored in data structures, and returned from functions.

2.8 The standard abstract-types

Acer provides special syntactic support for five basic types, namely *Integer*, *Real*, *Character*, *String*, and *Boolean*. Support for the first four types is provided in the form of literals. We have already seen examples of integer- and real-literals. A character-literal is of the form 'a' and a string-literal is of the form "abc". No additional special support is provided for these four types, each of which is defined in terms of an abstract-type encapsulated by a tuple.

Special syntactic support is also provided for the type *Boolean*, but not in the form of literals since these are denoted as *boolean.false* and *boolean.true*, or simply as *false* and *true* (see A.23). Syntactic support for *Boolean* values is provided in the form of short-circuit evaluation constructs for evaluating 'and' and 'or.' In particular, an andif-test

{*x* **andif** *y*}

is equivalent to

if *x* **then** *y* **else** *false* **end**

and similarly an orif-test

{*x* **orif** *y*}

is equivalent to

if *x* **then** *true* **else** *y* **end**

Although the andif-test and orif-test have the appearance of a dyadic-method-call, they cannot be implemented as such (by the *boolean* module, say) because both arguments would be evaluated before the method were called.

Acer's standard abstract-types receive no additional syntactic support. Acer attempts to provide general support for abstraction rather than attempting to provide specific support for a number of special built-in abstractions. This is in keeping with providing support for general manipulation.

2.9 Type-operator and operator-call

Type-operators are functions from types to types that are evaluated statically, i.e., at compile-time. For example, a type-operator can be introduced as

```
let Dyadic be
  Operator (BaseType :: Any; ResultType :: Any)
    Function ( : BaseType;  : BaseType) ResultType end
   end
```

This type-operator can then be called using an operator-call as

```
Dyadic (Integer, Boolean)
```

which denotes a type equivalent to

```
Function ( : Integer;  : Integer) Boolean end
```

In this way, type-operators provide a convenient short-hand notation for complex type expressions.

Acer does not support recursive type-operators since type-evaluation, i.e., the rewriting of every operator-call in terms of the definition of its type-operator, must be guaranteed to terminate. For instance, suppose we wanted a type-operator *BinaryTree* that we could call as *BinaryTree* (*Real*) or *BinaryTree* (vector.*Type*). Such a recursive type-operator might be incorrectly presented as

```
let BinaryTree be
  Operator (BaseType :: Any)
    Tuple
      x : BaseType
      leftChild : BinaryTree (BaseType)
      rightChild : BinaryTree (BaseType)
     end
   end
```

It should instead be defined in terms of a recursive type, e.g.,

```
let BinaryTree be
  Operator (BaseType :: Any)
    {let Type be
       Tuple
         x : BaseType
         leftChild : Type
         rightChild : Type
        end
      Type}
   end
```

The power of type-operators lies not in the ability to use them as a short-hand notation but in the ability to use them to define parametric abstract-types. For example, suppose we wanted to define an abstract-type for paired-objects of the same arbitrary base-type. Such an abstract-type is provided by the declaration

```
pair :
 Tuple
   Type :: Operator (BaseType :: Any) Any end
   construct :
     Function (BaseType :: Any; x : BaseType; y : BaseType)
       Type (BaseType)
      end
   x : Function (BaseType :: Any; p : Type (BaseType)) BaseType end
   y : Function (BaseType :: Any; p : Type (BaseType)) BaseType end
  end
```

which could be implemented by the tuple

```
tuple
  let Type be
    Operator (BaseType :: Any)
      Tuple x : BaseType; y : BaseType end
     end
  let construct be
    function (BaseType :: Any; x : BaseType; y : BaseType)
        : Type (BaseType)
      tuple x, y end
     end
  let x be function (BaseType :: Any; p : Type (BaseType)) p.x end
  let y be function (BaseType :: Any; p : Type (BaseType)) p.y end
 end
```

With the above declaration of *pair* visible, the function-call in

```
let pr be pair.construct (Integer, 10, 20)
```

yields a value of type

```
pair.Type (Integer)
```

The *x* component of the pair *pr* can then be selected using *pair.x* (*Integer*, *pr*). But this is very verbose, so method-calls are supported as a short-hand. Hence, the *x* component of the pair *pr* can also be selected using the value-selection *pr.x*, which is equivalent to the prefix-method-call *x.*(*pr*) since the type of *pr* is an abstract-type. When a prefix-method-call applies to an argument with an abstract-type given by an operator-call, as does *pr* with type *pair.Type* (*Integer*), the type arguments given in the operator-call are included as the first arguments in the function-call equivalent of the prefix-method-call. Therefore *x*.(*pr*) is equivalent to *pair.x* (*Integer*, *pr*). In this way, all forms of method-call, namely index, value-selection, prefix-method-call, dyadic-method-call, and unary-method-call, are supported for parametric abstract-types.

Type-operators enhance manipulation by supporting abstraction over type expressions, just as functions support abstraction over value expressions (see 4.14).

2.10 Pointer and reference

Acer provides two very similar types *Pointer* and *Reference* to support the modeling of updatable locations. These two parametric abstract-types are introduced respectively by the type-bindings

```
    let Pointer be pointers.Type
```

and

```
    let Reference be references.Type
```

Their modules are made available by the declarations

```
    pointers :
      Tuple
        Type :: Operator (BaseType :: Any) Any end
        new : Function (BaseType :: Any;  : BaseType)
                 Type (BaseType)
               end
       end
```

and

```
references :
  Tuple
    Type :: Operator (BaseType :: Any) Any end
    new : Function (BaseType :: Any;  : BaseType)
              Type (BaseType)
          end
    create :
      Function
          (BaseType :: Any
           fetch: Function () BaseType end
           store: Function ( : BaseType) Void end)
         Type (BaseType)
        end
   end
```

Pointer-values model updatable locations in terms of memory addressing whereas reference-values model updatable locations in terms of fetch/store functions.

A pointer-value p of type *Pointer* (*Integer*) can be created as

let p **be** *pointers.new* (*Integer*, 10)

We can then define q as

let q **be** p

so that the dereference p@ and the dereference q@ yield the value at the same storage location. Hence, the dereference p@ yields the value at the location pointed at by p. Such a dereference can be used anywhere that a variable of the base-type can be used. For instance,

{p@ **becomes** 100}

updates the value at the location pointed at by p.

Acer provides a pointer-call construct, which can be used to determine the address of an updatable location. Using this construct the pointer-value p could instead be created as

let p **be** {**let var** *location* **be** 10; **pointer** (*location*)}

Remember, Acer allocates the variable-value-bindings of a block on the heap rather than on the stack, so the above does not produce a dangling stack reference.

References are created in an analogous manner, e.g., a reference-value r of type *Reference* (*Integer*) can be created as

let r **be** *references.new* (*Integer*, 10)

The value at the location referenced by *r* is yielded by the dereference *r*@ and we can use

```
{r@ becomes 100}
```

to update the value at the location referenced by *r*. Acer provides a reference-literal[3] construct which could equivalently be used to create *r* as

```
let r be {let var location be 10; reference(location)}
```

From what has been described so far, pointers and references are equivalent. Their difference lies in the *references.create* function. The original definition of *r*:

```
let r be references.new(Integer, 10)
```

is equivalent to

```
let r be {let var location be 10
          references.create
            (Integer
             let fetch be function () location end
             let store be function (update: Integer)
                            {location becomes update}
                          end)}
```

Hence, a reference-value is implemented as a pair of functions, for fetching and storing the referenced value—the *fetch* function is called when the dereference *r*@ is used to yield a value and the *store* function is called when the dereference is used as the destination of an assignment. Therefore, references provide high-level control over dereference and assignment. For example, references could be used to record the number of assignments made to a particular reference, or to print out a message each time a reference is accessed.

To demonstrate a simple application of pointers and references we will show the implementation of an abstract-type for updatable paired-objects of the same arbitrary base-type. The binding

[3]Unlike a pointer-call, a reference-literal is a literal and so can be the target of a delayed reference, thereby supporting recursive dependencies, see A.7.2.

```
let pair be
 tuple
   let Type be
      Operator (BaseType :: Any)
         Tuple var x : BaseType; var y : BaseType end
       end
   let construct be
      function (BaseType :: Any; x : BaseType; y : BaseType)
          : Type (BaseType)
         tuple let var x be x'[1], let var y be y'[1] end
       end
   let x be function (BaseType :: Any; p: Type (BaseType))
                pointer (p.x)
               end
   let y be function (BaseType :: Any; p: Type (BaseType))
                pointer (p.y)
               end
 end
```

can be compiled to produce a declaration that provides such a type.

As a bit of an aside, notice in the definition of the *construct* function the use of the reused-value-identifier *x*'[1] to denote the outer definition of *x* in the signature, rather than the inner definition of *x* in the tuple. The general form of this notation is *x*'[*n*] where *n* is a non-negative integer-literal. This notation causes the search for the defining-occurrence of *x* to skip over the first *n* matching defining-occurrences. Hence, the reused-value-identifier *x*'[1] in the previous example refers to the outer definition of *x* rather than the inner one. Acer also provides reused-type-identifiers.

Reused-identifiers are provided to access identifiers that are defined in an outer scope and redefined in an inner scope, thereby ensuring that all identifiers remain visible. Reused-identifiers are not intended to encourage the use of nested renaming, which should only occur because the choice of identifier is forced. For example, nested reuse is forced if a tuple-literal with a component named *x* is to be defined in a scope in which *x* is already visible, and if the outer *x* must be used within that tuple.

To get back to our example of updatable pairs, recall that we earlier stated that if the prefix-method-call equivalent of a value-selection (or an index expression, as we shall see in the next section) has a *Pointer* or *Reference* type then the prefix-method-call equivalent is automatically dereferenced. Therefore, with *thePair* defined as

```
let thePair be pair.construct (Integer, 10, 20)
```

we could use *thePair.x* to yield 10 because the type of x.(*thePair*) is *Pointer* (*Integer*) and so *thePair.x* is equivalent to x.(*thePair*)@. Furthermore, we could also use

{*thePair.x* **becomes** 100}

to update the x component of *thePair*. In this way, pointers and references, in conjunction with value-selection (and index), provide support for modeling abstract-types with updatable components.

Pointers and references support manipulation by allowing functions to abstract over updatable locations (see 4.14).

2.11 Array

Arrays in Acer are defined as a parametric abstract-type introduced by the binding

let *Array* **be** *arrays.Type*

The implementation of *arrays* is made available by the declaration

```
arrays:
  Tuple
    Type :: Operator (BaseType :: Any) Any end
    error : Exception ( Void)
    length : Function (BaseType :: Any;  : Type (BaseType))
                Integer
              end
    index1 :
      Function (BaseType :: Any;  : Type (BaseType);  : Integer)
        Pointer (BaseType)
       end
    new : Function (BaseType :: Any;  : BaseType;  : Integer)
              Type (BaseType)
             end
   end
```

An array-value can be created by an array-literal, which is the only special syntactic support provided for arrays. For example,

array 12, 23, 34 **end**

creates an array-value of type *Array* (*Integer*). The base-type of an array-literal can be given explicitly as

array 12, 23, 34 **of** *Integer* **end**

An array-value can also be created as

arrays.new (*Real*, 0.0, 10)

which creates an array-value with 10 elements, each of which contains the value 0.0.

For an array-value a of type *Array* (*BaseType*), the length of a is yielded by *a.length* and the first element of a is yielded by a [0], which is equivalent to

index1.(*BaseType*, a, 0)@

Indexed arrays can be used in assignment, e.g.,

{a [*i*] **becomes** a [{*i* + 1}]}

The last element of a is yielded by

a [{*a.length* - 1}]

The exception *arrays.error* is raised when *arrays.index1* is called with an out-of-range subscript. We shall examine exceptions in the section that follows.

2.12 Exception and exception handling

In Acer, an exception is a special type of value that can be raised, along with a value of some particular base-type, to terminate normal evaluation. An exception-value e based on *Integer* can be defined as

let *e* **be exception** (*Integer*)

and is of type

Exception (*Integer*)

This exception could be raised with an associated base-value as

raise *e* **with** 0 **end**

When this raise expression is evaluated, normal evaluation is interrupted and the exception e, along with its associated value 0, is propagated back along the dynamic call chain until a handler for e is reached. The type of a raise expression is *Raise*, which is a special type that does not cause type-conflicts—an expression of type *Raise* does not yield a value but instead always raises an exception and therefore cannot cause a type-conflict. This does not mean that type-conflicts cannot occur within a raise expression, for the associated-value of a raise expression must certainly be of the exception's base-type, it simply means that the raise expression itself does not cause type-conflicts, regardless of the context in which it appears.

Acer's exceptions are different from those in other languages. In CLU [LAB+81] for instance, exceptions are static entities and functions must explicitly indicate which exceptions their evaluation may raise. This approach is not possible in Acer because exceptions are first-class values.

Acer provides two global exceptions

exit : *Exception* (*Void*)

and

fatal : *Exception* (*String*)

The exception *exit* is provided as a convenience, and may be raised using the short-hand

raise *exit* **end**

which is equivalent to

raise *exit* **with** {} **end**

because its base is *Void*. The exception *fatal* is special because if a program is terminated by raising this exception, the associated string is printed as an error message.

Acer provides several constructs for handling exceptions. The simplest such construct, a try-finally expression borrowed from Modula-3 [CDG+88], does not really handle exceptions but rather allows one to specify evaluations to be carried out regardless of whether an exception is raised. Consider trying to introduce two functions to be evaluated before and after each evaluation of some value *v*, regardless of whether *v* raises an exception. Simply using

begin *before* (); *v*; *after* () **end**

is not enough because if the evaluation of *v* raises an exception then *after* is not evaluated. Instead, a try-finally expression should be used as

```
try
  begin before(); v end
 finally
  after()
 end
```

so that *before* is evaluated before *v* and *after* is evaluated after *v*, even if the evaluation of *v* raises an exception. A try-finally yields the result of its body, and if both the body and the final-action raise exceptions, the exception raised by the final-action is raised; the value yielded by the final-action is ignored. Hence the above can be used to replace a value *v* with a traced equivalent that yields the same result as *v*. For instance, the function *before* could print the message, "About to evaluate *v*." and the function *after* could print the message, "Finished evaluating *v*." Such tracing could be used to implement a profiler [Ben87].

Acer provides the try expression for handling (trapping) exceptions. For example, the *integer* module supplies an exception *integer.error* of type *Exception*(*Void*) which is raised when a division by 0 is performed. Therefore,

```
try {x div y} then
 when integer.error then 0
end
```

can be used to intercept the exception raised when *y* is 0. A try expression yields the value of its body but if this evaluation raises an exception then the exception is compared against the exceptions appearing in the when-condition(s)—if a match is found, the try expression yields the value yielded by the evaluation of the consequent of the corresponding branch, otherwise propagation of the exception continues. Hence the above try expression substitutes the value 0 for {*x div y*} when *y* is 0.

In general, a try expression can have multiple when-branches, it can include an optional default-branch, each when-branch can have multiple exceptions, and each when-branch can include an optional defined-identifier, e.g.,

```
try body then
 when exception1, exception2 with definedIdentifier then branch1;
 when exception3, exception4 then branch2
 else defaultBranch
 end
```

If the body of a try raises an exception that is not matched by any branch and a default-branch is given then evaluation resumes with the default-branch thereby allowing unanticipated or unimportant exceptions to be handled. The defined-identifier of a when-branch is bound to the value associated with the exception handled by that branch to make that value available to the consequent. For example,

```
{let e be exception (String)
 try raise e with "Hello" end then
  when e with result then result
  end}
```

is a complicated way of yielding the value "Hello".

Acer's only other construct that handles exceptions is a keep-trying expression, which is also Acer's only looping construct other than high-level iteration (see 2.16). A keep-trying expression is very similar to a try expression , and has the general form

```
keep trying body then
 when exception1, exception2 with definedIdentifier then branch1;
 when exception3, exception4 then branch2
 else defaultBranch
 end
```

When a keep-trying expression is evaluated, the body is evaluated repeatedly until an exception is raised. This exception is then handled by the branches or default-branch of the keep-trying expression just as for a try expression. A keep-trying expression could be used to define the integer factorial function as

```
let ! be
 function (i : Integer)
   {let var i be i'[1]
    let var result be 1
    keep trying
      if {i < 1} then raise exit end
        else
         begin
           {result becomes {result * i}}
           {i becomes {i - 1}}
          end
        end
     then when exit then result
     end}
   end
```

which could be called as ! (10). Modula-3's loop construct [CDG+88] is similar to Acer's keep-trying construct but is less general in that it handles only one particular exception, the exit-exception, rather than all possible exceptions.

2.13 Variant

Acer provides discriminated unions in the form of the variant. A variant takes on different structures depending on the value of its discriminating tag, which must be an enumeration-value (or, as we shall see in section 2.14.3, an option-value). A particular variant *v*, based on an enumeration-type *E* with value *id*, is created by a variant-literal, e.g.,

```
let v be
  variant id of E with
    (let x be 10, let y be 10.0)
```

which has type

```
Variant E of
  when id then (x : Integer; y : Real)
 end
```

The defining-occurrence of the tag of a variant-literal or an identifier in the when-condition of a variant-type is determined by the literal-selection *E.id*.

The general form of a variant-type is

```
Variant EnumerationType of
  when id1, id2 then ();
  when id3, id4 then ()
 else ()
 end
```

Whereas the type of *v* is a variant-type with one alternative and an empty default-branch, in general, a variant-type may specify the structure of each possible alternative; a default-branch is used to specify the structure of any alternative not specified by the other branches. Such variant-types are related according to Acer's subtyping rules (see 2.14.5).

The tag of a variant cannot be changed and its components can be accessed only after its type is narrowed to a tuple-type using a variant-inspection. For example, the *x* component of *v* can be accessed using

```
inspect v then
 when id with t then t.x
 end
```

The general form of a variant-inspection is

```
inspect v then
 when id1, id2 with definedIdentifier then branch1;
 when id3, id4  then branch2
 else defaultBranch
 end
```

When a variant-inspection is evaluated, the selector v is evaluated first and its tag component is accessed to determine the matching branch. If the matching branch provides a defined-identifier then the narrowed variant-value is bound as a tuple-value to that identifier. Hence, in the preceding example, *t* has type

```
Tuple :E; x: Integer; y: Real end
```

because v has type

```
Variant E of
  when id then (x: Integer; y: Real)
 end
```

Finally, the matching branch is evaluated to yield the result of the variant-inspection. An empty default-branch in an inspection is equivalent to

```
raise fatal with "Inspection error." end
```

Variant-types could be used to define a lisp-like recursive type as

```
let Type be
  Variant Enumeration nil, cons end of
    when nil then ()
    when cons then (car: Type; cdr: Type)
   end
```

A variant-inspection can also be used as a Pascal-like case-statement if its selector is an enumeration rather than a variant. Such a variant-inspection is evaluated exactly as before except that the enumeration-value is examined directly rather than as the tag of a variant. Just as with a variant, the value of the selector is bound to the defined-identifier of the matching branch, although with an enumeration it has an enumeration-type rather than a tuple-type (see A.17.4).

2.14 Subtype

Up to this point issues involving subtyping have been carefully avoided, and yet subtyping affords much of Acer's flexibility. The question of subtyping is implicit in the question of whether the arguments of a function-call conform to the signature of the called function. Formally, one asks if an argument's type *T1* is a subtype of a parameter's type *T2*. Therefore, the subtyping rules of a language determine the flexibility of its parameter-passing mechanism.

In Acer, whether *T1* is a subtype of a type *T2* depends, of course, on the types that *T1* and *T2* denote. The subtyping rules for the various classes of types are described in the subsections that follow.

2.14.1 Tuple and single inheritance

The idea behind tuple subtyping is that when a tuple with certain components is expected, a tuple with additional or more specific components is also permitted. For example, suppose we have a function *f* which takes a tuple argument, e.g.,

```
let f be
  function (t : Tuple x : Integer end)
    t.x
   end
```

The function *f* may be called with an extended tuple:

```
f (tuple 10, 20, 30 end)
```

to yield the value 10. Notice that the above tuple-literal does not provide a name for its first component although in *f* this component is called *x*.

It is important to realize that every tuple-type is a subtype of **Tuple end**, which in turn is a subtype of **Any**, the root of Acer's subtype hierarchy. Therefore, the fewer components a tuple-type specifies, the closer it is to the root of Acer's subtype hierarchy and hence the more general it is considered to be.

Subtyping of tuple-types depends recursively on subtyping of components and so

```
Tuple : T1 end
```

is a subtype of

```
Tuple : T2 end
```

if *T1* is a subtype of *T2*. For example, the function

```
let g be
  function (t : Tuple x : Tuple y : Integer end end)
    t.x.y
   end
```

could be called as

```
g (tuple tuple 10, 20 end, 30, 40 end)
```

where the type of the first component of the tuple argument is a subtype of the type of the first component of the formal parameter *t* specified in *g*. Notice that we can say that the tuple

```
tuple tuple 10, 20 end, 30, 40 end
```

is more specific than is required by *g*.

The subtyping rule for fixed-value tuple components also applies for type components. For example, the tuple-type in

```
let T1 be Tuple Type :: Tuple end; value: Type end
```

is a subtype of the tuple-type in

```
let T2 be Tuple Type :: Any; value: Type end
```

Notice that an identifier introduced by a declaration of a tuple-type may also appear as an applied-occurrence, as does the identifier *Type* in each of the declarations of *value*. These identifiers are therefore involved in recursive subtyping, i.e., when determining whether *T1* is a subtype of *T2*. In general, identifiers introduced by distinct declarations are unrelated according to Acer's subtyping rules, but during recursive subtyping of two tuple-types *T1* and *T2*, each occurrence of an identifier introduced by a declaration in *T1* is *assumed* to be equivalent to the identifier of the corresponding declaration in *T2*. This is why, during recursive subtyping of *T1* and *T2*, the type of the declaration of *value* in *T1* is a subtype of the type of the declaration of *value* in *T2*, even though, generally, two such distinct declarations would be considered unrelated. (Of course if *T1* is not a subtype of *T2* for some other reason, i.e., if each had a third component but with different names, the assumption would be proven false, as in the general case.)

The subtyping rule for fixed-value and type components of tuples does not apply for variable-value components, which instead must have equivalent types. The necessity for this restriction can be seen in the following example. Suppose we have the tuple

```
let t be
  tuple
    let var x be tuple let y be 0 end
   end
```

from which we may extract the component *t.x.y*. Now suppose we pass *t* to the function *h*:

```
let h be
  function (t: Tuple var x: Tuple end end)
    {t.x becomes tuple end}
   end
```

which replaces the x component of t with a tuple that does not have a y component. After the call to h, the selection $t.x.y$ would no longer be valid since there is no longer a y component. But this fact is not reflected by the type of t. The call to h would be valid under the weaker subtyping rule for fixed-value components but the stronger subtyping rule for variable-value components prevents it.

Tuples in Acer are analogous to classes in object-oriented languages—both define the encapsulation of abstract-types, and both define a class hierarchy. However, in most object-oriented programming languages, such as C++ [Str86] and Eiffel [Mey88], a class is explicitly declared to be a subclass of some other class (or of other classes, if multiple inheritance is supported) and thereby inherits an interface and a default implementation; the interface can be extended and the implementation can be changed. In Acer, a tuple-type is implicitly a subtype of some other tuple-type by virtue of having a more specific declared structure; the sharing of implementations must be explicitly programmed.

Implementation sharing in Acer is a simple matter, however, because a tuple can be implemented as a copy of another. For example, suppose we have the global declaration

```
t: Tuple
    Type :: Any
    zero: Type
    succ: Function (x: Type) Type end
   end
```

We can then (automatically) produce the binding

```
let tt be tuple
          let Type :: Any be t.Type
          let zero: Type be t.zero
          let succ: Function (x: Type) Type end be t.succ
         end
```

which can be compiled to give a declaration with the same type as that of t. In effect, the implementation of t is "inherited" by tt. Also, the implementation of tt can be subsequently modified just as can default implementations in object-oriented languages. And, more importantly, even the type used by the implementation can be changed, e.g.,

```
let tt be tuple
          let Type :: Any be real.Type
          let zero : Type be 0.0
          let succ be function (x : Type) : Type {x + 1.0} end
         end
```

2.14.2 Record and multiple inheritance

Acer's tuples support a form of single "inheritance"—component order is significant so the subtype rules for tuples induce a lattice in the form of a tree, i.e., a given tuple-type has at most one direct parent in the lattice. With multiple inheritance, a type can have more than one direct parent, and the type lattice is a directed acyclic graph.

Acer provides records to support a form of multiple "inheritance." Records are exactly analogous to tuples except that component order is not significant and hence every record component must be named. Like tuples, records are created by record-literals, e.g.,

```
record let Type be Integer, let value: Type be 0 end
```

which have record-types, e.g.,

```
Record Type :: Any; value: Type end
```

Recall that during recursive subtyping of tuple-types we must consider the subtyping of corresponding components. For tuple-types these components correspond positionally, but for record-types corresponding components are instead determined by name. For example, consider the record-types

```
{let X be Record x : Integer end
 let XY be Record x : Integer; y : Integer end
 let XZ be Record x : Integer; z : Integer end
 let XYZ be Record x : Integer; y : Integer; z : Integer end
```

Acer's subtype rules specify that both *XY* and *XZ* are subtypes of *X*, that *XYZ* is a subtype of both *XY* and *XZ*, and that *XY* is unrelated to *XZ*. Other than the fact that tuple subtyping is positional and record subtyping is by name, tuple and record subtyping are completely analogous.

2.14.3 Enumeration and option

The idea behind enumeration subtyping is that when an enumeration-value of a given type is expected, an enumeration-value from an enumeration-type with fewer alternatives is also permitted. And because the values of an enumeration-type are ordered, an enumeration-type *E1* is a subtype of an enumeration-type *E2* if the identifiers in *E2* are a prefix of the identifiers in *E1*. Therefore, given the types

> {**let** *Day* **be Enumeration** *mon*, *tue*, *wed*, *thu*, *fri*, *sat*, *sun* **end**
> **let** *WeekDay* **be Enumeration** *mon*, *tue*, *wed*, *thu*, *fri* **end**
> **let** *WeekEnd* **be Enumeration** *sat*, *sun* **end**

we can say that *WeekDay* is a subtype of *Day* but *WeekEnd* is not related to either type. Notice that **Enumeration end** is a subtype of all enumeration-types and the more values an enumeration-type specifies, the closer that type is to **Any**, the root of the subtype hierarchy.

Acer's subtyping rule for enumeration-types specifies that order is significant so Acer also provides option-types, which are similar to enumeration-types except that order is not significant. Therefore, given the option-types

> {**let** *Day* **be Option** *mon*, *tue*, *wed*, *thu*, *fri*, *sat*, *sun* **end**
> **let** *WeekDay* **be Option** *mon*, *tue*, *wed*, *thu*, *fri* **end**
> **let** *WeekEnd* **be Option** *sat*, *sun* **end**

Acer specifies that *WeekDay* is a subtype of *Day* and also that *WeekEnd* is a subtype of *Day*. Just as for tuples and records, the subtype relation on enumeration-types induces a lattice in the form of a tree whereas the subtype relation on option-types induces a more general lattice in which a type can have more than one direct parent.

Option-types are completely analogous to enumeration-types except for the subtyping rule and the fact that the operations **ord** and **val** do not apply because of the lack of order. Options support literal-selection, can be used as the base-type of a variant, and can be used directly in a variant-inspection.

2.14.4 Function-type and type-operator

The idea behind function-subtyping is that when a function of a given type is expected a function with more general arguments and a more specific result is permitted. Therefore, the function-type

> **Function** (: *A1*) *R1* **end**

is a subtype of

> **Function** (: *A2*) *R2* **end**

if *R1* is a subtype of *R2* and *A2* is a subtype of *A1*.

Just as during recursive subtyping of tuples, during recursive subtyping of function-types *F1* and *F2*, an occurrence of an identifier introduced by a declaration in the signature of *F1* is considered equivalent to the identifier of the corresponding declaration in the signature of *F2*. For example,

```
Function (T :: Any;  : T) T end
```

is a subtype of

```
Function (T :: Tuple end;  : T) T end
```

precisely because of this equivalence; the result-types otherwise denote distinct types.

Since type-operators are functions from types to types, the subtype relation on type-operators is exactly analogous to that on functions.

2.14.5 Variant

Subtyping of variants makes use of the subtype relation defined for enumeration- and option-types. A variant-type *V1* is a subtype of a variant-type *V2* if first of all the base-type of *V1* is a subtype of the base-type of *V2*—intuitively, *V1* can only be a subtype of *V2* if there are the same or fewer alternatives that *V1* can specify. If the base-types are correct subtypes then *V1* is a subtype of *V2* if each signature given in *V1* is a subsignature of the corresponding signature in *V2*, where the subsignature relation is exactly analogous to the subtype relation on tuple-types. Therefore, *V1* can be a subtype of *V2* for two reasons, firstly *V1* can specify the structure of fewer alternatives and secondly *V1* can specify an extended or more specific structure for some alternatives. For example, the variant-type

```
Variant Enumeration nil end of
  when nil then (newStuff : T)
  end
```

is a subtype of

```
Variant Enumeration nil end of
  when nil then ()
 end
```

for the second reason, which in turn is a subtype of

```
Variant Enumeration nil, cons end of
  when nil then ()
  when cons then (car: AtomType; cdr: T)
 end
```

for the first reason.

2.14.6 Abstract-type

Acer has three forms of abstract-type. There are type-identifiers introduced by type-declarations, for example, *T* and *PT* are abstract-types when introduced by declarations in a signature as

```
(T :: Any; PT :: Operator (BaseType :: Any) Any end)
```

There are type-selections, for example, *v.Type* is an abstract-type when *v* is a tuple, record, or dynamic (see 2.15) with a component named *Type*. And there are operator-calls, for example, *PT* (*Integer*) is an abstract-type when *PT* is declared as above.

Every abstract-type is said to have a *quantifier* (see A.20.3), a concept closely related to the universal and existential quantifiers of logic. For instance, a function acts as a universal quantifier because it specifies a value for all possible parameter instantiations, whereas a tuple acts as a existential quantifier because it specifies the existence of one particular instantiation. For a type-identifier, the quantifier is the construct containing its declaration; this must be a signature (e.g., the signature containing *T* and *PT* above), a tuple-type, a record-type, or a dynamic-type (see 2.15). For a type-selection, the quantifier is given by the base (e.g., *v* above). And for an operator-call, the quantifier is determined recursively from the operator (e.g., from *PT* above). Intuitively, a quantifier encapsulates an abstract-type with its associated values and operations, that is, its methods. In fact, the defining-occurrence of the method-name of a prefix-method-call is determined by searching the quantifier of each argument whose type is an abstract-type.

An abstract-type is abstract because its definition is hidden and hence cannot be used to determine whether another type is a subtype of it. All that is known about an abstract-type is that it is a subtype of its declared type. Therefore, an abstract-type is considered to be a type unto itself—each abstract-type is distinct from all others and has no subtypes. Two abstract-types are equivalent only if they have the same quantifier, and for operator-calls, only if corresponding parameters are also equivalent.

Because a quantifier can be a value, as in the case of type-selections, statically determining if two quantifiers are the same requires a static equivalence relation for values. For example, *v1.Type* is a equivalent to *v2.Type* only if *v1* and *v2* are equivalent. A detailed definition of the static equivalence relation for values is left for the appendix (see A.20.3). Intuitively, two value expressions are equivalent only if the expressions are fixed and can be shown to always denote the same value.

To see why parameters must be equivalent when subtyping abstract-types, consider the types *PT* (*Integer*) and *PT* (**Any**) where *PT* is declared as before. If the hidden definition of *PT* is

> **Operator** (*BaseType* :: **Any**) **Function** () *BaseType* **end end**

it would be valid to consider *PT* (*Integer*) a subtype of *PT* (**Any**) since

> **Function** () *Integer* **end**

is a subtype of

> **Function** () **Any end**

However, if the hidden definition of *PT* is

> **Operator** (*BaseType* :: **Any**) **Function** (: *BaseType*) **Any end end**

it would not be valid to consider *PT* (*Integer*) a subtype of *PT* (**Any**) since

> **Function** (: *Integer*) **Any end**

is not a subtype of

> **Function** (: **Any**) **Any end**

because **Any** is not a subtype of *Integer*. Therefore, in general, there is no fixed subtype relationship between *PT* (*Integer*) and *PT* (**Any**). Only when corresponding parameters are equivalent can a relationship exist.

Because an abstract-type is given a type by its declaration and because we can determine whether this type is a subtype of some other type, we can, by transitivity, determine whether an abstract-type is a subtype of a *concrete-type*, that is, a type that is not an abstract-type. In general, if the declared type of an abstract-type, which must be a concrete-type, is a subtype of some other concrete-type then the type implementing the abstract-type is also a subtype of that concrete-type. Therefore we can say that an abstract-type is a subtype of a concrete-type if its declared type is a subtype of that concrete-type.

This mechanism allows an abstract-type to partially reveal its definition by specifying that its definition is a subtype of a given concrete-type. For example, given an abstract-type declared as

T :: **Tuple** x : *Integer* **end**

and a value declared as

$v : T$

it is valid to select the x component of v using $v.x$ because although the type of v is T, which is an abstract-type with no method named x, the type of the type of v (i.e., the kind of v) is a tuple-type, which specifies that v is a tuple with an x component. Also, v can be a passed to a function f declared as

f : **Function** (: **Tuple** x : **Any end**) *Void* **end**

using the function-call $f(v)$ because v is known to be a tuple with an x component of type *Integer*.

To summarize the subtyping rules for abstract-types:

- An abstract-type *T1* is equivalent to an abstract-type *T2* if the quantifier of *T1* is equivalent to the quantifier of *T2* and corresponding parameters are equivalent.
- An abstract-type *T1* is a subtype of a concrete-type *T2* if the (declared) type of *T1* is a subtype of *T2*.

The first part reflects the fact that each declared abstract-type is distinct and the second part reflects the fact that every abstract-type is known to be implemented by a type that is a subtype of its declared type.

This completes the overview of Acer's subtyping rules. A number of subtle points that are fully examined in Appendix A have been overlooked. However, these points are not germane to the discussions that follow.

2.15 Dynamic

All the features of Acer described so far support static type-checking. To also provide support for dynamic type-checking, Acer provides a data structure similar to a tuple called a *dynamic*. Like tuples, dynamics are created by dynamic-literals, e.g.,

dynamic let *Type* **be** *Integer*, **let** *value* : *Type* **be** 0 **end**

which have dynamic-types, e.g.,

```
Dynamic Type :: Any; value : Type end
```

Also, the subtyping rules for dynamic-types are the same those as for tuple-types. Type- and value-selection also apply for dynamics. Dynamics differ from tuples, however, because the first component of a dynamic must be a type and because a representation of this type is actually stored as a run-time component of the dynamic-value. This type-tag component can be examined at run-time using dynamic-inspection.

For example, the dynamic-value *d* in

```
let d be
  dynamic
    let Type be Integer
    let value : Type be 0
   end
```

could be inspected as

```
inspect d Then
 when Integer with t then t.value
 end
```

to yield the value 0. A dynamic-inspection is similar in appearance to a variant-inspection except **Then** (instead of **then**) is used to introduce the branch-list and the condition of each when-branch specifies a type (instead of values).

When a dynamic-inspection is evaluated, the selector is evaluated first to determine the type-tag of the resulting dynamic-value. This type is then compared to the types appearing as the conditions of successive branches until a condition with a supertype is found. Note that the subtyping rules used for dynamic-inspection are the same as the static subtyping rules. In addition, a dynamic-inspection may provide an **else** part to specify a default-branch. If the matching branch of a dynamic-inspection provides a defined-identifier then the narrowed dynamic-value, which then takes on the form of a tuple-value, is bound to that identifier. Finally, the consequent of the matching branch is evaluated to yield the result of the inspection.

The defined-identifier of a type-when-branch of a dynamic-inspection is bound to a dynamic-value whose type is narrowed to be tuple-type. This tuple-type has the same declarations as the dynamic-type with the following exceptions: the initial type-declaration is replaced by an anonymous fixed-value-declaration of type **Any**, and each applied-occurrence of the defined-identifier of the initial type-declaration is replaced by a copy of the type-when-branch's condition. For example, in the dynamic inspection above, the selector *d* has type

Dynamic *Type* :: **Any**; *value*: *Type* **end**

and since the condition of the matching branch is *Integer*, the type of *t* is

Tuple : **Any**; *value*: *Integer* **end**

There are restrictions on which types can be used as the tag of a dynamic. In particular, the type-tag of a dynamic-literal may not depend on abstract-types that can be bound to different definitions during program execution. For example, if this were allowed, consider evaluating the block

```
{let var d be
   dynamic
     let Type :: Any be Integer
     let value: Type be 1
    end
 let f be
   function (T :: Any; x : T )
     {let result be
        inspect d Then
          when T with dt then dt.value
         else x
         end
      begin
        {d becomes
           dynamic let Type be T; let value: Type be x end}
        result
       end}
     end
   tuple f (Real, 1.0), f (Character, '1') end}
```

The updatable variable *d* is used by the function *f* for two purposes. First, *f* inspects *d* and if *d* has *T* as its type-tag then the *value* component of *d*, which then has type *T*, is stored in *result*. Second, before yielding the value stored in *result*, *f* assigns to *d* a dynamic-value encapsulating *T* and its value *x*. During the first call to *f* there is no problem (*f* simply yields *x*) but during the second call, the dynamic-inspection succeeds because *d* then contains a dynamic-value with *T* as its type-tag.[4] Therefore, during the second call, the *value* component of the dynamic is stored in *result*, which is subsequently yielded by *f*. But this value is yielded as a *Character* value even though it is actually the *Real* value 1.0 supplied to *f* during the first call.

[4]Remember, there is no way for the function to ascertain the type bound to *T*.

The problem with abstract-types goes even beyond this because, as we will see shortly, a dynamic-value can be stored in a file that is loaded by another program, or by a later run of the same program. Therefore, even if an abstract-type is fixed during program execution it can nevertheless be recompiled, thereby invalidating any stored dynamic-values that depend on it. As a consequence, it may seem prudent to disallow completely abstract-types as type-tags of dynamics. However, many basic types such as *Integer*, *Real*, and *Character* are defined as abstract-types and it will most certainly be necessary to support dynamic-values based on these. Therefore, an abstract-type is permitted as part of the type-tag of a dynamic only when it can be statically shown to be bound to the same definition throughout a single program execution, i.e., if it is a *closed type* (see A.14). The problem of an abstract-type's implementation changing, thereby invalidating stored dynamic-values based on it, is left as an open problem.

Operations on dynamic-values are provided by the module *dynamics* which supports copying, reading, and writing of dynamic-values:

```
dynamics :
  Tuple
    error : Exception ( Void)
    copy : Function ( : Dynamic Type :: Any; : Type end)
               Dynamic Type :: Any; value: Type end
             end
    input : Function (filePath: String)
               Dynamic Type :: Any; value: Type end
             end
    output : Function (filePath: String
                        : Dynamic Type :: Any; : Type end)
                Void
              end
   end
```

For example, given the function *g* defined as

```
let g be
  {let var x be 0
   function ()
     begin {x becomes {x + 1}}; x end
    end}
```

we can create a dynamic copy *h* of *g* as follows:

```
let h be
  inspect
    dynamics.copy
      (dynamic
         let Type be Function () Integer end
         let value: Type be g
        end)
   Then when Function () Integer end with t then t.value
   end
```

Given g and h as defined above, we could create the tuple

```
tuple {g () + g ()}, {h () + h ()} end
```

which creates a tuple equivalent to

```
tuple 3, 3 end
```

Dynamic copying preserves the sharing and circularities within the object being copied. This can be seen in the above example from the fact that even the closure of g is copied, not shared between g and h.

The function g could also be copied using *input* and *output* as follows

```
let h be
  inspect
    begin
      dynamics.output
        ("filename"
         dynamic
           let Type be Function () Integer end
           let value: Type be g
          end)
      dynamics.input ("filename")
     end
   Then when Function () Integer end with t then t.value
   end
```

Evaluating the above inspection involves storing a representation of g in the file `"filename"` and then retrieving this representation, thereby producing a copy of g. The file `"filename"` can even be retrieved by other programs or by subsequent executions of the same program, so *dynamics* provides support for persistent storage.

2.16 Iterator and accumulator

Acer provides constructs for supporting high-level iteration and accumulation [Cam89]. Iterators (i.e., sequence producers) and accumulators (i.e., sequence consumers) are tuples with appropriate exception and function components.

An iterator-type is given in terms of a base-type as *Iterator*(*BaseType*), where *Iterator* is introduced by the global type-binding

```
let Iterator be
  Operator (BaseType :: Any)
    Tuple
      done : Exception (Void)
      produce : Function () BaseType end
      terminate : Function () Void end
    end
  end
```

Since *Iterator* is not an abstract-type, we can create an iterator-value with a tuple-literal that has the required exception and function components. For example, the following function creates an iterator that produces a sequence of integers in the specified range:

```
let range be
  function (first : Integer; last : Integer) : Iterator (Integer)
    tuple
      let done be exception (Void)
      let produce be
        function ()
          if {next > last} then raise done end
           else
            {let result be next
             begin
               {next becomes {next + 1}}
               result
              end}
           end
         end
      let terminate be function () {} end
      let var next be first
     end
   end
```

An iterator generates a sequence of values through repeated calls to its *produce* function, which raises the *done* exception when the sequence is exhausted. An iterator is terminated by a call to its *terminate* function, which is called either because the *done* exception is raised or because no further values are required. Iterators are typically used in conjunction with Acer's version of the for-loop called an *iteration*, e.g.,

```
for i in range(0, {a.length - 1}) do
  {a[i] becomes 0.0}
 end
```

As we shall see below, an iteration expression can also include an accumulator that consumes the sequence of values to produce a single value.

An accumulator-type is given in terms of a base-type and a result-type as

```
Accumulator(BaseType, ResultType)
```

where *Accumulator* is introduced by the global type-binding

```
let Accumulator be
  Operator (BaseType :: Any;  ResultType :: Any)
    Tuple
      done : Exception (Void)
      consume : Function (: BaseType)  Void end
      terminate : Function () ResultType end
     end
   end
```

Just like an iterator, an accumulator can be created by an appropriate tuple-literal. For example, the following function creates an accumulator that adds up a sequence of values

```
let sum be
  function (initial: Integer) : Accumulator(Integer, Integer)
    tuple
      let done be exception (Void)
      let consume be
        function (i : Integer)
          {result becomes {result + i}}
         end
      let terminate be function () result end
      let var result be initial
     end
   end
```

An accumulator uses up a sequence of values through repeated calls to its *consume* function, which raises the *done* exception when the accumulation is complete. An accumulator is terminated by a call to its *terminate* function, which is called either because the *done* exception is raised or because there are no more sequence values. An accumulator's *terminate* function yields the result of the accumulation.

An accumulator can be used in an iteration expression as follows:

```
for i in range(1, 5) do
  sum(0) i
 end
```

which for the above definitions yields 15. In general, an iteration expression may introduce several defined-identifiers and may contain a filter expression, e.g.,

```
for i in x;  j in y andif filter(i, j) do accum body(i, j) end
```

which is defined to be equivalent to

```
{let iter1 be x;  let iter2 be y;  let a be accum
 keep trying
   {let i be iter1.produce();  let j be iter2.produce()
    if filter(i, j) then a.consume(body(i, j)) end}
  then
   when iter1.done, iter2.done, a.done then
    begin
      iter1.terminate();  iter2.terminate();  a.terminate()
     end
  end}
```

A missing accumulator is equivalent to *discard* which is globally declared as

```
discard:
  Tuple
    done: Exception(Void)
    consume: Function (: Any) Void end
    terminate: Function () Void end
   end
```

and is implemented as

```
let discard be
  tuple
    let done be exception(Void)
    let consume be function (: Any) {} end
    let terminate be function () {} end
   end
```

A missing filter is equivalent to *true*.

Accumulators can also be used in Acer's accumulation expression, e.g.,

```
sum(0)([1, 2, 3, 4, 5])
```

which is defined, in general, to be equivalent to

```
{let a be sum(0)
 try begin
       a.consume(1)
       a.consume(2)
       a.consume(3)
       a.consume(4)
       a.consume(5)
       a.terminate()
      end
   then when a.done then a.terminate()
  end}
```

Accumulation expressions are particularly useful as a notation for expressing "literals" of abstract-types. For example,

```
list.construct(Integer)([2, 3, 5, 7])
```

might be used to create an list of integers, i.e., a value of type

```
list.Type(Integer)
```

The definitions of Acer's high-level iteration and accumulation constructs demonstrate just how easily Acer's syntax can be extended to provide convenient notations when such notations are deemed necessary or desirable.

2.17 Code-patch

The final feature we shall examine is the code-patch. A code-patch is similar in appearance to a compound-value but is used to specify machine-dependent evaluation. For example, the code-patch

```
code Integer; move(d1, d0); d0 end
```

might be used to move the contents of register *d1* to register *d0* and to yield the final value of *d0* as an *Integer* result. The code-patch

code *Pointer* (**Any**); *A7* **end**

might be used to yield the value of the stack-register *A7* as a pointer.

A code-patch is a list of expressions. The first expression must be a type, which specifies the type of value yielded by the code-patch. The remaining expressions are interpreted as the data and instructions of some particular machine—the interpretation varies from implementation to implementation. Typically, the final expression is used to specify the effective-address of the value to be yielded, and the expressions between the first and the last are used to specify the machine instructions to be executed.

Code-patches support manipulation by providing low-level access to features that would otherwise be outside the domain of Acer, see 4.15.

2.18 Summary

This completes our description of the typeful programming language Acer. Acer has been carefully designed not only as a flexible high-level language but also as a language that directly supports program manipulation. In terms of syntax, Acer supports manipulation by providing a concrete syntax that avoids syntactic ambiguity, including pairwise ambiguity between grammar productions. In terms of semantics, Acer supports manipulation by providing a flexible abstract syntax that corresponds directly to the concrete syntax. These topics will be addressed in detail in the chapters that follow.

Chapter 3

The representation of syntax

3.1 The nature of syntax

Language design is primarily concerned with semantics, which is described in terms of abstract syntax; concrete syntax is largely irrelevant as it merely provides a view of abstract syntax. The goal of language design is to represent meaning in the simplest and most flexible way. Since programming languages represent algorithms, which must be translated by computers and yet be readable to humans, a representation of programs is flexible only to the extent to which it supports both machine and human manipulation. Also, programs are not simply written and then read, as with natural language, programs are created and then modified by maintenance activities. These activities are best carried out by manipulating an abstract representation.

This chapter describes a GRAMPS-style approach [CI84] for representing context-free syntax and a relational approach, inspired by NURN [Dyc90], for representing context-dependent syntax. According to these approaches, syntactic objects are represented as the interrelated nodes of a syntax tree. A GRAMPS-style grammar, as we will see, can be used either to specify just abstract syntax or to specify both abstract syntax and concrete syntax. A language can therefore be designed without regard to appearance. To further this goal, a graphical view of nodes is presented as the basis for the design of pure abstract syntax.

During preliminary design, a language designer should ignore lexical concerns, such as choice of symbols, and parsing concerns, such as choice of keywords and punctuation, and should instead focus on the representation of meaning. Only when a suitable abstract syntax has been designed should the design of concrete syntax begin. An ASCII view, being necessary for portability, should be considered during the design of concrete syntax but the limitations of ASCII must not restrict attempts to provide more readable views.

To demonstrate the representation techniques of this chapter in action, examples are given in terms of Acer.

3.2 Context-free syntax

The fundamental semantic objects of Acer are classified into the categories type, value, declaration, and binding (see A.2). In the abstract syntax such categories are specified using a GRAMPS-style alternation rule as follows, in which || is used to separate alternatives:

$$\langle \mathit{Declaration} \rangle ::= \langle \mathit{TypeDeclaration} \rangle \parallel \langle \mathit{ValueDeclaration} \rangle$$

This rule is interpreted to mean that a declaration is either a type-declaration or a value-declaration.

An alternation rule does not specify node structure, it specifies choice of structure; node structure is specified by GRAMPS-style lexical, construction, or list rules. To be more exact then, each lexical, construction, or list rule defines a *node class* whereas each alternation rule defines a set of node classes, or *node category*. Every node belongs to a particular node class and, depending on the kind rule that specifies its class, is either a lexeme, a construction, or a list; each will be considered in turn.

3.2.1 Lexeme

Lexemes, along with childless lists and childless constructions, form the leaves of a syntax tree. Typical lexemes in programming languages are identifiers, numbers, characters, and strings. Lexemes are encoded as sequences of symbols but the details of this encoding are of no concern during the design of abstract syntax.

Abstractly, a lexeme has a *spelling* that must conform to a regular expression written in terms of a particular set of symbols. Although lexical rules are clearly in the domain of concrete syntax, it is safe to assume that an ASCII-based concrete syntax will be provided. Further discussion of lexical rules is deferred until section 3.4.1 when we deal with concrete syntax.

For now, assume that a lexeme's spelling can be illustrated as a sequence of symbols enclosed by a rectangular box. Examples of each of Acer's six lexeme classes are

$$\boxed{\quad \mathit{two} \quad} \quad \boxed{\quad \mathit{Two} \quad} \quad \boxed{\quad 2 \quad} \quad \boxed{\quad 2.0 \quad} \quad \boxed{\quad \text{`2'} \quad} \quad \boxed{\quad \text{"II"} \quad}$$

In Acer, the node class of each lexeme, that is, whether it is a value-identifier, type-identifier, integer-literal, real-literal, character-literal, or string-literal, is obvious from its spelling.

3.2.2 Construction

Constructions are used to model semantic objects consisting of a fixed set of components. For example, an Acer type-declaration consists of an optional type-identifier and a type. This is specified by the following GRAMPS-style construction rule, which uses ⟦ ⟧ to indicate optionality:

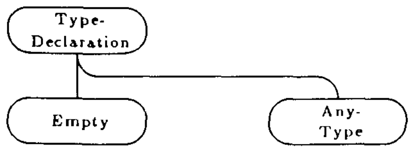
$$\langle \mathit{TypeDeclaration} \rangle ::= [\![\langle \mathit{DefinedIdentifier}{:}\mathit{TypeIdentifier} \rangle]\!] \langle {:}\mathit{Type} \rangle$$

A construction rule specifies a name for each component—if an explicit component name is omitted, the component name is implicitly taken to be the same as the node class name. Accordingly, an Acer type-declaration consists of an optional defined-identifier, which must be a type-identifier, and a type, which must be a type expression. Perhaps it is confusing to use type as both a node class name and a component name but it seems natural to do so when nodes represent types and when constructions have types as components.

Abstractly, a construction is the parent of a fixed number of named children nodes. These children are ordered and so can be referenced either by name or by position, that is, as the nth child.

Optional construction components are handled by the provision of empty nodes. Every GRAMPS-style grammar must provide a construction rule that defines empty to be a childless construction.[1] When a component is specified optional, a node of class empty is permitted to appear in place of that component. Therefore, a construction component is never missing, although it may be empty.

A construction node is illustrated as a labeled box connected to each of its children. An example of an Acer type-declaration is



Notice the empty node as the defined-identifier and the childless construction node of class any-type as the type.

[1]In section 3.4.2 we shall see how the construction rule that defines empty can be used to specify a visible concrete representation for empty nodes.

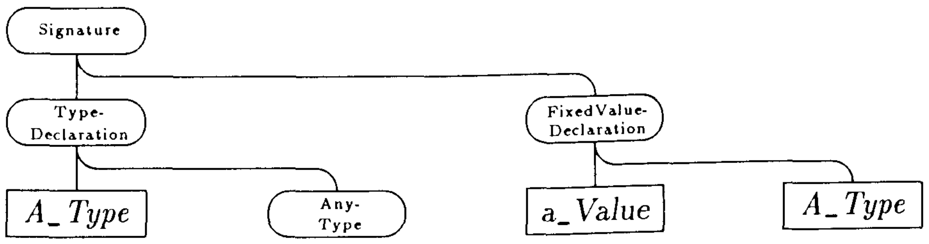
3.2.3 List

Lists are used to model semantic objects consisting of an arbitrary number of elements. For example, an Acer signature consists of a series of declarations. This is specified by the following GRAMPS-style list rule, which uses {[ ]} to indicate zero or more repetitions:

$$\langle \mathit{Signature} \rangle ::= \{\!\![\ \langle \mathit{Declaration} \rangle\ ]\!\!\}$$

Abstractly, a list is the parent of zero or more children nodes. The children of a list are ordered and can be referenced only by position—list children are not named as are construction children.

A list node is illustrated as is a construction node, for example, an Acer signature is illustrated as



3.3 Context-free relations

The membership of nodes in classes and categories determines the unary context-free relations of a language. The tree structure of nodes determines the ternary parent-position-child relation, which can be described in terms of two binary relations as

- the parent relation, which relates a child node to its parent node;
- and the position relation, which relates a child node to its position with respect to its parent.

The binary child-name relations, which relate a construction to its named child node, provide an alternative description of a construction's tree structure. Together, the membership relations and the tree relations completely describe context-free syntax.

It is reassuring to realize that nodes are sufficient for representing the structure of syntactic objects—complexity at this stage will only be compounded by the complexity of the language we are to design. The importance of using a simple representation of abstract syntax cannot be over emphasized.

3.3.1 Context-free manipulation

Context-free syntax is manipulated by creating nodes and by attaching nodes to form syntax trees. Creating new unattached nodes is a simple matter: an unattached lexeme is created given an appropriate spelling; an unattached construction is created given the correct number of unattached children, each of the appropriate class; and an unattached list is created given any number of unattached children of the appropriate class. When a construction or list is created, its children become attached. (As a convenience, if an attached child is supplied, its structure can be copied to yield an unattached child.)

Existing syntax trees are modified by altering the attachment of nodes. This can be done in one of three ways: a child of a construction or list can be replaced by an unattached node of the appropriate class, leaving the replacement node attached and the replaced node unattached; a list child can be deleted, leaving the child unattached and the list with one less child; or an unattached node of the appropriate class can be inserted at any position of a list, leaving the node attached and the list with one more child. Manipulation of nodes is very straightforward indeed.

Manipulation is made easier by consistency. For example, every node in a tree has a parent, except for the root, which is unattached. For consistency then, the parent of an unattached node will be given by a special unattached node called *the unattached empty node*. Therefore, every node has a parent and the unattached empty node is its own parent; the unattached empty node is special because it is a parent and yet has no children.

Similarly, every node in a tree has a position with respect to its parent, except for the root, so for consistency the position of an unattached node will be given by zero.

To support automated manipulation, a GRAMPS-style grammar can be used to generate a *metaprogramming system*, an implementation of nodes as an abstract data-type in a host programming language. The manipulation of nodes by programs is then done according to the notions that have just been described. In particular, a metaprogramming system provides facilities for creating nodes, for traversing nodes, and for editing nodes—a lexeme is created given its spelling and a construction or list is created given its children; nodes are traversed via the parent and child relations that connect them; and nodes are modified by replacing, deleting, or inserting children. More details of Acer's metaprogramming system are given in Chapter 5 and Appendix C.

Before moving on to examine how programming language concepts such as scoping and typing are represented as relations on nodes, let us examine how a GRAMPS-style grammar specifies concrete syntax.

3.4 Concrete syntax

Concrete syntax provides a way of viewing nodes in terms of a particular set of symbols; presumably ASCII will be used, if only to provide a portable view. Concrete syntax can be specified by the same GRAMPS-style grammar used to specify abstract syntax.

3.4.1 Lexeme

The concrete syntax of a lexeme is specified by a regular expression that uses $\|$, $[\![\ ]\!]$, $\{\!\{\ \}\!\}$, and $(\!(\ )\!)$ to indicate choice, optionality, zero or more repetitions, and grouping, respectively. For example, the concrete syntax of an Acer real-literal is specified as

$$
\begin{array}{l}
\langle \mathit{RealLiteral} \rangle ::= \\
\quad [\![\texttt{-}]\!]\ \langle \#\mathit{Digit} \rangle\ \{\!\{ \langle \#\mathit{Digit} \rangle \}\!\}\ .\ \langle \#\mathit{Digit} \rangle\ \{\!\{ \langle \#\mathit{Digit} \rangle \}\!\} \\
\qquad [\![(\!(\mathbf{e} \,\|\, \mathbf{E})\!)\ [\![\texttt{-}]\!]\ \langle \#\mathit{Digit} \rangle\ \{\!\{ \langle \#\mathit{Digit} \rangle \}\!\}]\!]
\end{array}
$$

A GRAMPS-style grammar provides character description rules, as is indicated by a '#' in the rule name, for naming regular expressions. For example, digit is defined as

$$\langle \#\mathit{Digit} \rangle ::= \mathbf{0} \,\|\, \mathbf{1} \,\|\, \mathbf{2} \,\|\, \mathbf{3} \,\|\, \mathbf{4} \,\|\, \mathbf{5} \,\|\, \mathbf{6} \,\|\, \mathbf{7} \,\|\, \mathbf{8} \,\|\, \mathbf{9}$$

3.4.2 Construction

The concrete syntax of a construction is specified by the placement of tokens, that is, keywords and punctuation. For example, the concrete syntax of an Acer type-declaration is specified as

$$\langle \mathit{TypeDeclaration} \rangle ::= [\![\langle \mathit{DefinedIdentifier{:}TypeIdentifier} \rangle]\!]\ ::\ \langle {:}\mathit{Type} \rangle$$

Examples of Acer type-declarations are

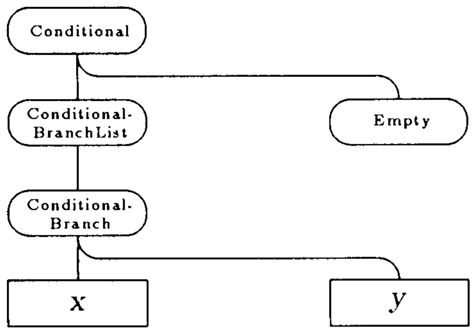
$$:: \mathbf{Any} \qquad T :: \mathbf{Any} \qquad \mathit{Type1} :: \mathit{Type2}$$

Notice how an empty component disappears in the concrete view.

Tokens can be placed within the optional brackets of a construction-rule component and are then printed only if the associated component is not empty. For example, Acer's conditional expression is defined as

$$
\begin{array}{l}
\langle \mathit{Conditional} \rangle ::= \\
\quad \langle \mathit{Branches{:}ConditionalBranchList} \rangle \\
\qquad [\![\mathbf{else}\ \langle \mathit{DefaultBranch{:}Value} \rangle]\!]\ \mathbf{end}
\end{array}
$$

so a conditional viewed graphically as



can be viewed in ASCII as

if *x* **then** *y* **end**

As an alternative treatment of empty nodes, a GRAMPS-style grammar can specify a visible concrete syntax for empty nodes. An empty optional component is then made visible by printing it as a keyword—optional tokens associated with that component are then also printed. For example, Acer defines the concrete syntax of an empty node as

$\langle \mathit{Empty} \rangle ::= \textbf{nothing}$

so the conditional in the previous example could be equivalently viewed as

if *x* **then** *y* **else nothing end**

3.4.3 List

The concrete syntax of a list, as that of a construction, is specified by the placement of tokens. For example, the concrete syntax of an Acer signature is specified as

$\langle \mathit{Signature} \rangle ::= (\ \{\!\![\ \langle \mathit{Declaration} \rangle\ ]\!\!\}^{[\![\,;\,]\!]}\ )$

A superscripted token following $\{\!\![\ ]\!\!\}$ indicates that the token separates adjacent list elements; a token enclosed by $[\![\ ]\!]$ indicates that the token itself is optional. Accordingly, the declarations of an Acer signature are separated by optional semicolons, for example, a signature viewed as

(*A_Type* :: **Any**; a_*Value*: *A_Type*)

could be equivalently viewed as

(*A_Type* :: **Any**   a_*Value*: *A_Type*)

All commas and semicolons are optional in Acer.

Depending on the purpose of a concrete syntax, there are many considerations to take into account. For example, a concrete syntax must be readable, both to machines and humans. These issues are discussed in Chapter 4. Let us move on now to examine context-dependent syntax.

3.5 Context-dependent relations

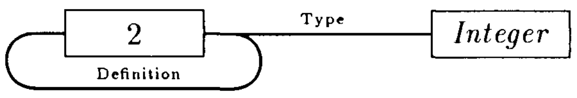
The usual approach for defining context-dependent syntax involves using either an attribute grammar, which maps program semantics to arbitrary data structures, or denotational semantics, which maps program semantics to mathematical values and functions. In constrast, the relational approach described here maps program semantics to nodes and node relations. This is the essence of the double-duty strategy, i.e., nodes represent both context-free syntax and context-dependent syntax.

Representing static semantic entities as nodes and *only* as nodes enhances support for program manipulation. For instance, if the type of an expression is represented as a node, this node can be used to introduce a new object of that type. On the other hand, if the type were not represented as a node, how would one declare a variable to hold an expression of that type? The double-duty strategy eliminates such problems.

Using the double-duty strategy, context-dependent syntax can be completely specified as relations on nodes. The nature of these relations varies from language to language but three relations dominate the definitions of most programming languages:

- *defining-occurrence* relates an identifier node to a corresponding identifier node used to define the identifier for the current scope,
- *type* relates an expression node to the node that represents its type,
- and *definition* relates a node to the node that represents its meaning according to rewrite rules.

Each of these three relations is binary and can therefore be graphically represented much like the parent relation. A context-dependent relation is graphically illustrated by connecting the right side of the first node to the left side of the second node. If more than one sort of context-dependent relation is to be illustrated in a single diagram then the relations are labeled to distinguish them. For instance, the following illustrates that the integer-literal 2 has the type-identifier *Integer* as its type and that it has itself as its definition:



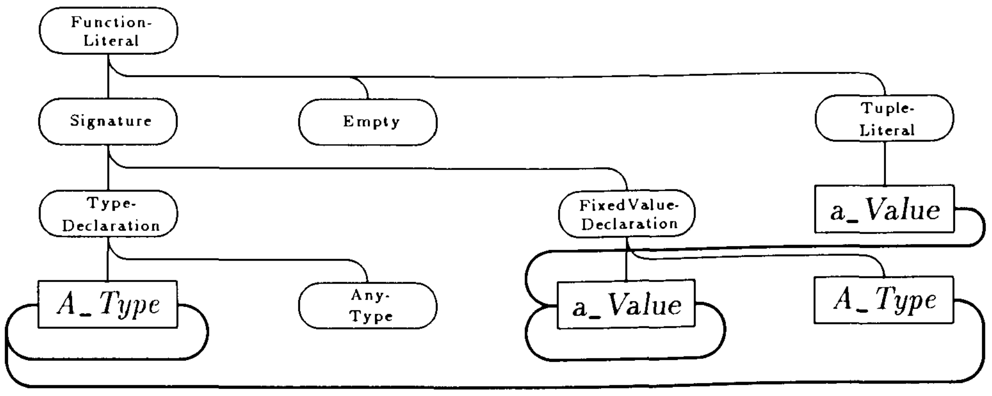
Each relation will be briefly described in the sections that follow. A more detailed treatment of the relations is presented in section A.3.

3.5.1 Defining-occurrence

The defining-occurrence relation as it applies to

function (A_Type :: **Any**; a_*Value*: A_Type) **tuple** a_*Value* **end end**

is illustrated as



An identifier that is its own defining-occurrence is called a defining-occurrence, every other identifier is called an *applied-occurrence.* Notice that the left side of each defining-occurrence is connected to the right side of each of its applied-occurrences—binary relations can be interpreted in two directions.

The specification techniques of NURN could be used to define formally the rules for how each identifier comes to be related to its defining-occurrence. However, I am not so much concerned with how relations are specified as I am with the nature of the relations themselves. Acer is defined in terms of relations specified in English; NURN could be used to formalize this definition but I have not done so. To reiterate, I am more concerned with what is specified than with how it is specified.

To specify the defining-occurrence relation, a language designer describes how a syntax tree is searched during identifier-lookup (see A.3.1). Defining scope in this way closely mirrors the natural process of incremental identifier-lookup.

3.5.2 Type

For some languages, particularly languages with name type-equivalence, the type relation is simple because the syntax tree contains the necessary nodes for representing the type of each expression. In general, however, a type can be derived from the syntax of an expression and there may be no node representing it available in the syntax tree. In such languages, of which Acer is one, derived nodes must be created to represent types.

To see how the need for derived nodes arises, consider the function-literal we saw previously

function (*A_Type* :: **Any**; a_*Value*: *A_Type*) **tuple** a_*Value* **end end**

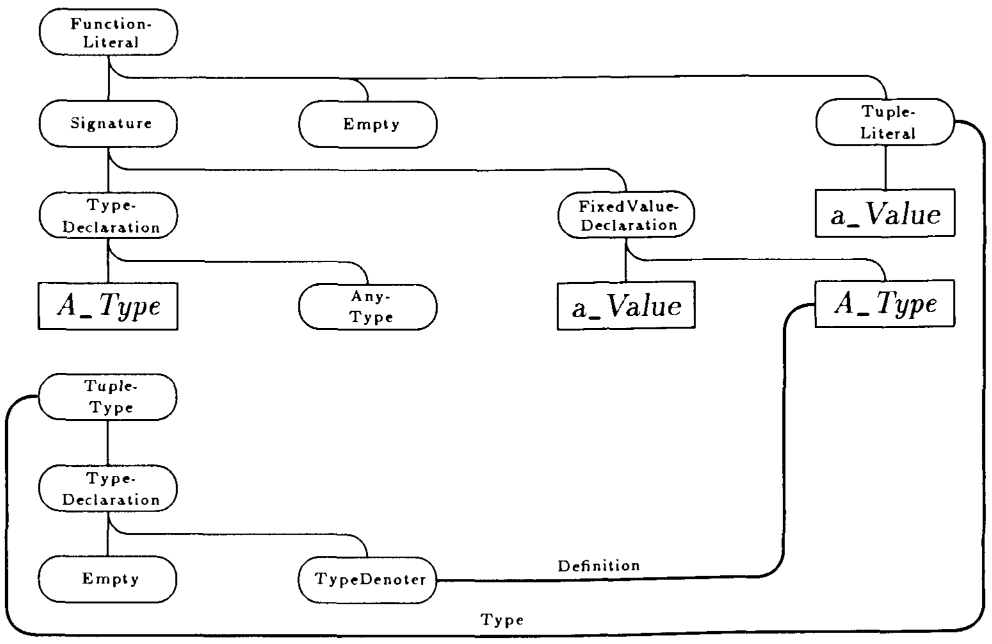
The type of the tuple-literal in the body is given by a tuple-type as

Tuple : *A_Type* **end**

since the type of a_*Value* is given by *A_Type*. This much is straightforward but how does the type-identifier *A_Type* refer back to its declaration in the signature of the function-literal?

The problem with a derived node is that it is not connected by context-free relations to the tree from which it is derived; for it to refer to the nodes in that tree some mechanism must be provided. I propose the notion of a denoter node, a childless construction node that acts as a placeholder for its definition node and is permitted only in derived nodes. A denoter acts as an anonymous identifier.

The following diagram shows how Acer, with the help of a type-denoter, defines the type of the tuple-literal we saw in the previous example:

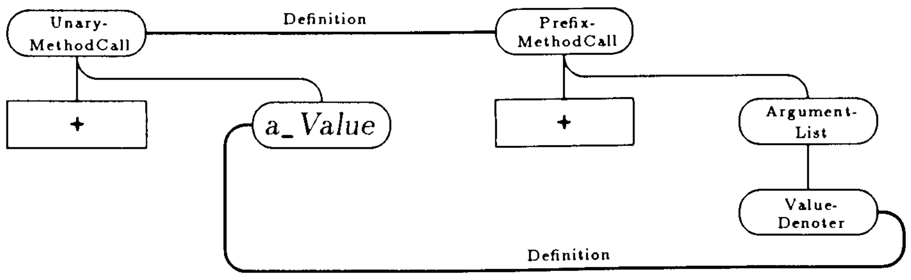


Notice how the type-denoter in the declaration of the tuple-type is used to refer to the type-identifier in the original syntax tree via its definition relation.

What makes a denoter special is the fact that its definition relation must be fixed at creation time. Every other context-dependent relation can be derived from the context-free relations whenever necessary. A denoter's definition can be derived only when creating the derived node that uses it. Conceptually, it is best to think of a denoter's definition relation as the context-dependent equivalent of an nth-child relation.

3.5.3 Definition

In many programming languages certain constructs are defined in terms of other constructs according to rewrite rules. I call this the definition relation, which can be specified using derived nodes to represent the result of each rewrite. The definition relation applies to all expression nodes—an expression is either its own definition or there is some other node that represents its definition. We have already seen the definition relation as it applies to denoters. Another example is the definition of an Acer unary-method-call, which is defined in terms of a prefix-method-call as



Notice the use of a value-denoter to connect the derived node to the original syntax tree via the definition relation.

Many other context-dependent relations are defined for programming languages but, just as the defining-occurrence, type, and definition relations, they can be specified in terms of relations on nodes. More complex context-dependent relations are handled by the provision of derived nodes created to represent semantic objects; denoters are used to permit derived nodes to reference other nodes.

3.5.4 Context-dependent manipulation

The context-dependent relations of a language are derived from the context-free relations so context-dependent syntax is manipulated indirectly by manipulating the context-free syntax; every change to the context-free relations is reflected by changes to the context-dependent relations. Meaningful manipulation is guided by the information embodied in these context-dependent relations, for example, finding all the applied-occurrences of a given identifier or finding all the expressions of a given type. Therefore, to support such manipulation, context-dependent relations must be automatically available.

3.6 Summary

This chapter has described techniques for representing semantic objects as nodes. Node structure is the basis of meaningful program manipulation for both machines and humans. In terms of machine manipulation, a metaprogramming system can be created to manipulate programs according to an abstract data-type that implements nodes. In terms of human manipulation, a language-based editor, like PCAcer described in Appendix B, can be created to manipulate programs viewed conceptually as nodes. In either case, node structure is the basis for manipulation.

To summarize, language design is the design of node structure and program manipulation is the manipulation of node structure. By specifying a language in terms of node structure, a language designer specifies precisely the concepts programmers use to understand and manipulate programs.

Chapter 4

Principles of language design

This chapter describes principles for guiding language design. These principles focus on various manipulation issues and are also presented in [MDC92]. Their application to Acer, in particular, is also discussed.

Each section that follows begins with a statement of principle followed by a discussion. There are 15 principles in total: principles 1–6 deal with the relationship between concrete and abstract syntax; principles 7–11 deal with properties of abstract syntax that significantly affect manipulation; and principles 12–15 deal with program transformation.

4.1 Concrete syntax is just a view

Principle 1 *A language definition should provide a standard abstract syntax in one-to-one correspondence with a standard concrete syntax.*

Meaningful program manipulation is best carried out in terms of an abstract representation. Consequently, this first principle emphasizes abstract syntax as the conceptual foundation of language—concrete syntax is seen as a way of viewing abstract syntax.

To see the importance of syntax consider the representation of positive integers. Roman numerals are an adequate representation, and are certainly better than unary numerals, but Arabic numerals are by far superior. This is because Arabic numerals facilitate semantic manipulation, such as addition and multiplication, through straightforward syntactic manipulation. Thus, although Roman numerals provide an adequate representation, only Arabic numerals provide a manipulable representation.

For a language of programs, as for a language of numbers, supporting manipulation is a fundamental goal. Therefore, a programming language should define a manipulable representation. For Acer I chose to use a GRAMPS-style representation based on the simple notion of nodes as discussed in Chapter 3. Using a GRAMPS-style grammar to define concrete syntax and abstract syntax ensures conformance with the first principle; node structure is designed first and concrete syntax is a view.

Emphasizing the manipulation characteristics of nodes and treating concrete syntax as a way of viewing nodes helps put the relative importance of abstract syntax and concrete syntax in a proper perspective. This should aid in guiding the design of programming languages that are less like Roman numerals and more like Arabic numerals. Merely representing a program is adequate but representing a program so as to facilitate its manipulation is preferable.

4.2 Ambiguity

Principle 2 *The standard concrete syntax of a language should be defined by an unambiguous context-free grammar.*

Concrete syntax provides the means of viewing nodes. Thus, the natural criterion for evaluating concrete syntax is readability. The above principle reflects the notion that a concrete view is readable only if it unambiguously determines node structure.

It may be argued that a parser can use static analysis to disambiguate but this reveals another weakness of the ambiguous grammar: an ambiguous context-free grammar expresses a distinction that can only be resolved in the context-dependent syntax. I contend that a context-free grammar should not express notions that are in the domain of context-dependent syntax.

Consider the problems of supporting an abstract representation in the presence of syntactic ambiguity. Static analysis must be used to disambiguate node structure so the structure of a node (i.e., its node class) could be affected by changes to context. Therefore, node structure would not be an invariant property affected by direct modification only, that is, by replacing, inserting, or deleting children. Node structure would change depending on context. This clearly is a major complication.

Furthermore, the ability to manipulate phrases—syntactically complete textual fragments generated according to some grammar production—would also be complicated since an ambiguous phrase must be parsed with respect to an appropriate context to determine correctly its node structure. Since such an appropriate context is not always available, how should phrases that cannot be disambiguated be handled? And indeed, program manipulation tools routinely manipulate phrases for which no context exists, so it is crucial that syntactic structure can be determined without static analysis. Thus, it is best to simply avoid all these difficulties, as I did for Acer, and define concrete syntax in terms of an unambiguous context-free grammar.

Desirable ambiguities can be handled by *semantic overloading*, that is, by defining a single overloaded context-free structure with various context-dependent interpretations (definitions). For example, in Acer, the value-selection *x.y* has one of two possible definitions. First, if *x* has an abstract-type, such as *vector.Type*, the value selection *x.y* has the prefix-method-call *y.*(*x*) as its definition. Otherwise, *x* must be an aggregate (a tuple, record, or dynamic), in which case the value-selection *x.y* is its own definition. Overloading the semantics of a value-selection thereby achieves the same effect as defining a value-selection and an abstract-value-selection as two different classes of node that have the same concrete syntax. Thus, avoiding context-free ambiguity gains much and loses little since semantic overloading achieves the same effect.

Note that a language designer cannot simply define an ambiguous context-free syntax and expect semantic overloading to resolve all ambiguities. He must design the syntax to provide a syntactic way of distinguishing between various context-dependent interpretations. For example, in Pascal, the syntax specifies that the parentheses of a function-call to a parameterless function be omitted. As a result, an applied-occurrence of an identifier *f* could stand for a function-call. Now if semantic overloading is to resolve this ambiguity, one would like to be able to say that the definition of *f* is *f* () when *f* stands for a function-call, and that the definition of *f* is *f* otherwise. Unfortunately this is invalid because Pascal's syntax does not permit a function-call with empty parentheses. Thus Pascal's syntax does not support the technique of semantic overloading, although the simple change of permitting an empty parameter-list in a function-call would be sufficient to provide such support.

4.3 Phrase ambiguity

Principle 3 *The standard concrete syntax of a language should not contain phrase ambiguities.*

An unambiguous context-free grammar can still contain what will be called phrase ambiguity, that is, syntactic ambiguity resulting from including in the grammar a rule that is an alternation of all other rules. The principle might therefore be restated as, "The standard concrete syntax of a language should contain a rule that is an alternation of all other rules."

(In Acer, this rule is called *arbitrary*, see A.5.3.) This, in combination with avoiding syntactic ambiguity, is sufficient for avoiding phrase ambiguity. The restatement of the principle also reflects the importance of being able to parse arbitrary phrases as well as complete programs.

If a language avoids phrase ambiguity, as does Acer, then a parser, or a human reader, can always determine the node structure of a printed representation, even for phrases out of context. On the other hand, if a language is phrase ambiguous, as are all well known programming languages, then a parser requires a start symbol as well. For example, a Pascal parser cannot simply parse *foo*(*arg*), it must parse *foo*(*arg*) as an expression or as a statement to determine whether it is a function-call or a procedure-call. (This could be avoided by defining instead a single overloaded routine-call.) Thus, the presence of phrase ambiguity complicates not only the implementation of manipulation tools, but also the interface of the parser and hence the user interface of the manipulation tools.

Clearly phrase ambiguity is considered acceptable to most language designers but there are considerable benefits in avoiding it. The ability to parse correctly any textual phrase simplifies the textual entry of nodes; also having to specify intended structure is inconvenient at best. It is true, however, that avoiding phrase ambiguity severely restricts the choice of concrete syntax but there are also costs, perhaps acceptable, in not doing so.

In favor of allowing phrase ambiguity is the argument that similar things should appear similar [Mac87, page 46], for example, a function-call should look like a procedure-call because they are both routines. However, in such cases it also generally makes sense to use semantic overloading to capture the similarities. Furthermore, Acer function-calls are similar in appearance to operator-calls and yet a parser can still distinguish them. For instance, a function-call $f(x)$ is distinguishable from an operator-call $O(T)$ because, in general, a function-call has a value as its first component and an operator-call has a type as its first component. Therefore, avoiding phrase ambiguity does not imply that constructs cannot appear similar.

In fact, constructs can appear similar as long as they can be distinguished by the syntactic class of their components. For example, in Acer, a type-selection, a value-selection, and a literal-selection each consists of a component, a dot, and second component. But a type-selection such as $x.T$ consists of a value and a type-identifier, a value-selection such as $x.v$ consists of a value and a value-identifier, and a literal-selection such as *Color.red* consists of a type and a value-identifier. Thus the three constructs are syntactically distinguishable because type and value expressions are syntactically distinguishable. And ultimately, the syntactic categories for type and value are distinguishable simply because type- and value-identifiers are distinguishable.

A consequence of avoiding phrase ambiguity is the requirement than an empty input must give rise to an empty node. Therefore, to distinguish textually a childless list node from an empty node, every list rule must contain at least one token. With the design of Acer I carried this requirement one step further: every construction rule also must contain at least one token. Therefore, since lexical rules themselves specify tokens, every Acer lexical, list, and construction node gives rise to at least one token in the textual representation. (Recall that an Acer empty node can either be invisible or appear as the token **nothing**.) Because of this syntactic property, it is possible to select any Acer node by simply pointing at one of its tokens in the textual representation. This is particularly important for the interactive manipulation of programs.

It may be argued that these restrictions on concrete syntax are too stringent, particularly requiring all construction rules to contain tokens. Remember, however, that the restrictions are intended to ensure various syntactic properties—the importance a language designer places on these properties determines the importance of conforming. Furthermore, only the standard concrete syntax need conform; alternative views can be specified freely. Also, Acer demonstrates that conformance can in fact be easily achieved and that the resulting concrete syntax is quite readable.

4.4 Comments

Principle 4 *The positions of comments in a language should be defined in the concrete grammar of the language.*

Most modern languages treat comments as white space that is discarded during parsing. However, it should be possible to preserve comments when a program is parsed [Gro89, Kae88], so that a commented form can be regenerated from the node structure. Moreover, since nodes can be modified in arbitrary ways, the original positions of comments may also change. It is therefore important to consider how a parser associates an arbitrarily placed comment with a specific node in the parse tree, and where a pretty-printer writes the comment associated with a node. In addition, if comments are used as annotations for nodes, it is crucial that the association of comments with nodes is unchanged when a node is printed and then parsed back in.

Comments play a central role in a number of situations. Not only do they serve as documentation, but they are frequently used as assertions or compiler directives (pragmas) as well. A well-designed commenting facility provides the capability to annotate nodes with arbitrary textual information. Tools can use this facility not only to receive tool-specific information (e.g., switches for the compiler, assertions for the correctness-prover, formatting hints for the pretty-printer, etc.) but also to produce parsable annotated listings (e.g., a compiler listing, a profiled listing, a cross-referenced listing, etc.).

In Acer, the property that every node owns at least one token in the textual representation provides a simple way of associating comments with nodes: each comment is associated with the node that owns the preceding token. An Acer comment begins with a '%' and continues to the end of the line; several lines of comments may appear (see A.4.2.1). In this way, Acer supports the textual annotation of nodes. Furthermore, because the rule for associating a comment with a node is so simple, there is no need to complicate Acer's grammar with a specification of allowable comment placement.

However, because a node can own several tokens in the concrete representation it may be desirable to specify in the grammar which particular token is to be annotated by a standard unparser. Of course, a user would still be permitted to annotate any token owned by a node. But when a node is printed, the annotations would associate with the one specified token. Such comment placement is not actually specified for Acer because it is irrelevant; different implementations may choose different comment placement. The issue is a matter of taste.

4.5 Macros

Principle 5 *Languages should neither include nor need text-based macros.*

Just as comments constitute a problem for program manipulation if the information they represent is not encoded in the node structure, so too textual macros are a problem. Because they define textual objects rather than syntactic objects, textual macros cannot be manipulated syntactically and thus must be expanded before parsing proceeds. Information about the use of macros is therefore lost, and cannot be reintroduced to print a modified program in terms of its original macros.

Macros provide a great deal of power but the fact that such power is provided in addition to a language's existing abstraction mechanisms seems to imply a language weakness. It is interesting that C++ [Str86] eliminates much of C's [KR78] need for macros by providing for constant definitions and in-line functions. Although text-based macros are unacceptable, a language designer may consider providing syntax-based macros, that is, macros that are parsed first and then expanded as nodes later. In this way, macros are defined as a feature of the language itself and so appear in the abstract syntax. Acer simply avoids the use of macros.

4.6 Incomplete phrases

Principle 6 *A language should have a standard representation for syntactic placeholders.*

Whereas a phrase is a complete expansion of some grammar production, an incomplete phrase is only a partial expansion, that is, it contains non-terminal symbols. An incomplete phrase can be expressed by substituting a placeholder or metavariable for each unexpanded syntactic component.

Metavariables have a number of applications. The need for metavariables lies at the heart of template-driven syntax editors [BS86,TR81], which facilitate program construction by the expansion of placeholders. Furthermore, incomplete phrases can be viewed as patterns, making them useful for pattern-matching and for specifying transformations [DGKLM84].

Acer supports syntactic placeholders without the inclusion of a special metavariable syntax. A lexeme is concise enough to stand as its own placeholder (a metavariable would have a syntax at least as complicated). A list may be empty, thereby acting as an appropriate placeholder. And a construction acts as a placeholder when appropriate placeholders for each of its components are provided.

Construction placeholders are the most complex case since such a placeholder must also specify placeholders for each of its components. However, Acer's syntax is such that a placeholder for any construction component can be provided by either a lexeme, a childless list, or an empty node. Therefore, only two levels of placeholder need ever be provided so Acer's syntax, as it stands, provides adequate support for placeholders.

Moreover, the notion of metavariables (i.e., named placeholders) is supported by Acer's comment facility, which can be used to annotate placeholders as metavariables. For instance, the following function-literal expresses a pattern

```
function () %*Signature
  ? %*Body
 end
```

in which the signature is named 'Signature' and the body is named 'Body.' Thus Acer's syntax need not be complicated with a special metavariable syntax, annotated placeholders are sufficient.

This concludes the discussion of principles concerning the relationship between concrete and abstract syntax. In summary, the abstract syntax of a language should be designed as a manipulable, node-based representation of the required semantic objects, and concrete syntax should be designed as a phrase unambiguous view of annotated nodes. In addition, careful consideration should be given to ensuring beneficial properties of the concrete syntax, such as the requirement that every node gives rise to a least one token.

4.7 Semantic objects

Principle 7 *All semantic objects should be representable as constructs in the language.*

This most fundamental manipulation principle stems from the fact it is difficult to manipulate that which cannot be represented. The implications of conforming to this principle (or of not conforming) are therefore far-reaching. Consider a typical example, a language that requires every expression to have a type. Should it not follow that all types are expressible in the language? A type is, after all, a semantic object so it too deserves to be represented in the abstract syntax. It is then possible to provide a manipulation system that responds to a query about the type of an expression node by producing the type node that represents it, as does PCAcer.

Depending on a language designer's perspective, the principle can be restated in various ways. In terms of denotational semantics, it requires each semantic domain to correspond to a syntactic domain. In terms of specifying semantics with attribute grammars [Knu68], it requires each node attribute to be a node, a basic value (such as an integer, character or string), or a relation on nodes and basic values. In crude implementation terms, it requires that basic values, along with data structures for representing nodes and for representing relations on nodes and basic values, be sufficient for representing, in a natural way, context-dependent syntax. Common to all this is the need to minimize the complexity of manipulating semantics in terms of syntax.

Acer conforms to the principle by ensuring that every identifier has a defining-occurrence, that every expression node has a definition, that every expression has a type, and, most importantly, that every defining-occurrence, definition, and type is actually represented as a node—either as a node appearing in the original parsed syntax tree or as a node created for that purpose (see A.3). Thus Acer goes one step beyond the statement of principle so that not only are all semantic objects representable as nodes but all semantic objects are, in fact, so represented.

4.7.1 Denotation and kind

As extensions of the definition relation and the type relation, Acer specifies a denotation relation (see A.3.2) and a kind relation (see A.3.3). The denotation of a node is the final node that results from repeated queries as to the definition of a node, the definition of that definition, and so on. Similarly, the kind of an expression is the final node that results from repeated queries as to the type of an expression, the type of that type, and so on.

In a correct Acer program the kind relation is equivalent to applying the type relation twice because the type of an expression is a type and the type of a type must be a type that is its own type. For the denotation relation, however, the definition relation must be applied an arbitrary number of times to reach a node that is its own definition. This can even lead to circularity. For example, in the following (incorrect) Acer program

```
{let x be y
 let y be x
```

the definition of the *y* in the first binding is the *y* in the second binding, the definition of that *y* is the *x* in the second binding, the definition of that *x* is the *x* in the first binding, the definition of that *x* is the *y* in the first binding, and so on. An Acer manipulation system must detect such invalid circularity (see discussion of implementation techniques in C.19); for this example, the denotation of each identifier is given by the special value *error* (see A.25).

4.7.2 First-class values

Another aspect of Acer's design, which goes beyond the statement of principle, is that all ground-level semantic objects are represented as first-class values. Functions, modules, and exceptions are all first-class; each is supported by a value-expression syntax for expressing its literals and a type-expression syntax for expressing its types.

Such uniform treatment not only enhances program manipulation by eliminating the need for special-purpose support, it enhances the ability of programs to manipulate ground-level objects. For instance, in Modula-2 one cannot write a function that generates a module based on the values of parameters, but in Acer one can write such a module-generating function. Thus, including ground-level semantic objects as values supports both program manipulation and the ability of programs to manipulate semantic objects. Indeed, it is difficult to separate the notion of supporting program manipulation from the notion of a programming language supporting the manipulation of its semantic objects.

4.8 Scoping

Principle 8 *The scope rules of a language should be simple.*

The notion of associating with a semantic object a name and thereafter referring to that object by name is universal to all programming languages. Static scoping with nested block structure is the standard manner in which this notion is syntactically represented. The notion gives rise to the defining-occurrence relation, the primary means by which distant nodes in a syntax tree come to be related. Since other relations rely heavily on the defining-occurrence relation, it must be a simple relation that is easy to compute.

Acer specifies scoping in terms of two concepts (see A.3.1). First is the concept of the *name-layer*. Certain classes of Acer node, such as binding-lists, signatures, and tuples, contain constructs, such as bindings and declarations, that introduce defined-identifiers. The name-layer of a node from such a class is the set of defined-identifiers introduced by its immediate children. A name-layer may not contain duplicate spellings.

Name-layers are searched during *identifier-lookup*, Acer's second scoping concept. Identifier-lookup involves a recursive traversal from child to parent in which at each step the class of the parent, given that lookup is at the particular child, determines the name-layer(s) that must be searched and whether lookup should terminate. By default, no name-layers are searched and traversal continues with the parent.

Intuitively, Acer's scope rules are specified in terms of searching, and the nature of this search is very simple indeed. In Acer, the order of declarations and bindings does not affect scope since scope is defined in terms of name-layers—a name-layer is a set of nodes, and hence has no order (although it may be ordered according to spelling to speed searching). Supporting arbitrary forward reference allows recursive dependencies to be naturally expressed and does not complicate scoping. But it does imply the burden of dependency analysis for detecting erroneous denotation and kind relations (see A.7.2). Thus Acer's scope rules are simple but dependency analysis is somewhat complex.

The approach to scoping in Acer may be contrasted with that in Quest, which requires recursion to be explicitly specified with recursive binding constructs. For instance, in Acer, the recursive factorial function can be defined as

```
let ! be function (x : Integer)
            if {x <= 0} then 1 else {x * !({x - 1})} end
         end
```

just like any other function. But in Quest, a special form of recursive value-binding must be used:

```
let rec ! be function (x : Integer)
                if {x <= 0} then 1 else {x * ! ({x - 1})} end
              end
```

Similarly, in Quest, recursive type-bindings must be used to define recursive types. Since the restrictions imposed by Acer's dependency analysis are the same as those imposed by Quest's recursive bindings, it can be argued that Acer's approach is simpler and more direct—recursion does not require additional syntactic constructs which complicate manipulation.

The advice to language designers, then, is that scoping should be flexible enough to allow recursion to be directly expressed and hence that order of definition or declaration should not affect scope. After all, recursion is an essential programming language feature that should not require special syntactic support. Of particular concern are constructs such as Pascal's forward-declarations, whose sole purpose is to circumvent a deficiency in the scoping.

4.9 Locality of objects

Principle 9 *A language should allow an object to be declared local to the scope in which it is actually required.*

One of the primary advantages of nested block structure is the ability to introduce objects at a nesting level that makes them visible only over the region in which they are used. This is beneficial for program manipulation because new objects can be introduced locally at the point of modification, thereby limiting the area affected by the change.

In terms of program manipulation, the ease of introducing a new variable is particularly important because it is one of the most commonly performed modifications; variables that act as temporary storage for a value must frequently be introduced into programs. Thus Acer provides type- and value-blocks, which can be arbitrarily nested. For instance, suppose we have the expression

```
{{{x * x} + {y * y}} / {{x * x} - {y * y}}}
```

We could optimize this expression as follows:

```
{let xx be {x * x}; let yy be {y * y}; {{xx + yy} / {xx - yy}}}
```

Such a transformation is purely local, affecting nothing outside of the expression itself.

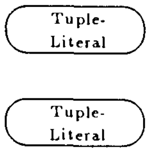
4.10 Referring to objects

Principle 10 *A language should allow references to existing objects to be easily introduced.*

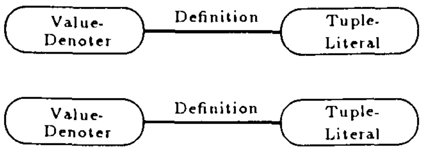
When a program is modified, it is frequently necessary to refer to a specific semantic object at a given point in the program. Program manipulation is therefore simplified if new references to such objects can be introduced without global changes to the program. With nested block structure the objects visible at the defining point of some particular object are generally visible where that object is visible. Therefore, nested block structure tends to make objects visible where they are actually needed.

Acer's nested block structure provides complete support for the principle. In fact, given a node (representing a semantic object) and a target location, an Acer manipulation system can automatically create a *definition-copy-at* (see A.5.2.2), a node representing the same object as the original node but expressed in terms of the scope of the target location. Naturally, a definition-copy-at is not possible for all node/target-location combinations, since objects can be out of scope, but it is possible whenever it is sensible. More on this in the following two sections.

Acer's denoters also provide support for referring to existing objects, although this support is not directly available at the source-level. However, when writing a metaprogram, i.e., a program that manipulates Acer programs according to the node representation described in Chapter 3, it is possible to create a denoter to an existing node and to use that denoter as if it were an identifier referring to that node. For example, suppose a metaprogram created two empty tuple-literal nodes:

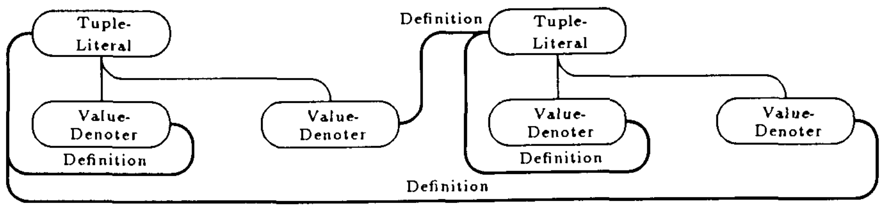


A denoter to each of these nodes could then be created:



and each denoter could be inserted[1] as both the first child of its own tuple-literal and the second child of the other tuple-literal:

[1]Note that when an attached node is inserted, it is copied, and that when a denoter is copied, a denoter with the same definition as the original is created.



If either of these two tuple-literals is then to be inserted into a source program, in which denoters are not permitted, definition-copy-at can be used to automatically express them as:

```
{let unnamed1 be tuple unnamed1, unnamed2 end
 let unnamed2 be tuple unnamed2, unnamed1 end
 unnamed1}
```

The result is a node with the semantic structure of the original but expressed without denoters.

So Acer's denoters allow programs to be expressed and manipulated without regard to the normal limitations of scope. In addition, any object expressed in terms of denoters can be automatically re-expressed using normal scoping rules.

4.11 Accidental information hiding

Principle 11 *Names should only be hidden using information hiding constructs specifically designed for the purpose.*

Nested block structure generally conforms with this principle; the only exception is the unintentional hiding of a name when the name is reused for a different purpose in a nested scope. This creates holes in the outer identifier's scope.

One solution would be to disallow nested renaming, which is certainly possible for languages like Modula-2. However, in Acer, choice of identifier can affect subtyping, as with the names of the declarations of a tuple-type, so identifier choice cannot be arbitrarily restricted. Therefore, Acer gets around the problem of unintentional hiding by providing a notation for referring to the hidden identifiers, namely the reused-identifier. Acer's definition-copy-at facility is possible precisely because reused-identifiers can be used to reference identifiers that are hidden by renaming. For example, suppose we have the following function:

```
function (x : Integer)
  tuple let x be ? end
 end
```

and that we want to express the x in the signature in place of the ? in the tuple-literal. Applying definition-copy-at with the x in the signature as the source and the ? in the literal as the target and replacing the ? with the result produces the following:

```
function (x: Integer)
  tuple let x be x'[1] end
 end
```

Note that Ada deals with the problem of unintentional hiding in two ways. First of all, Ada allows identifiers to be overloaded, using various type restrictions to determine the defining-occurrence referenced by a particular applied-occurrence. Unfortunately, this solution does not work for *homograms*, i.e., identifiers that have indistinguishable types. The second approach is to allow scopes to be named and to use qualification to reference a particular defining-occurrence. However, not all scopes are named so during automated manipulation arbitrary new scope identifiers must be introduced. This results in local transformations that produce non-local changes. Furthermore, the problem of hiding may also occur for the scope name. Ada's approach is complex, thereby hindering manipulation.

4.12 Intentional information hiding

Principle 12 *If an information-hiding construct makes visible the name of an object, it should also make visible the names of the objects referenced in its definition, unless the definition constitutes representation-dependent information.*

The methodology of abstract data types requires the ability to hide representation-dependent information, e.g., the implementation of a type or routine. Such information hiding conflicts with the principle of ensuring that references to objects can be easily introduced; but it does so precisely to prevent manipulation that depends on the hidden objects. Information hiding is such an important aspect of software engineering that a programming language must support it.

Acer supports information hiding with the provision of declarations, which specify the type of an object but leave the definition unspecified. A declaration provides enough information to use the declared object without revealing its static definition. In addition, unhiding definitions is also simple in Acer because a reference to a declaration can always be replaced by a reference to a binding providing a definition.

Although most modern languages provide a special module construct, Acer does not, because modules are represented as tuples or records. Therefore, Acer does not encounter the problem of unintentional hiding that can result from the import/export rules that control intermodule visibility. For example, in Modula-2, it is possible to export a variable without exporting its type. Acer's simple nested block structure ensures conformance with the principle.

This concludes the discussion of principles relating to properties of the abstract syntax. To summarize, then, semantic objects should be represented as nodes and these nodes should be organized so that they can be named and made visible over the regions in which they are needed. The visibility rules should be simple, should permit references to be introduced easily, and should prevent unintentional hiding.

4.13 Unfolding

Principle 13 *There should be a straightforward general way of replacing a call to a user-defined routine with an equivalent in-line version of the routine body.*

This principle reflects the importance of supporting semantics-preserving manipulation, i.e., transformation. Arbitrary syntactic manipulation, within the constraints imposed by context-free syntax, is supported by any language so the important question is whether simple syntactic changes can be used to effect precise semantic changes. Surely the goal of supporting program manipulation in general requires that the important special case of transformation be supported in particular.

Unfolding a routine, a typical transformation performed during program optimization, is only possible if the effects of calling a routine (e.g., parameter binding) can be achieved at the calling point. Therefore, conforming with the principle ensures that a suitable in-line equivalent is provided for each mechanism available to routines.

To see how Acer supports unfolding consider a function-call with a function-literal as its function, for example,

function $(x : \mathit{Integer};\ y : \mathit{Integer})\ \{x + y\}$ **end** $(10, 20)$

Such a function-call can always be replaced by a value-block with a body given by the literal's body and a binding-list determined by the argument-list and the signature, for example,

$\{$**let** $x : \mathit{Integer}$ **be** $10;$ **let** $y : \mathit{Integer}$ **be** $20;\ \{x + y\}\}$

Now, to provide complete support for unfolding, all that is still required is the ability to express function-literals at their calling points. But this ability is provided, in conformance with the principle that references to existing objects should be easy to introduce, by Acer's definition-copy-at facility. Therefore, Acer provides complete support for unfolding.

Note that the static evaluation of an operator-call, i.e., determining the definition of an operator-call, is essentially an unfolding transformation. For example, given *Dyadic* defined as

```
let Dyadic be
  Operator (BaseType :: Any; ResultType :: Any)
    Function ( : BaseType;  : BaseType) ResultType end
   end
```

unfolding the operator-call *Dyadic* (*Integer*, *Boolean*) results in:

```
{let BaseType be Integer
 let ResultType be Boolean
 Function ( : BaseType;  : BaseType) ResultType end}
```

And this type is equivalent to the definition of the operator-call.

4.14 Folding

Principle 14 *There should be a straightforward general way of replacing any expression with an equivalent user-defined routine.*

Folding, encapsulating expressions as parameterized routines, is a transformation of fundamental importance. It lies at the heart of abstraction. Whereas support for manipulation is often overlooked in language design, support for folding cannot be overlooked. However, because support for abstraction usually provides support for folding, support for folding is not often recognized as a relevant goal.

Acer supports folding of type and value expressions as follows. A type T can be folded as a type-operator used in type-call:

```
Operator () T end()
```

And similarly a value v can be folded as a function-literal used in a function-call:

```
function () v end()
```

However, for values, if v appears in a context requiring a reference or pointer to v rather than just the value referenced by v, as does v in

{*v* **becomes** *something*}

then *v* must be folded either as

{**function** () **reference** (*v*) **end**()@ **becomes** *something*}

or as

{**function** () **pointer** (*v*) **end**()@ **becomes** *something*}

The second form is permitted only if **pointer** (*v*) is valid (see A.30). Any Acer type or value can be folded in these ways and hence anything that an expression can do a routine can do.

Folding in the form that has just been shown is, of course, of little direct use. The real value of folding lies in the reuse of routines. The routines resulting from the folding above are used but once.

To use a folded expression more than once it must be hoisted to a higher scope, certainly higher than the call in which it is used. Parameterization is necessary whenever a routine is hoisted beyond the scope of some object referenced in its body—that object must then be passed as a parameter instead of simply being visible. Acer's signatures support hoisting by providing for both type and value parameters. Clearly, Acer's support for polymorphism provides better support for hoisting than do many programming languages.

As an example of hoisting, consider folding the applied-occurrence of *x* in the block

```
{let construct be
    function (T :: Any; x : T; y : T)
      tuple x, y end
     end;
 construct (Integer, 10, 20)}
```

which results in

```
{let construct be
    function (T :: Any; x : T; y : T)
      tuple function () x end (), y end
     end;
 construct (Integer, 10, 20)}
```

Hoisting the new function-literal up one level results in

```
{let construct be
    function (T :: Any; x : T; y : T)
      {let unnamed be function () x end;
       tuple unnamed (), y end}
     end;
 construct (Integer, 10, 20)}
```

and hoisting it up three levels results in

```
{let unnamed be function (T :: Any; x : T) x end;
 let construct be
   function (T :: Any; x : T; y : T)
     tuple unnamed(T, x), y end}
    end;
 construct (Integer, 10, 20)}
```

The ability to parameterize over types affords great expressive power.

4.15 Low-level manipulation

Principle 15 *A good target language should support both high-level manipulation and low-level manipulation.*

The single most important program manipulation activity is translation, a semantics-preserving transformation from a high-level language to a low-level language better suited for direct execution. In [Mer87], the implementation of a compiler as a series of source-to-source transformations is explored. Using the source language (Modula-2) as the intermediate language, translation is carried out by source-to-source transformations that gradually replace high-level constructs with equivalent lower-level constructs until the assembly-level is reached. The work in [Mer87] is related to [KH89] except that Modula-2 itself is used as the intermediate language rather than a special-purpose intermediate language. Compilation as a source-to-source transformation is also examined in [FM91] and in [LC87]. Ideally, a language should provide limited access to the low-level language so that implementation-dependent manipulation is possible [Tur82].

Acer provides a single construct for accessing low-level features, the code-patch, which is treated as a value. Syntactically, a code-patch is a list of expressions. The first expression must be a type to indicate the type of the code-patch. Any trailing expressions are interpreted in an implementation dependent way as the data and instructions of some machine.

To achieve greater portability, the PCAcer compiler translates to an abstract machine language, which is a simplified form of MC68000 code [Cas84]. Like an MC68000, there are various registers. The contents of the *d0* register, interpreted as an integer, can be accessed by the following PCAcer code-patch.

```
code Integer; d0 end
```

The first expression, of course, represents the code-patch's type and the last expression, for PCAcer, represents the effective-address of the code-patch's value, register *d0* in this case.

A PCAcer code-patch can also include machine instructions, which are bracketed by the initial type and the final effective-address. For example, a function that moves the contents of register *d0* to register *d1* could be defined as

```
let move_d0_to_d1 be
  function () code Void; move(d0, d1); #(0) end end
```

Notice that this code-patch reveals how the void-literal is represented in PCAcer, namely as a machine zero.

With code-patches, every implementation of Acer provides a convenient notation for expressing machine dependencies. Furthermore, since code-patches are represented in terms of existing syntax, Acer's context-free syntax is the same for all implementations. Only the context-dependent interpretation of a code-patch's trailing expressions changes from implementation to implementation.

It should not be surprising that existing syntax can be so easily reused to represent the semantic objects of another language. A programming language, after all, must be good at representing objects.

4.16 Summary

This chapter has examined three general areas of language design that influence the nature of program manipulation. It has been stressed that the concrete syntax should serve as an abstract syntax so that meaningful semantic manipulation can be carried out as simple syntactic manipulation. Next, scoping and typing were explored, emphasizing important properties such as the ability to introduce easily references to objects. Finally, issues involving support for folding and unfolding were considered, issues which are the foundation for semantics-preserving transformations in general.

To achieve a useful correspondence between semantic structure and syntactic structure, the design of a target language should begin with a classification of its objects, e.g., variables, types, values, expressions, statements, procedures, modules, and so on. Next, the attributes associated with each class of object should be identified. Some of these will be defined and others derived, e.g., a variable has a name and type that are defined and a storage location that is derived. For each different class of object, a grammar production to represent its defined attributes should be included in the abstract syntax. Statically scoped nested block structure should then be used to integrate the abstract syntax for the objects into a framework in which all hiding is intentional rather than accidental. Finally, the abstract syntax should be carefully augmented with keywords and punctuation to obtain a concrete syntax with appropriate manipulation characteristics.

Chapter 5

The implementation of Acer

5.1 Implementing support for manipulation

In many ways, enhancing support for program manipulation is synonymous with simplifying the metaprogramming system that implements the associated language. A metaprogramming system, after all, implements an abstraction for manipulating target-language programs as host-language data objects. Hence, language complexity is reflected directly by metaprogramming system complexity, wherein it impedes manipulation.

Of course, a degree of complexity is inherent in any language, so complexity is reduced not by simply eliminating language features, although this too may be useful, but by rigorously ensuring a uniformity of features. This is the general notion of orthogonality, that is, that features should interact in systematic and predictable ways. In Chapter 4, detailed principles were presented that elaborated on this notion and other ways of enhancing support for manipulation.

A good metaprogramming system acts as a unifying basis for the implementation of all tools; independently designed tools may have conflicting views of the language. Certainly the work on DIANA [GWEB83] and the work on object-oriented metaprogramming systems [MN88] reflects this view. As does the work of Kurn on the G language [Kur91], which is a high-level language for writing metaprograms.

A great deal of programming environment research has been focused on the design of end-product tools, i.e., a programming environment consisting of an editor, a compiler, and a debugger. In addition, much of this research deals with support for existing languages. As a result, effort is expended supporting language features that could be better designed had support for manipulation been considered, e.g., syntactic ambiguity incurs costs that can be avoided.

The approach advocated here is that particular attention should be focused on the one tool used to implement all other tools, the metaprogramming system. Current research is valuable from this perspective because it clarifies what a metaprogramming system must provide to implement effectively an end product [DMS84].

What is the advantage of an approach that stresses the importance of a metaprogramming system over that of the final environment it implements? First, it makes clear the feed-back mechanism that should exist between environment design and language design—as the design and implementation of basic manipulation support progresses problematic language features are identified and redesigned. This synergy frees designers to spend their time designing more powerful kinds of support rather than waste their time designing ingenious techniques for supporting poorly designed features.

And second, the approach brings programmers back into the sphere of influence by providing an abstraction for implementing application-dependent manipulations, e.g., analyzers to check for correctness, optimizers to make data-abstraction more efficient, formatters to present data-abstractions more readably, transformers to implement new editing commands, generators to construct automatically programs or program templates, and so on. For a general-purpose language, which supports a wide variety of applications, the range of application-dependent manipulations is potentially huge. And a sophisticated programmer, already skilled at manipulating abstract values with programs, can readily use a metaprogramming system to write programs that manipulate programs.

5.2 Acer's metaprogramming system

To demonstrate the feasibility of the metaprogramming approach, a metaprogramming system for Acer, as well as a visual interactive editor and a compiler, have been implemented. The implementation, which uses Turbo Pascal running under MS-DOS, is described in Appendix C; the user's manual for the editor and compiler is described in Appendix B.

Ideally, however, rather than a Pascal host-language metaprogramming system for Acer, an Acer host-language metaprogramming system for Acer should be provided. This unfortunately involves a boot-strap step not yet undertaken, partly due to the memory limitations imposed by MS-DOS. Nevertheless, the essential facilities of an Acer host-language metaprogramming system for Acer will be described as if it actually exists. This should not be considered misleading because the facilities of this system are supported by the Pascal host-language system described Appendix C. After all, both Pascal and Acer support data abstraction and so the abstract node representation described in Chapter 3 can be supported in either language.

However, because Acer supports useful mechanisms not available in Pascal, such as iterators and accumulators, symbolic value-identifiers, exceptions, and so on, the Acer host-language metaprogramming system to be described contains features not directly available in the Pascal host-language version. For example, the Acer host-language lexical analysis and parsing functions make use of exceptions and also iterators and accumulators, but the Pascal host-language versions have a less elegant interface (see C.12 and C.13). Such differences highlight the advantages of Acer over Pascal, and since Appendix C gives a precise description of the Pascal host-language metaprogramming system, the two versions can be compared.

The interface to the Acer host-language metaprogramming system for Acer is provided by the types *TokenClass*, *Token*, *NodeClass*, and *Node*, and the modules *token*, *node*, and *grammar*. The types *TokenClass* and *NodeClass* are defined as option-types that categorize tokens and nodes, respectively; the types *Token* and *Node* are abstract-types encapsulated by the modules *token* and *node*, respectively. The module *grammar* provides generic access to Acer's grammar.

One of the great advantages of a GRAMPS-style metaprogramming system, as we shall see in the sections that follow, is that much of its functionality is evident from the grammar. Thus the grammar serves as documentation for specifying the facilities used to carry out manipulation.

5.3 Concrete syntax

Let us begin by looking at how concrete syntax is handled. Clearly, a metaprogramming system must support the conversion of a stream of ASCII characters to a stream of tokens and the conversion of that stream of tokens to a node. Conversely, it must also support the conversion of a node to a stream of tokens and the conversion of that stream of tokens to a stream of ASCII characters.

This section describes how the conversion between character streams and token streams is supported by the types *TokenClass* and *Token* and the module *token*. The section that follows will deal with nodes and the conversion between tokens and nodes.

ASCII characters are supported, in Acer, by character-literals and the module *character* and since streams are supported as iterators, no additional support for characters is necessary.

5.3.1 Token

Acer tokens, on the other hand, are supported as follows. First, the type *TokenClass* is defined as an option-type that contains the name of each class of token:

let *TokenClass* **be Option** *an_accumulation*, ..., *a_when* **end**

(Each token name is prefixed by a_ (or *an_*) to distinguish it from an Acer keyword.) And second, the type *Token* is defined as an abstract-type:

let *Token* **be** *token.Type*

implemented by the module *token*:

```
token:
  Tuple
    Type: Any
    class: Function (aToken: Type) TokenClass end
    spelling: Function (aToken: Type) String end
    x: Function (aToken: Type) Integer end
    y: Function (aToken: Type) Integer end
    make:
      Tuple
        an_accumulation: Function (x:Integer; y:Integer) Type end
        ...
        a_valueIdentifier:
          Function (spelling:String; x:Integer; y:Integer) Type end
       end
    error:
      Exception (Tuple message: String; x: Integer; y: Integer end)
    analyze: Function (aCharacterStream: Iterator (Character))
                Iterator (Type)
              end
    unAnalyze: Function (aTokenStream: Iterator (Type))
                 Iterator (Character)
               end
  end
```

This module indicates that each token *t* of type *Token* has a class *t.class*, a spelling *t.spelling*, a column position *t.x*, and a row position *t.y*.

Each token is derived from a character stream, but the stream is not strictly linear for it contains ASCII formatting characters that imply the existence of discrete lines. The first character of a character stream, therefore, is at column 1 row 1; each subsequent character occurs at increasing column position, except that a carriage-return sets the next column position to 1 and a line-feed increments the next row position, leaving column position unchanged. In this way, tokens can be considered to have column and row positions.

To create a token, *make* functions are provided, one for each class of token. If the token is a lexical token, such as an identifier, the *make* function requires a spelling and a position, otherwise, it requires only a position because the spelling can be deduced from the class.

Finally, to support conversion between character streams and token streams, the module *token* provides the functions *analyze* and *unAnalyze*, as well as the exception *error*. The function *analyze* requires a character iterator and yields a token iterator that raises *error*, with an associated error message and position, when it cannot produce a valid token.

The inverse of *analyze* is provided by *unAnalyze*, which takes a formatted stream of tokens and yields a character stream. The position information for tokens is used to generate appropriate white space, i.e., spacing, indentation, and line-breaks. Determining token position is the responsibility of an unparser, which is provided by the abstraction for dealing with nodes.

The module *token* could provide many other useful functions as well. For example, it could provide a record named *isa* containing a recognizer for each class of token, e.g., instead of writing {*aToken.class* is *TokenClass.x*} to determine whether *aToken* is of class *x* one would write *token.isa.x* (*aToken*). Little would be gained by further elaborating on such non-essential features so we shall move on now to the handling of context-free syntax.

5.4 Context-free syntax

The types *NodeClass* and *Node* and the module *node*, support the manipulation of nodes much like the types *TokenClass* and *Token* and the module *token* support the manipulation of tokens. The module *grammar* provides generic access to Acer's abstract grammar. It is described first.

5.4.1 Grammar

The type *NodeClass* is defined as an option-type that contains the name of each class of node:

```
let NodeClass be
   Option accumulation, accumulationList, ..., whenCondition end
```

The names are derived directly from Acer's grammar, one for each lexical, construction, or list rule. Remember that alternation rules define node categories not node classes.

The module *grammar* provides further information about Acer's (abstract) grammar as follows:

```
grammar:
   Tuple
      spelling : Function (aNodeClass : NodeClass) String end
      lexeme : Set (NodeClass)
      construction : Set (NodeClass)
      list : Set (NodeClass)
      abstractType : Set (NodeClass)
      ...
      whenBranch : Set (NodeClass)
      error : Exception (String)
      numberOfComponents :
         Function (aConstruction : NodeClass) Integer end
      nthComponentDomain :
         Function (aConstruction : NodeClass; n : Integer)
            Set (NodeClass)
          end
      nthComponentName :
         Function (aConstruction : NodeClass; n : Integer) String end
      baseDomain : Function (aList : NodeClass) Set (NodeClass) end
    end
```

The spelling of a node class *nc* is yielded by *grammar.spelling* (*nc*). Whether *nc* is a lexeme, construction, or list is determined by testing whether *nc* is a member of either *grammar.lexeme*, *grammar.construction*, or *grammar.list*; it must be a member of precisely one.

The set of node classes specified by an alternation rule a is yielded by *grammar.a*. For a construction rule *c*, *grammar.numberOfComponents* (*c*) yields the number of construction components, *grammar.nthComponentName* (*c*, *n*) yields the name of the nth component, and *grammar.nthComponentDomain* (*c*, *n*) yields the set of node classes that may appear as the nth component—if this set includes *NodeClass.empty*, the component is optional. For a list rule *l*, *grammar.baseDomain* (*l*) yields the set of node classes that may appear as children.

In this way, *grammar* provides all the information concerning Acer's abstract grammar. Note that the exception *grammar.error* is raised with an error message if one of the above functions in incorrectly applied.

5.4.2 Node

The type *Node* and the module *node* provide support for the manipulation of nodes. The type *Node* is defined as an abstract-type:

```
let Node be node.Type
```

implemented by the module *node*:

```
node:
  Tuple
    Type :: Any
    theUnattachedEmptyNode: Type
    class: Function (aNode: Type) NodeClass end
    parent: Function (aNode: Type) Type end
    position: Function (aNode: Type) Integer end
    comment: Function (aNode: Type) Pointer (List (String)) end
    length: Function (aNode: Type) Integer end
    error : Exception (String)
    index1 : Function (aNode: Type; n: Integer) Type end
    accumulator: Function (aNode: Type) Type end
    ...
    =: Function (x : Type; y : Type) Boolean end
    <: Function (x : Type; y : Type) Boolean end
    make: Record ... end
    construct : Function (aConstructionOrList: NodeClass)
                    Accumulator (Type, Type)
                  end
    isa: Record ... end
    parse: Record ... end
    unParse: Record ... end
    copy: Function (aNode: Type) Type end
    replace: Function (source: Type; destination: Type) Void end
    exchange: Function (x: Type; y : Type) Void end
    delete: Function (aNode: Type) Void end
    insert: Function (destination: Type;
                      position: Integer;
                      element: Type) Void end
    ...
  end
```

Every node x has a class *x.class*, a parent *x.parent*, a position *x.position*, a comment *x.comment*, and a length *x.length*, where the length of a lexeme is 0. The parent of an unattached node is a special node of class empty called *theUnattachedEmptyNode*. (It is special because it is its own parent.) The position of an unattached node is 0.

For accessing the nth child of a node x, presumably a construction or list node with at least *n* children, the notation x[n] is used. As a convenience, negative indexing selects children in reverse order, e.g., x[-1] selects the last child.

A child of a construction x can also be selected using its component name *c* as x.*c*. Thus, for each component name in the grammar, a function by that name is included in *node*. (They are not all shown in the above.) Note that since different constructions can use the same component names, each selector function may be correctly applied to more than one class of node.

Two nodes x and *y* can be compared for structural equality using {x = y} and for lexicographic order using {x < y}. Of course, #, <=, >, and >= can be provided as well.

For constructing nodes, various *make* functions are provided, one for each class of node:

```
make:
  Record
    accumulation:
      Function (accumulator: Type; accumulationList: Type) Type end
    accumulationList: Function () Type end
    ...
    valueIdentifier: Function (spelling : String) Type end
  end
```

Each function has the same name as the class of node it constructs: for a construction, the function requires a node for each component; for a list, the function requires no components; and for a lexeme, the function requires a spelling. Since only empty lists can be directly constructed with *make* functions, a *construct* function is provided as well. This function, given a construction or list node class, yields an accumulator that makes the specified class of node. For example,

node.construct (*NodeClass.nc*) ([x, *y*, *z*])

makes either a construction with three children or a list with three children. Whenever an attached child is provided, it is automatically copied.

It is also convenient to have *generic make* functions. For example, *node.make.identifier* makes either a type-identifier or value-identifier, depending on the spelling; *node.make.call* makes either a function-call or operator-call, depending on the class of the first argument; and *node.make.selector* makes either a type-selection, a value-selection, or a literal-selection, depending on the class of each argument. Whenever several semantically similar constructs are distinguishable by the structure of their components, a generic constructor is possible. This possibility could be indicated in a GRAMPS-style grammar by grouping the constructs as an alternation and using the name of that alternation as a constructor name. This is possible with blocks for instance. But for declarations a generic constructor is not possible because a fixed-value-declaration is indistinguishable from a variable-value-declaration given only the component classes.

If any of the above functions are incorrectly applied, the exception *node.error* is raised with an error message.

For easy recognition of nodes, *node* provides *isa* recognizers, one for each class and category of node:

```
isa:
  Record
    accumulation: Function (aNode: Type) Boolean end
    ...
    arbitrary: Function (aNode: Type) Boolean end
    ...
   end
```

In general, for a node class *nc*, *node.isa.nc* (*x*) is equivalent to {*x.class* **is** *NodeClass.nc*} and for a node category *cat*, *node.isa.cat* (*x*) is equivalent to:

{*x.class member grammar.cat*}

For parsing nodes, *node* provides various parse functions:

```
parse:
  Record
    error: Exception (Tuple message: String; x: Integer; y: Integer end)
    tokens: Function () Accumulator (Token, Type) end
    string: Function (aString: String) Type end
    strings: Function () Accumulator (String, Type) end
    ...
   end
```

The exception *node.parse.error* is raised, along with an error message and a position, when parsing fails. The primary parsing function is *tokens*, which yields an accumulator that consumes tokens to produce a node. It could be used as follows:

```
for t in token.analyze (aCharacterStream) do
  node.parse.tokens () t
 end
```

Many additional parsing functions are provided, e.g.,

```
node.parse.string ("let x be y")
```

parses a string argument to a node. All such additional functions can be defined in terms of the *tokens* parse function.

For unparsing nodes, *node* provides various unparse functions:

```
unParse:
  Record
    tokens: Function (aNode: Type
                      theWidth: Integer
                      theMaximumDenoterWidth: Integer
                      printEmptyNodes: Boolean)
               Iterator (Token)
             end
    ...
  end
```

The primary unparsing function is *tokens*, which yields an iterator that produces tokens. Such an iterator can be passed to *token.unAnalyze* to yield an iterator that produces characters, which can then be printed as desired. In actual fact, the unparsing functions provided will be more flexible than indicated here, e.g., the unparser described in C.13 is parameterized to allow programmers to write customized formatting routines [Cam88]; it can even be customized to print proportional characters. However, we shall not concern ourselves with such details here.

For editing nodes, the functions *copy*, *replace*, *delete*, *exchange*, and *insert* are provided. They behave as one would expect. If the second argument to *replace* or *insert* is attached, it is automatically copied.

Numerous other useful functions are provided as well. For example, given a node x, x.*children* yields an iterator that sequences through the children of x; x.*descendants* yields an iterator that sequences pre-order through the descendants of x; x.*root* yields the root node of x; and $\{x$ *encloses* $y\}$ yields *true* if y is a descendant of x. Although the possibilities are endless, the functions already described are sufficient, they can define all these additional functions. So let us move on now to examine how context-dependent syntax is manipulated.

5.5 Context-dependent syntax

Support for context-dependent manipulation is provided by extending the module *node*:

```
node:
  Tuple
    ...
    type: Function (anExpression: Type) Type end
    kind: Function (anExpression: Type) Type end
    definition: Function (anExpression: Type) Type end
    denotation: Function (anExpression: Type) Type end
    definingOccurrence: Function (anIdentifier: Type) Type end
    attributeParent: Function (aNode: Type) Type end
    ==: Function (x: Type; y: Type) Boolean end
    <<==: Function (x: Type; y: Type) Boolean end
    abstractName: Function (anAbstractType: Type) Type end
    abstractBase: Function (anAbstractType: Type) Type end
    quantifier: Function (anAbstractType: Type) Type end
    definitionCopy: Function (aNode: Type; aLocation: Type;
                              theSubstitutions: Index (Type, Type))
                      Type
                    end
    definitionCopyAt:
      Function (aNode: Type; aLocation: Type) Type end
    validate: Function (aNode: Type) Index (Type, List (String)) end
    global: Function (anIdentifierSpelling: String) Type end
    ...
  end
```

Most importantly, the semantic selector functions *type*, *kind*, *definition*, *denotation*, and *definingOccurrence* are provided. Thus, every expression node x has a type $x.type$, a kind $x.kind$, a definition $x.definition$, and a denotation $x.denotation$; every identifier node x has a defining-occurrence $x.definingOccurrence$. Notice that the functions *type* and *definition* already appear in *node* because they are also component names. Therefore, the meaning of each existing function is further overloaded, e.g., the type of a node can be either a component, as when the node is a declaration, or a semantic attribute, as when the node is an expression.

The idea of an attribute-parent is that an unattached node y that is created by the metaprogramming system to act as the type or definition of some other node x will have x as its attribute-parent. Thus, applying *attributeParent* to a node z yields either *theUnattachedEmptyNode* or the node x that has the root node y of z as its type or definition. A node that has *theUnattachedEmptyNode* as its attribute-parent is not an attribute, a node that does not, is an attribute and cannot be edited. Of course, a node can be an attribute of many nodes, but only the node for which it was created is considered its attribute-parent. (See C.17 for a discussion of the implementation issues involved in the notion of an attribute-parent.)

Subtyping and equivalence information is provided by <<== and ==. The expression {x <<== y} yields *true* if the type x is a subtype of the type y. Similarly, the expression {x == y} yields *true* if the expression x is equivalent[1] to the expression y. Of course, the functions ##, <<, >>, and >>== would also be provided to test for unrelatedness, proper subtype, proper supertype, and supertype, respectively.

Additional semantic selectors, dealing with various semantic aspects of Acer as described in Appendix A, are provided as well. For example, for a node x referring to an abstract-type, x.*abstractName* yields the abstract-name, x.*abstractBase* yields the abstract-base, and x.*quantifier* yields the quantifier. Semantic recognizers are also added to the *isa* recognizers, e.g., *node.isa.variableIdentifier*(x) yields *true* if x denotes a variable-identifier.

The ability to definition-copy a node is provided by the function *definitionCopy*. It takes a node to be copied, a location node, and an index (see C.5) of substitutions. The purpose of the location node is to specify that any identifier used in the definition-copy having a defining-occurrence enclosed by the location should have its definition definition-copied—most often the location node will just be the parent of the first argument. The purpose of the substitution index is to provide substitutions to be made during definition-copying. As well, the substitution index is modified as a side-effect of *definition-copy* so that it maps each node in the source to the node in the definition-copy that it gives rise to. These substitutions can then be used in another call to defining-copy so that node sharing is achieved.

The function *definitionCopyAt* is somewhat simpler in that it takes just a node and a location in terms of whose scope the node is to be expressed. The resulting node will not contain denoters. The function *definitionCopyAt* could be used to make explicit the definition of every empty node in *aNode* as follows:

```
for x in aNode.descendants andif node.isa.empty(x) do
  {x replace {x.definition definitionCopyAt x}}
 end
```

[1]Two types are equivalent if each is a subtype of the other; two values are equivalent according to the rules in A.20.4 relating quantifiers.

The *validate* function is provided to check for context-dependent correctness. Given a node, it yields an index that maps each erroneous node to a list of error message strings. If the yielded index is empty, the node is correct.

Many additional semantic facilities can be included in *node*. For example, the expression *node.global*("x") determines whether an identifier with the spelling "x" is globally visible; it yields the same result as

```
node.make.identifier("x").definingOccurrence
```

We have now seen enough metaprogramming facilities to write useful metaprograms.

5.6 Metaprogramming applications

5.6.1 Generating a meta-interface

When manipulating a program expressed in terms of a user-defined abstraction, it is necessary to recognize and construct the notations introduced by that abstraction. In this section, we shall see how to generate a *meta-interface*, i.e., a metaprogramming system interface to an abstraction's interface.

For instance, consider the following simple abstraction

```
simple:
  Tuple
    Type :: Any
    value : Type
    operation : Function (x : Type) Type end
   end
```

To manipulate a program that uses this abstraction, it must be possible to recognize the particular nodes where the abstraction is used. For example, applied-occurrences of *simple* must be identifiable. Also, because *simple* is a tuple, it must be possible to recognize applied-instances of the selections *simple.Type*, *simple.value*, and *simple.operation*. And because *simple.operation* is a function, it is frequently necessary to build a node representing a call to it.

To address these considerations, the following meta-interface can be generated:

```
let meta be
  tuple
    let simple be
      tuple
        let ! be node.global("simple")
        let a_Type be
          tuple
            let ! be
              node.make.selection(!'[1], node.make.identifier("Type"))
           end
        let value be
          tuple
            let ! be
              node.make.selection(!'[1], node.make.identifier("value"))
           end
        let operation be
          tuple
            let ! be
              node.make.selection(!'[1], node.make.identifier("operation"))
            let makeCall be
              function (x: Node)
                node.make.call
                  ({copy !}, node.construct(NodeClass.ArgumentList)([x]))
               end
           end
       end
   end
```

It makes available the following notations. The selection *meta.simple.!* yields the defining-occurrence of *simple*, which can be used to test for applied-occurrences of *simple*. The selections *meta.simple.a_Type.!*, *meta.simple.value.!*, and *meta.simple.operation.!* respectively yield the selections *simple.Type*, *simple.value*, and *simple.operation*, which can be used to test for applied-instances of each selection, e.g., using {*x* == *meta.simple.value.!*}. And the selection *meta.simple.operation.makeCall* yields a function, which given a node, creates a node representing a call to *simple.operation* with the given node as the argument.

In the discussion that follows, we shall see how to implement the Acer function, *makeMetaInterface*, which generates a meta-interface using Acer's host-language metaprogramming system. Additional examples of actual meta-interfaces are also presented.

The function *makeMetaInterface* is defined as:

```
let makeMetaInterface be
  function (x: Node)
    extendMetaInterface
      (x, makeTopMetaInterface (x.definedIdentifier))
  end
```

It requires a node argument x, which must be a top-level type-binding or fixed-value-declaration; and yields the meta-interface created by *makeTopMetaInterface* and modified by *extendMetaInterface*.

The basic form of a meta-interface, for a particular globally visible defining-occurrence *id*, is indicated in the definition of *makeTopMetaInterface*:

```
let makeTopMetaInterface be
  function (id: Node)
    node.substitute
      (node.parse.strings ()
        ([" let ? be                                   "
          "   tuple let ! be node.global(?) end "]))
      ([makeMetaInterfaceName (id)
        node.make.stringLiteral (id.spelling)])
  end
```

Two metaprogramming features must be explained to understand this function. First, the call *node.parse.strings* () yields an accumulator that parses strings, which in the above is used to express a node template. And second, the template is passed to *node.substitute*, which yields an accumulator that replaces each successive ? (or _) in the template with each successive node it consumes. Thus, in the above template, the first ? is replaced by the identifier generated by *makeMetaInterfaceName*, and the second ? is replaced by the string-literal representing the spelling of *id*.

The function *makeMetaInterfaceName* is defined as:

```
let makeMetaInterfaceName be
  function (id: Node)
    inspect id.class then
      when valueIdentifier then {copy id}
      when typeIdentifier then
        node.make.valueIdentifier ({"a_" + id.spelling})
      when empty then
        node.make.valueIdentifier
          ({"empty" + convert.integerToString (id.parent.position)})
    end
  end
```

For a value-identifier, it yields a copy of *id*; for a type-identifier, it yields a value-identifier that has the spelling of *id* but with *a_* appended to the front. (And for an empty defined-identifier, it yields a value-identifier that has the spelling *empty* but with the position of *id*'s parent appended as a string.)

As concrete examples, *makeTopMetaInterface* applied to the module *token* yields:

```
let token be
  tuple let ! be node.global("token") end
```

Applied to the type *TokenClass* it yields:

```
let a_TokenClass be
  tuple let ! be node.global("TokenClass") end
```

These can be collected into a tuple called *meta*:

```
let meta be
  tuple
    let token be
      tuple let ! be node.global("token") end
    let a_TokenClass be
      tuple let ! be node.global("TokenClass") end
   end
```

which can then be compiled. As a consequence, *meta.token.*! refers to the defining-occurrence of the module *token*, and *meta.a_TokenClass.*! refers to the defining-occurrence of the type *TokenClass*.

Of course, *meta.id.*! has little advantage over *node.global*("id") but the advantage of the meta-interface is more apparent when we consider how it is extended by the function *extendMetaInterface*, which is defined as:

```
let extendMetaInterface be
  function (x: Node; theInterface: Node)
    begin
      inspect x.class then
        when typeBinding, typeDeclaration then
          inspect x.kind.denotation.class then
            when optionType, enumerationType then
              extendForOptionOrEnumeration
                (x.kind.denotation, theInterface)
            when operatorType then
              extendForOperatorOrFunction
                (x.kind.denotation, theInterface)
          end
        when fixedValueDeclaration, variableValueDeclaration then
          inspect x.kind.denotation.class then
            when tupleType, recordType, dynamicType then
              extendForAggregate(x.kind.denotation, theInterface)
            when functionType then
              extendForOperatorOrFunction
                (x.kind.dentotation, theInterface)
          end
      end;
      theInterface
    end
  end
```

The extension depends on the class of *x* and the class of the denotation of the kind of *x*, which must be a concrete-type or type-operator. We shall consider each case in turn in the sections that follow. Note that the function *extendMetaInterface* is called recursively during extension for aggregates and so is generalized to handle all classes of declaration.

5.6.1.1 Extension for options or enumerations

An option-type or enumeration-type introduces a number of literals. To provide access to these, the basic meta-interface is extended by:

```
let extendForOptionOrEnumeration be
  function (theType: Node; theInterface: Node)
    for id in theType.children do
      insert.(theInterface.definition, -1,
              makeSubInterface(id))
    end
  end
```

which, for each identifier *id* in *theType*, uses *makeSubInterface* to generate an additional component according to the template in:

```
let makeSubInterface be
  function (id: Node)
    node.substitute
      (node.parse.strings ()
         ([" let ? be                            "
           "   tuple                             "
           "     let ! be                        "
           "       node.make.selection           "
           "         ({copy !'[1]},              "
           "          node.make.identifier(?)) "
           "     end                             "]))
       ([makeMetaInterfaceName(id)
         node.make.stringLiteral(id.spelling)])
    end
```

As a concrete example, for the identifier *an_arbitrary* in *TokenClass*, the function *makeSubInterface* yields:

```
let an_arbitrary be
  tuple
    let ! be node.make.selection
                ({copy !'[1]},
                 node.make.identititifier("an_arbitrary"))
   end
```

Thus, ! holds a literal-selection with a base determined by the ! of its containing meta-interface. Hence, *meta.a_NodeClass.an_arbitrary.*! refers to the node representing *NodeClass.an_arbitrary*.

5.6.1.2 Extension for operators and functions

A type-operator or function is frequently used in an operator-call or a function-call, so the basic meta-interface can be extended with a function that creates such a call:

```
let extendForOperatorOrFunction be
  function (theType: Node; theInterface: Node)
    insert.(theInterface.definition, -1,
            makeOperatorFunctionMetaInterface(theType));
   end
```

This function inserts into the meta-interface the template constructed by

```
let makeOperatorFunctionMetaInterface be
  function (theType: Node)
    modifyFunctionOperatorInterface
      (node.parse.strings ()
        ([" let makeCall be                          "
          "   function ()                            "
          "      node.make.call                      "
          "        ({copy !},                        "
          "         node.construct                   "
          "           (NodeClass.argumentList)([])))  "
          "    end                                   "])),
       theType)
   end
```

The template is modified by

```
let modifyFunctionOperatorInterface be
  function (theInterface: Node; theType: Node)
    {let theSignature be theInterface.definition.signature;
     let theAccumulationList be
       theInterface.definition.body.arguments[2].accumulationList;
     begin
       for decl in theType.signature.children do
         begin
           insert.(theSignature, -1,
                   node.make.fixedValueDeclaration
                     (makeInterfaceName(decl.definedIdentifier),
                      node.make.identifier("Node")));
           insert.(theAccumulationList, -1,
                   makeMetaInterfaceName(decl.definedIdentifier))
          end
        end;
       theInterface
     end}
  end
```

which inserts declarations into the signature and arguments into the accumulation-list.

As a concrete example, when applied to the type *Array*, the following function is included along with `!`:

```
let makeCall be
  function (a_BaseType: Node)
    node.make.call
      ({copy !},
       node.construct (NodeClass.argumentList) ([a_BaseType]))
  end
```

Hence, the call

```
meta.a_Array.makeCall(node.make.identifier("Integer"))
```

creates the node representing *Array*(*Integer*).

5.6.1.3 Extension for aggregates

An aggregate is frequently used in a type- or value-selection, so the basic meta-interface can be extended, for each declaration in the aggregate, by a component that holds such a selection:

```
let extendForAggregate be
  function (theType: Node; theInterface: Node)
    for decl in  theType.children
        andif {not node.isa.empty(decl.definedIdentifier)} do
      insert.(theInterface.definition, -1,
              extendMetaInterface
                (decl, makeSubInterface(decl.definedIdentifier)))
     end
   end
```

This function uses *makeSubInterface* as described earlier, and recursively extends each resulting sub-interface using *extendMetaInterface* also described earlier.

As a result, when a meta-interface has been generated for the module *node*, the selection *meta.node.parse.string.*! yields the node representing *node.parse.string*, and the call

```
meta.node.parse.string.makeCall
  (node.make.stringLiteral("let x be 10"))
```

constructs the function-call *node.parse.string*("let x be 10").

5.6.1.4 Summary of the meta-interface

To summarize, a meta-interface provides access to the notations made available by global identifiers. A global identifier *v* is available as *meta.v.*!; a global identifier T is available as *meta.a_T.*!. Selections of the form *v.v*, *v.T*, and *T.v*, which are based on global identifiers, are available as *meta.v.v.*!, *meta.v.a_T.*!, and *meta.a_T.v.*!, respectively. In the above selections, *v* may itself be a value-selection making all possible selections available.

In addition, if any *meta.*α.! is of kind type-operator or function-type, the selection *meta.*α*.makeCall* is a function that, given the appropriate number of node arguments, yields a node representing a call to the type-operator or function.

From this example we can see just how easily a metaprogramming system can be used to automate the generation of programs. In the next section, we shall see how semantics-preserving transformations can be easily implemented.

5.6.2 Program transformation

The examples considered in this section deal with manipulations that preserve semantics. To begin with, consider how to optimize the use of the metaprogramming system's function *node.parse.string*, which is frequently used to express node literals. This will serve to demonstrate the general notion of static evaluation.

5.6.2.1 Optimizing data abstraction

The idea behind the optimizing transformation presented in this section is to replace each call to *node.parse.string* that is applied to a string-literal argument, with the appropriate calls to *make* and *construct*. After all, if the argument to the parser is statically known, it can be statically parsed and the resulting node can be explicitly built using *make* and *construct* calls. For example, the call

```
node.parse.string ("let x be tuple 10, 20 end")
```

constructs the node

```
let x be tuple 10, 20 end
```

But so does

```
node.make.fixedValueBinding
   (node.make.valueIdentifier ("x"),
    node.make.empty (),
    node.construct (NodeClass.tupleLiteral)
       ([node.make.integerLiteral("10"),
         node.make.integerLiteral("20")]))
```

Clearly, the first call is more readable while the second is more efficient. To reconcile the two notations, the transformation that follows facilitates the readability advantage of the first approach while achieving the efficiency advantage of the second.

First, let us consider the implementation of a function that takes a node argument and yields the metaprogramming system calls that construct the node argument. The function *literalize* does exactly this:

```
let literalize be
  function (x : Node)
    if {x.class member grammar.list} then
      node.make.accumulation
        (meta.node.construct.makeCall
           (node.make.selection
              ({copy meta.a_NodeClass.!},
               node.make.identifier (grammar.spelling (x.class)))),
         for theChild in x.children do
           node.construct (NodeClass.accumulationList)
           literalize (theChild)
          end)
    else
     node.make.call
       (node.make.selection
          ({copy meta.node.make.!},
           node.make.identifier (grammar.spelling (x.class))),
        if {x.class member grammar.lexeme} then
          node.construct (NodeClass.argumentList)
            ([node.make.stringLiteral (x.spelling)])
         else
          for theChild in x.children do
            node.construct (NodeClass.argumentList)
            literalize (theChild)
           end
         end)
    end
  end
```

If x is a list, it creates an accumulation that applies the accumulator created by the call *node.construct* to the literalized form of each of the list's children. Otherwise, it creates a call to the appropriate *node.make* with either the string-literal spelling of the lexeme x or the literalized children of the construction x. The function, given the fixed-value-binding illustrated at the beginning of this section, generates precisely the expression, also illustrated at the beginning of this section, that constructs the fixed-value-binding.

Now, given *theNode* in which to perform optimization, the follow expression does the substitution:

```
for theChild in theNode.descendants
    andif {all true}
      ([node.isa.functionCall(theChild)
        {meta.node.parse.string.! == theChild.function}
        node.isa.stringLiteral(theChild.arguments[1])]) do
   {theChild replace
    literalize(node.parse.string (theChild.arguments[1].spelling))}
  end
```

The expression {*all true*} yields an accumulator that acts as a short-circuit Boolean 'and.' Hence, the above iterates through all descendants of *theNode* that satisfy the condition following the **andif**, which checks if *theChild* is a function-call, if the function of the call is equivalent to *node.parse.string*, and if the one argument to that call is a string-literal. If these hold, the body is applied—it replaces *theChild* with the literalized form of the node resulting from the application of *node.parse.string* to the spelling of the string-literal argument.

That such a useful transformation can be so easily implemented certainly lends credence to the claim that Acer is easy to manipulate. Furthermore, the fact that data abstraction is easily optimized also supports the design of more powerful, yet efficient, abstractions. For example, even the use of the *node.parse.strings*() accumulator in conjunction with the *node.substitute* function to express node templates can be easily optimized. The use of templates for generating meta-interfaces was illustrated in the previous section. Such templates can be optimized by the following:

```
for theChild in theNode.descendants
    andif {all true}
      ([node.isa.accumulation (theChild)
        node.isa.functionCall (theChild.accumulator)
        {meta.node.substitute.! == theChild.accumulator.function}
        node.isa.accumulation (theChild.accumulator.arguments[1])
        node.isa.functionCall
          (theChild.accumulator.arguments[1].accumulator)
        {meta.node.parse.strings.! ==
         theChild.accumulator.arguments[1].accumulator.function}
        for s in theChild.accumulator.arguments[1]
                  .accumulationList.children do
          {all true}
          node.isa.stringLiteral (s)
         end]) do
   {theChild replace
    literalizeAndSubstitute
      (theChild.accumulationList.children,
       for s in theChild.accumulator.arguments[1]
                  .accumulationList.children do
         node.parse.strings () s.spelling
        end)}
  end
```

The above iterates through the descendants of *theNode* that satisfy the condition, which tests if *theChild* is an accumulation, if the accumulator is a call to *node.substitute*, if the one argument to that call is an accumulation, if that accumulation's accumulator is a call to *node.parse.strings*, and if every element of that accumulation is a string-literal. If this holds, *theChild* is replaced by the result of *literalizeAndSubstitute* applied to the iterator that maps through the substitutions and the node resulting from the application of *node.parse.string* () to the spellings of all the string-literals.

The function *literalizeAndSubstitute* can be implemented by modifying *literalize* as follows:

```
let literalizeAndSubstitute be
  function (theSubstitutions: Iterator (Node); x: Node)
    if {node.isa.identifier (x) andif
        {{x.spelling = "?"} orif {x.spelling = "_"}}} then
      theSubstitutions.produce ()
    elsif ...
    else ...
    end
  end
```

An additional iterator argument is included and the initial test is for the substitution case. The rest of the function is implemented as before, except for the passing of the additional iterator argument to the recursive calls.

Applying the above transformation to the expression

```
node.substitute
  (node.parse.strings ()
    ([" let ? be "
      "   tuple  "
      "     10,  "
      "     20   "
      "    end   "]))
   ([aNode])
```

results in

```
node.make.fixedValueBinding
  (aNode,
   node.make.empty (),
   node.construct (NodeClass.tupleLiteral)
     ([node.make.integerLiteral ("10"),
       node.make.integerLiteral ("20")]))
```

In general, the above transformation approach can be used to support any data abstraction that expresses literals in textual form. The textual form can be statically analyzed and converted to a more efficient form. Such is the expressive power that support for manipulation affords. Furthermore, similar techniques can be used to support all manner of static evaluation. For example, the following

```
for theChild in theNode.descendants
   andif {all true}
     ([node.isa.functionCall (theChild.denotation)
       {meta.integer.+.! == theChild.denotation.function}
       node.isa.integerLiteral (theChild.denotation.arguments[1])
       node.isa.integerLiteral (theChild.denotation.arguments[2])]) do
  {theChild replace
   node.make.integerLiteral
     ({convert.stringToInteger (theChild.denotation.arguments[1].spelling) +
       convert.stringToInteger (theChild.denotation.arguments[2].spelling)})}
 end
```

statically evaluates *integer.+* applied to integer-literal arguments. This kind of static evaluation is applicable in a wide variety of situations.

5.6.3 Metaprograms as a command language

In a programming environment, such as the PCAcer environment described in Appendix B, metaprograms can be used as interactive high-level editing commands. For example, suppose a programmer wants to write an exhaustive variant-inspection for some particular selector and that there are a large number of variants. Instead of manually writing the inspection, he could use the following function to generate it:

```
let makeInspection be
  function (theSelector: Node)
   {let theKind be theSelector.kind.denotation;
    inspect theKind.class then
      when enumerationType, optionType then
        node.make.variantInspection
          (theSelector,
           for id in theKind.children do
             node.construct (NodeClass.valueWhenBranchList)
             node.make.valueWhenBranch
               (node.construct (NodeClass.whenCondition) ([id]),
                node.make.empty (),
                node.make.identifier (?))
            end,
           node.make.empty ())
      else theUnattachedEmptyNode
      end
    end
```

After entering this function, a PCAcer command could be invoked so that the function is compiled and applied to the selected node of the window in which the command is invoked, presumably the selector for which an inspection is to be generated. After the invocation, the result could be made available in a newly create window.

Applied to the expression *x.class*, where *x* is of type *Node*, the above would yield the variant-inspection:

```
inspect x.class then
  when accumulation then ?
  when accumulationList then ?
  . . .
  when whenCondition then ?
 end
```

In this way, high-level commands are constructed on the fly.

Now, suppose a programmer wishes to find all the applied-occurrences of some particular identifier in a given node. He could write the following function:

```
let findAppliedOccurrences be
  function (id: Node; theTarget: Node)
    for x in theTarget.descendants
        andif {node.isa.identifier(x) andif {x.definingOccurrence == id}} do
      node.construct (NodeClass.arbitraryList)
      node.make.denoter (x)
     end
   end
```

which, given a defining-occurrence *id* and *theTarget* in which to search, constructs an arbitrary-list of denoters to the applied-occurrences of *id*. It could be invoked by a PCAcer command that applies the function to the selected node of the selected window and to the selected node of the window in which the command is invoked. The result could then be made available in a newly created window. From this, the programmer could see the entire set of applied-occurrences and by following the definition link of each denoter, he could access each applied-occurrence in turn.

Certainly some of these kinds of commands will be a standard part of an environment like PCAcer. The point is, however, that it is not the responsibility of environment designer to provide an exhaustive set of useful commands. Instead, commands can be created on the fly to suit the needs of individual programmers and the applications with which they work. Such flexibility is not afforded by more conventional environments.

Only the imagination limits the commands that are possible. For example, the following could be used just as the previous function, but to determine all expressions of a particular type:

```
let findTypedValues be
  function (theType: Node; theTarget: Node)
    for x in theTarget.descendants
        andif {node.isa.value(x) andif {x.type == theType}} do
      node.construct (NodeClass.arbitraryList)
      node.make.denoter (x)
     end
   end
```

For instance, it could be used to find all expressions of type *Node*. The possibilities afforded by a metaprogramming system are unlimited.

5.7 Summary

Looking at the implementation of PCAcer described in Appendix C, it may be surprising that, with a metaprogramming system, implementing a transformation as complex as that of a compiler is relatively trivial. After all, a compiler is typically a major undertaking, but with so many general-purpose features being implemented as part of the metaprogramming system, a compiler is very simple to implement indeed. For instance, the implementation of the compiler used in PCAcer accounts for roughly 8% of the total implementation effort, in terms of the number of lines of code, and even less when considered in terms of the amount of time spent. Surely this demonstrates the efficacy of the metaprogramming approach.

Emphasis on metaprogramming systems, then, encourages the provision of general-purpose implementation tools that can be used for much more than just translating programs to object code. With so much effort going into the implementation of supporting environments, there should be more to show for that effort than just an end-product environment.

Chapter 6

Evaluation

6.1 A principled approach

It is well recognized that language design should proceed according to guiding principles. The principles presented in Chapter 4 are intended to extend a language designer's repertoire of such guiding principles. Similar guidance has been offered in the past, either in the form of pragmatic advice, such as Wirth's advice in [Wir87] and Hoare's advice in [Hoa87], or as principles rooted in the semantician's meta-principle of orthogonal language features. For example, Harland [Har84], and also Tennent [Ten81], present the following:

- Principle of Procedural Abstraction: *Any syntactic clause can be abstracted over, so it may be repeatedly invoked.*
- Principle of Completeness: *All data types must be first-class citizens, that is, they can be passed as parameters, assigned, stored as data-structure components, and returned from functions.*
- Principle of Declaration Correspondence: *If a data type can be declared as a parameter, it can be declared in-line and vice versa.*

Certainly Acer conforms to these principles, providing even first-class modules. As well, the principles of Chapter 4 are a manipulation-oriented extension of these ideas.

Harland used the above principles to guide the design of Poly [Har84], and has made many of the same decisions as made in Acer. However, Acer does not include types in the value-space, as does Poly, instead treating types as static objects. I would argue that Poly's heavy use of dynamic typing is too powerful, leading to programs with weak static properties.

Harland's decision to go the way of dynamic typing stems partly from a perceived conflict [Har84, page 115] between supporting both polymorphism and supporting static typing. But Acer contradicts this perception, supporting polymorphism and static typing. As well, the work in [ACPP91] shows how dynamic typing can be supported in the framework of an otherwise statically typed language.

Another good source of principles for guiding language design is given in [Mac87], which presents 16 principles and evaluates various well-known programming languages in terms of their conformance to these principles. The principles are very general and are set in a historical context; many are restatements of advice offered by Parnas, Dijkstra, Hoare, and others, and are now well-entrenched in the minds of language designers. In contrast, the principles of Chapter 4 are more narrow in their focus, dealing primarily with manipulation issues, and can be considered as corollaries of the general principles.

6.2 Transformational programming

It is important to emphasize that stressing support for manipulation is not the same as advocating transformational programming, as in the work of Partsch [Par90]. Manipulation is a more general term: transformation involves pure semantics-preserving manipulation guided by a logical calculus [PS83,Pep84], whereas manipulation involves all activities in which programs are the target of interest.

Furthermore, Acer does not even address the problem of program specification, although tuple-types do support the specification of abstract data-types and could be extended with logical constructs, such as the pre- and post-conditions in the style used by Meyer for his object-oriented language Eiffel [Mey88]. The issue of program specification is avoided because significant technical problems must be overcome to make transformational programming a generally applicable methodology. Automated support for proofs, in particular, are an open problem.

This is not intended to demean the transformational approach, or the work on program specification, but is intended to highlight the different perspective taken in this thesis. Transformations are seen as just another vehicle in the quest to provide better support for program synthesis, analysis, and maintenance. This is different from advocating the use of transformations and specifications as a mathematically correct way of deriving an efficient algorithm from a specification, or of verifying a logically complex notion of correctness.

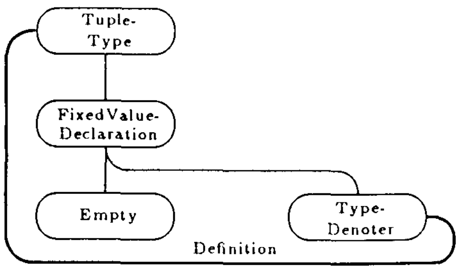
Nevertheless, the approach to language design advocated in this thesis is supportive of the transformational approach to programming. It provides a flexible way in which to express and implement transformations and a logical framework for the inclusion of specifications.

6.3 The style of semantic definition

In this thesis, a relational-style semantic definition, which maps programs to nodes and node relations, was used to define Acer. This is quite different from using a denotational-style semantic definition, which maps programs to mathematical values (and functions). Since meaning must be represented in some particular way a choice must be made as to which approach to use. Certainly a denotational-style definition is an appropriate choice. However, since nodes are sufficient and natural for rigorously defining the structure of abstract syntax there is no pressing need to provide both a denotation-style definition and a relational-style definition. Moreover, I hold the view that node structure and its relations are the meaning of a program.

Nodes, in fact, are the ideal conveyors of meaning. After all, nodes are the high-level objects manipulated in a programming environment. And an abstract data-type to represent nodes can be adequately supported in most modern programming languages.

To map the meaning of Acer programs to pure mathematical values would be cumbersome. For instance, nodes easily represent such things as self-referencing types and data-structures, e.g.,



To describe such structures as pure mathematical values would certainly be less intuitive. And if we are to treat programs as objects in their own right, objects that can be manipulated, a node representation is more pragmatic.

An attribute-style semantic definition, which maps programs to data structures [Knu68], could be used to specify formally the semantic relations of Acer programs by specifying nodes and relations instead of arbitrary data structures [HT85,HT86]. However, it would be just as suitable to specify formally Acer's semantics using NURN relations as described in [Dyc90]. This would ensure conformance to the principle that all semantic entities should be represented as nodes and node relations, not as arbitrary data structures. However, an attribute-style definition is not inconsistent with the approach advocated in this thesis.

It is important to realize that this thesis has concentrated on defining Acer's semantics by specifying Acer's supporting metaprogramming system. Thus, it is not so much concerned with a particular style of formal definition [Pag81], as long as the behavior of the supporting metaprogramming system is adequately specified.

6.4 On programming environments

The wealth of current research in programming environments has served as a motivating stimulus for this thesis. Commonalities in the various approaches in such environments as Mentor [DGKLM84], PSG [BS86], CPS [TR81], and SbyS [Min90] directly support the view that nodes must be the focal point; they are invariably the objects that are manipulated in such environments. Minor echoes this view [Min90, page 47] when he says, "It would be of great value if a canonical form for abstract grammar could be agreed upon, and the structure of programming languages were defined in this notation." It is hoped that the GRAMPS-style node representation described in [CI84] and in this thesis will further the goal of providing a standard abstract representation.

One danger of providing such a standard abstract representation is that there is always a temptation to make the representation increasingly complex to deal the multitude of idiosyncratic features of various languages. This thesis has demonstrated that a simple representation of nodes is adequate for handling the job of representing context-free and context-dependent syntax. So the lesson is that languages should be defined according to a *simple* abstract node representation.

The PCAcer language-based environment has not been described in detail in the body of this thesis, but its design and implementation has influenced the design of Acer. After all, a language design cannot be taken seriously unless it is fully implemented, for only by doing so are various deficiencies revealed. The design of the PCAcer environment draws from the approaches in the literature. In particular, PCAcer attempts to provide a compromise, much like that advocated in [BS86], by supporting both a concrete textual view and an abstract node view. Hence, low-level manipulation that is conveniently expressed in terms of text can be accomplished in that way, and high-level manipulation best expressed in terms of nodes is expressed in terms of nodes. As well, the two types of view are easily interchanged, sharing a common core of editing commands, e.g., select, delete, insert, copy, and so on.

Realize, however, that PCAcer has not been designed so much to be innovative, but to demonstrate how easily Acer can be supported. For instance, requiring all semantic objects to be represented as nodes ensures a syntactic representation for precisely the values that Acer programs manipulate. Hence, values, like programs, are viewed as nodes, and support for their manipulation is unified under the guise of these nodes. This is what allows the result of evaluating a program in PCAcer to be viewed as a node.

Other properties of Acer also help to support PCAcer's implementation. For example, the property that every node gives rise to at least one token in the concrete view facilitates the direct selection of nodes by pointing at their tokens. As well, the property that every type, definition, and defining-occurrence is represented as a node allows PCAcer to generate nodes to respond to semantic queries. It is my view that the consequences of a particular language design only become apparent when a supporting environment is implemented—even the implementation of just a compiler is insufficient in this regard.

The notion of using metaprograms written in Acer as a command language for an Acer environment (see 5.6.3) is an innovative approach, which facilitates unlimited extensibility. Unfortunately, the memory limitations of PCAcer makes implementing an Acer host-language metaprogramming system for Acer difficult. Therefore, the realization of this approach is left as future work.

6.5 On Acer itself

Acer simplifies Quest's syntax and its domain of semantic objects. For instance, kinds are eliminated from the syntax. Also, Acer extends Quest with a number of innovative constructs, such as the various forms of method-call. These constructs strengthen the argument in [Car89] that Quest unifies the notions of polymorphism and "inheritance." Indeed, Acer's method-calls provide an object-oriented style of method invocation, and defining value-selection and indexing in terms of method-calls facilitates great expressiveness.

The implementation of Acer's form of "inheritance" is interesting in that each object need not be tagged at run-time to indicate the applicable methods. The abstract-type of each object is statically known, which determines the applicable methods. Thus Acer extends Quest by providing a form of method-call, which is simply rewritten as a function-call in which the called function is selected as a component of the data-structure (tuple or record) determined by the static type of the target object. For example, if x has type *module.Type*, the method-call *method.*(x) is equivalent to *module.method*(x), assuming of course that *module* declares an appropriate function named *method*. Thus the called method is determined by the selection *module.method* and no run-time search of a method dictionary is required.

Furthermore, *module* could implement its abstract values so that each x contains potentially different function components. The call *module.method* (x) could then be implemented to invoke the particular function associated with the x given in the call. This implements the notion of *virtual* methods in which the called method is determined dynamically for each object. Clearly the style of "inheritance" supported by Acer and Quest is extremely flexible.

Another innovative feature of Acer are references, which can be used like pointers but are implemented as fetch/store functions. They support the high-level modeling of updatable locations. For instance, a reference could be used to model a location that does not even exist in physical memory; a fetch could be implemented as a read from the file-system, and a store could be implemented as a write to the file-system.

Iterators and accumulators are yet another feature of Acer not present in Quest. Their design is based on the description in [Cam89]. Acer's iterators and accumulators are implemented as tuples with appropriate components rather than as a new type of semantic object. As a result, they introduce minimal additional complexity while providing maximal utility—iterators and accumulators can even be recursively defined. Also, the accumulation-literal, which is an accumulator applied to literal arguments, has proved to be very useful for supporting data constructors that take an arbitrary number of arguments, e.g., a list could be constructed by *list.construct* (*BaseType*) ([x, y, z]).

Acer's abstraction mechanisms can be seen in a very favorable light. As Hilfinger [Hil83, page 3] puts it, "Proper design of the abstraction facilities of a language not only increases its utility to programmers, but can also simplify the language and reduce the language designer's temptation or need to provide all things to all programmers." Acer reinforces this view. For instance, Acer has a very small number of built-in types and most notations are applicable for abstract data-types in general; even basic types such *Integer* and *Real* and structured types such as *Array* are defined as abstractions. As a result, Acer's definition need not specify the properties and operations of a large number of built-in types. This certainly simplifies the language and at the same time supports extensibility via powerful abstraction mechanisms.

Folding is particularly well supported in Acer because functions can abstract over arbitrary value expressions and type-operators can abstract over arbitrary type expressions.

Even variables can be folded as reference- or pointer-yielding functions. Quest does not provide the same level of support for folding because it does not include pointers or references and because a function's body cannot refer to non-local (free) variables. In Quest, non-local variables must be contained by a tuple or some other data-structure to be accessible, but in Acer, variables are allocated on the heap so dangling stack references are of no concern. Moreover, an implementation of Acer could allocate particular variables directly on the stack, as Quest does for all variables, when a reference or pointer to the variable is not required and the variable is not referenced in a function body.

6.6 Future work on Acer

Implementing an Acer host-language metaprogramming system for Acer is the crucial next step in the development of Acer. Such an application would serve to demonstrate Acer's performance characteristics (i.e., execution efficiency and memory usage). It would also highlight the advantages and disadvantages of programming with a typeful programming language.

The reader might ask why such an implementation is not supported by PCAcer. The main reason is the memory limitation imposed by MS-DOS, which limits the size of programs that can be handled. Of course, various swapping techniques could be applied to make better use of the limited memory space, but this would greatly complicate PCAcer's already extensive implementation, which consists of 36,000 lines of Pascal code. As well, for portability, PCAcer translates Acer to an interpreted machine language, so execution speed would be expectedly slow. Porting the implementation of PCAcer to a more powerful architecture and translating to native machine code would facilitate further evaluation of Acer.

Another area for further work concerns the problems associated with persistent storage as supported by Acer's dynamics. The ability to store persistently abstractly typed values has its implications on data abstraction because it implies that any value can be recursively duplicated. This must be taken into account by the designer of an abstraction. For instance, although the metaprogramming system module *node* provides *theUnattachedEmptyNode* as the parent of all unattached nodes including itself, this node can be duplicated as yet another empty node that has itself as its parent. Therefore, to test whether a node *x* is attached or unattached, one should not test:

{*x.parent* is *node.theUnattachedEmptyNode*} one should test *node.isa.empty* (*x.parent*) because *theUnattachedEmptyNode* is not necessarily unique. Hence, the declaration for *theUnattachedEmptyNode* should not be provided to prevent the impression that it is a unique individual. But then, were it not for persistent storage, *theUnattachedEmptyNode* could very well be a unique individual.

The fact that persistent storage interacts with abstraction in such an unexpected way is somewhat forboding. Unless some care is taken, certain abstractions could be crashed e.g., copying a value representing an open file may not be meaningful. Note that Quest restricts persistent storage to values of concrete-types, but that stifles its usefulness. Moreover, since Acer defines even basic types such as *Integer* as abstract-types, dynamics restricted to concrete-types would be quite useless.

Another problem with persistent storage for values of abstract-types is that no test is performed to ensure that the abstract-type has not been recompiled in terms of a different implementation type. Clearly such recompilation would invalidate all persistently stored values based on that abstract-type.

But perhaps we should not be unduly concerned, after all, when a program can input a function-value and call it, it can never be completely sure that it is not loading a 'Trojan horse.' Thus, persistent storage has its implications, take it or leave it.

Further rumination concerning the particular features included in Acer is also in order. For instance, enumeration-types could be excluded in favor of option-types, and even tuple-types could be excluded in favor of record-types. This would simplify the language but could harm performance.

Another semantic issue that needs attention concerns the ability of a function-call to act as a constructor, see A.7.2. A constructor, such as a tuple-literal or function-literal, supports recursive definitions. For example,

let *x* **be tuple** *x* **end**

defines a recursive tuple. But, if we define a function *makeTuple* as

let *makeTuple* **be function** (*x* : *Type*) **tuple** *x* **end end**

we could not use it as

let *x* **be** *makeTuple*(*x*)

to define a recursive tuple, because x cannot be passed to a function until its definition is completely determined.

A solution to this problem would be to define a restricted type of function called a constructor-function. Thus Acer would define a constructor-literal and a constructor-type exactly analogous to a function-literal and a function-type. For example,

let *makeTuple* **be constructor** (x: *Type*) **tuple** x **end end**

which has type:

Constructor (x: *Type*) **Tuple** : *Type* **end end**

A constructor-literal would be restricted to have a body that denotes a constructor and to treat each of its arguments as a delayed-occurrence (see A.7.2), i.e., as a reference to a data-structure that is not yet completely defined. Acer's subtyping rules would be extended so that a constructor-type is considered to be a subtype of a function-type according to the regular function subtyping rules, but not vice versa. A constructor-function could then be used in a regular function-call, but in this case the arguments would be permitted to be delayed-occurrences. It would then be valid to write

let x **be** *makeTuple*(x)

Constructor-values could be implemented so that a run-time constructor-value contains an instance of the uninitialized data-structure that the constructor-literal creates. A delayed-occurrence referring to a call to a constructor-function would duplicate this data-structure, and when the constructor is finally called, this duplicate would be provided to the constructor-function, in a register perhaps, and the constructor-function would initialize it. Also, to support the notion that a constructor-type is a subtype of a function-type, when a constructor-function is used as a regular function, the uninitialized data-structure would not be provided as part of the call and the constructor-function would simply create the data-structure itself. The importance of providing user-defined constructors is not clear at this point in Acer's evolution.

Constructor-functions, as described above, are related to the notion of lazy-evaluation: both involve delaying the complete realization of a definition. However, lazy-evaluation is more powerful (as is indicated by the lazy functional language Miranda [Tur86]) because it supports the definition of potentially infinite data-structures (e.g., the list of all primes). Constructor-functions, on the other hand, can only support the definition of recursive data-structures. In Acer, lazy-evaluation must be explicit, i.e., in place of a particular value, a function that yields the value should be used instead. In Miranda, such mechanisms are implicit from context.

6.7 Beyond Acer

Clearly Acer is but a step on the quest towards more manipulable languages. When innovative language features and programming methodologies arise, they will have to be either incorporated into a new language or meshed with an existing language. Acer provides a conceptual framework for meshing with such innovations: new kinds of types and values, and new kinds of computational structures, can easily be included in the existing framework (e.g., just as the extension for constructor-functions is easily included.)

Beyond Acer, however, lies a vaste domain of unexplored territory, and the direction in which to go is uncertain. For instance, what style of programming is best, object-oriented programming, functional programming, or logic programming? And is it even reasonable to ask such questions? After all, various styles of programming have different advantages and disavantages.

The advice offered by this thesis, therefore, attempts to deal with fundamental manipulation concerns that should be addressed regardless of the particular style of programming language chosen. To this end, three general areas of language design that influence the nature of program manipulation were identified in Chapter 4, namely the mapping between concrete syntax and abstract syntax, the mapping between context-dependent syntax and abstract syntax, and the mapping between equivalent language constructs. Abstract syntax-tree nodes and their context-dependent relations, as described in Chapter 3, are the unifying concept in this realm.

Reiterating the summary of Chapter 4, to achieve a useful correspondence between semantic structure and syntactic structure the design of a language should begin with a classification of its entities, e.g., variables, types, values, expressions, statements, procedures, modules, etc. Next, the attributes associated with each class of entity should be identified. Some of these will be defined and others derived, e.g., a variable has a name and type that are defined and a storage location that is derived. For each different class of entity, a grammar production to represent its defined attributes should be included in the abstract syntax. Statically scoped nested block structure with modules should then be used to integrate the abstract syntax for the semantic entities into a framework in which all hiding is intentional rather than accidental. Finally, the abstract syntax should be carefully augmented with keywords and punctuation to obtain a concrete syntax with appropriate manipulation characteristics.

Also, to support transformation (i.e., semantics-preserving manipulation), the design of a language should, in general, support folding and unfolding; it should be possible to abstract over any expression and it should be possible to rewrite any invocation in terms of its definition. Meaningful manipulation can only be well-supported when such basic activities are well-supported.

And most importantly, to ensure that manipulation is in fact well-supported, a metaprogramming system for the language should be constructed. All other support tools can then be implemented in terms of the standard abstract view implemented by that metaprogramming system.

6.8 Final words

If this thesis achieves only the goal of prompting language designers to consider support for program manipulation in the design of their languages, it will have accomplished much. Perhaps the design of Acer, as a framework for a manipulable imperative language, will be seen in a positive light as well, but this is of lesser concern.

The fundamental advice to language designers is that they should specify and implement a supporting metaprogramming system based on some reasonably simple concept of nodes. If improved programming languages, methodologies, and environments are to meet the ever increasing demand for better software, language designers must recognize that support for program manipulation is a vital concern.

Appendix A

The Acer definition manual

A.1 Introduction

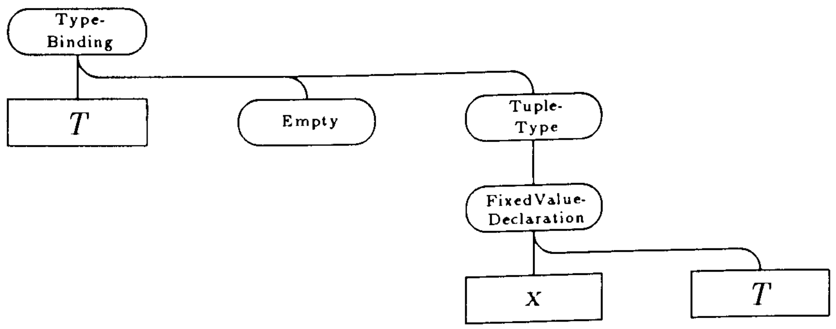
This appendix describes the syntax and semantics of Acer, an imperative, general-purpose, expression-oriented language with an elaborate type system that supports type quantification, polymorphism, and structural subtyping. Acer's design is based on that of the typeful programming language Quest but there are significant differences because Acer's design, as discussed in Chapter 4, is guided by manipulation principles. In particular, Acer's three-level type system is supported by just two levels of syntactic object, types and values, rather than by three levels of syntactic object, kinds, types, and values. Also, Acer provides constructs to support "object-oriented" programming as well as constructs to support high-level iteration and accumulation.

Acer's context-free syntax is specified in terms of GRAMPS-style nodes; its context-dependent syntax is specified in terms of NURN-style node relations. A formal context-free grammar is given but formal NURN relations are not. Instead, relations are described informally, as is dynamic semantics.

The abstract properties of nodes, and the concepts that underlie GRAMPS and NURN, are fully described in Chapter 3. Very briefly, a GRAMPS-style grammar has five classes of rule, namely lexical, construction, list, alternation, and character description. Lexical, construction, and list rules specify node classes, either lexeme classes, construction classes, or list classes; alternation rules specify sets of node classes (i.e., node categories); and character description rules specify regular expressions for use in lexical rules.

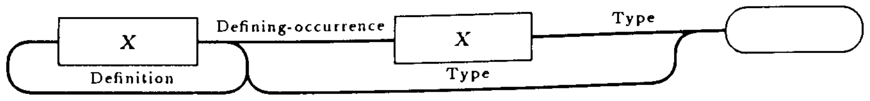
Acer is defined in terms of nodes and relations on nodes because nodes, as the syntactic representation of semantic objects, are the basis of all manipulation. When node structure is well-defined, a semantic object is synonymous with its representation, just as the number ten is synonymous with its representation as the numeral 10. Moreover, simple manipulation can then produce meaningful semantic effects, just as combining the digits of two numerals produces the sum of two numbers. Emphasis on nodes reflects the view that specifying how constructs are abstractly represented and manipulated is as important as specifying what they mean.

Nodes will often be illustrated using a graphical representation, e.g.,



An oval represents a construction or list node of the indicated class. A box represents a lexeme with the indicated spelling—the node class of a lexeme is implied by its spelling. A connection between the top of one node and the bottom of another represents an instance of the parent relation. Children are shown in left-to-right order to fully indicate the parent-position-child relation.

Context-dependent relations will also be illustrated graphically, e.g.,



An unlabeled oval represents an unspecified node. A labeled connection from the right-side of one node to the left-side of another represents a binary relation—a node's position in the relation is determined by the side that is connected. When a binary relation is interpreted to mean that a node has another node as its attribute, the connections to the right of a node lead to its attributes, e.g., the above indicates that an identifier x has as its defining-occurrence some other identifier x, that it has as its type an unspecified node that is also the type of its defining-occurrence, and that it is its own definition.

The remainder of this appendix is organized as follows. Section A.2 presents all of Acer's alternation rules, introducing the names of Acer's node categories, node classes, and semantic objects. Section A.3 outlines Acer's primary context-dependent relations, namely defining-occurrence, definition, denotation, type, kind, and subtype. Section A.4 presents Acer's lexical and character description rules, including Acer's technique of associating comments with nodes. And finally, sections A.5 through A.33 describe Acer's construction and list rules and discuss their associated relations and dynamic semantics.

A.2 The syntactic domains

An alternation rule defines and names a node category, that is, a set of node classes. Alternation rules refer to node classes by name so a brief overview of Acer's alternation rules serves to introduce much of the terminology used in describing Acer.

Acer's alternation rules give rise to a classification tree whose root is defined by Acer's most inclusive alternation rule

$$\langle Arbitrary \rangle ::= \langle Argument \rangle \parallel \langle Declaration \rangle \parallel \langle Miscellaneous \rangle$$

A discussion of the subcategories argument, declaration, and miscellaneous is presented in the subsections that follow.

Several additional alternation rules are defined, namely block, denoter, identifier, and reused-identifier:

$$\langle Block \rangle ::= \langle TypeBlock \rangle \parallel \langle ValueBlock \rangle$$

$$\langle Denoter \rangle ::= \langle TypeDenoter \rangle \parallel \langle ValueDenoter \rangle$$

$$\langle Identifier \rangle ::= \langle TypeIdentifier \rangle \parallel \langle ValueIdentifier \rangle$$

$$\langle ReusedIdentifier \rangle ::= \langle ReusedTypeIdentifier \rangle \parallel \langle ReusedValueIdentifier \rangle$$

These rules do not fall neatly into Acer's classification tree but since they are not used in any other grammar rule they can be ignored. They merely provide convenient terminology, that is, generic names for semantically analogous constructs.

The presence of arbitrary in Acer's grammar is significant because Acer's grammar completely avoids syntactic ambiguity. This implies that a parser for every node class and node category exists, and consequently that a parser for arbitrary exists. Therefore, Acer is phrase unambiguous, that is, the node structure of any textual phrase can be unambiguously determined.

A.2.1 Argument

Of the disjoint categories argument, declaration, and miscellaneous, argument includes the most of alternatives. It is defined as

$\langle Argument \rangle ::= \langle Binding \rangle \parallel \langle Expression \rangle$

Binding and expression are grouped into a single category called argument because an argument-list (see A.15) can contain both bindings and expressions.

In general, a binding consists of an identifier, a type, and an expression. Its effect is to bind the expression to the identifier so that the expression can be referred to by name. (The type indicated by a binding is optional and is used to restrict the type of expression that can be bound to the identifier; such restriction involves subtyping as described in section A.3.4.) The similarity between bindings and expressions stems from the fact that in a context permitting an argument, an expression is treated as an anonymous binding, that is, as a binding with no name.

Consider now the subcategories of binding and expression.

A.2.1.1 Binding

Binding is defined as

$\langle Binding \rangle ::= \langle TypeBinding \rangle \parallel \langle ValueBinding \rangle$

which mirrors the partitioning of Acer's domain of semantic objects into the disjoint categories type and value.

A type-binding comes in only one form but a value-binding comes in one of two forms:

$\langle ValueBinding \rangle ::= \langle FixedValueBinding \rangle \parallel \langle VariableValueBinding \rangle$

Two forms of value-binding are present because the identifier of a variable-value-binding can be rebound by assignment (see A.30.4) but the identifier of a fixed-value-binding cannot. Thus Acer makes a distinction between a *variable* value-identifier and a *fixed* value-identifier (see A.6).

A.2.1.2 Expression

Expression is the general term that encompasses both types and values. It is defined as

$\langle Expression \rangle ::= \langle Type \rangle \parallel \langle Value \rangle$

The majority of Acer's node classes are expressions.

Type is defined as

$\langle \mathit{Type} \rangle$ ::=
  $\langle \mathit{AbstractType} \rangle$ || $\langle \mathit{ConcreteType} \rangle$ || $\langle \mathit{ReusedTypeIdentifier} \rangle$ ||
  $\langle \mathit{TypeBlock} \rangle$ || $\langle \mathit{TypeDenoter} \rangle$ || $\langle \mathit{TypeDesignation} \rangle$ ||
  $\langle \mathit{TypeOperator} \rangle$

for which the alternatives abstract-type and concrete-type are defined as

$\langle \mathit{AbstractType} \rangle$ ::=
  $\langle \mathit{OperatorCall} \rangle$ || $\langle \mathit{TypeIdentifier} \rangle$ || $\langle \mathit{TypeSelection} \rangle$

$\langle \mathit{ConcreteType} \rangle$ ::=
  $\langle \mathit{AnyType} \rangle$ || $\langle \mathit{DynamicType} \rangle$ || $\langle \mathit{EnumerationType} \rangle$ ||
  $\langle \mathit{FunctionType} \rangle$ || $\langle \mathit{OptionType} \rangle$ || $\langle \mathit{RecordType} \rangle$ ||
  $\langle \mathit{TupleType} \rangle$ || $\langle \mathit{VariantType} \rangle$

Together these three rules introduce the node class names of all types.

Value is defined as

$\langle \mathit{Value} \rangle$ ::=
  $\langle \mathit{Accumulation} \rangle$ || $\langle \mathit{AndIfTest} \rangle$ || $\langle \mathit{Assignment} \rangle$ || $\langle \mathit{CodePatch} \rangle$ ||
  $\langle \mathit{CompoundValue} \rangle$ || $\langle \mathit{Conditional} \rangle$ || $\langle \mathit{Dereference} \rangle$ ||
  $\langle \mathit{DyadicMethodCall} \rangle$ || $\langle \mathit{DynamicInspection} \rangle$ || $\langle \mathit{FunctionCall} \rangle$ ||
  $\langle \mathit{Index} \rangle$ || $\langle \mathit{IsTest} \rangle$ || $\langle \mathit{IsNotTest} \rangle$ || $\langle \mathit{Iteration} \rangle$ || $\langle \mathit{KeepTrying} \rangle$ ||
  $\langle \mathit{Literal} \rangle$ || $\langle \mathit{OrdCall} \rangle$ || $\langle \mathit{OrIfTest} \rangle$ || $\langle \mathit{PointerCall} \rangle$ ||
  $\langle \mathit{PrefixMethodCall} \rangle$ || $\langle \mathit{Raise} \rangle$ || $\langle \mathit{Try} \rangle$ || $\langle \mathit{TryFinally} \rangle$ ||
  $\langle \mathit{UnaryMethodCall} \rangle$ || $\langle \mathit{ValCall} \rangle$ || $\langle \mathit{ValueBlock} \rangle$ || $\langle \mathit{ValueDenoter} \rangle$ ||
  $\langle \mathit{ValueSelection} \rangle$ || $\langle \mathit{VariantInspection} \rangle$

for which the alternative literal is defined as

$\langle \mathit{Literal} \rangle$ ::=
  $\langle \mathit{ArrayLiteral} \rangle$ || $\langle \mathit{CharacterLiteral} \rangle$ || $\langle \mathit{DynamicLiteral} \rangle$ ||
  $\langle \mathit{ExceptionLiteral} \rangle$ || $\langle \mathit{FunctionLiteral} \rangle$ || $\langle \mathit{IntegerLiteral} \rangle$ ||
  $\langle \mathit{LiteralSelection} \rangle$ || $\langle \mathit{RealLiteral} \rangle$ || $\langle \mathit{RecordLiteral} \rangle$ ||
  $\langle \mathit{ReferenceLiteral} \rangle$ || $\langle \mathit{ReusedValueIdentifier} \rangle$ || $\langle \mathit{StringLiteral} \rangle$ ||
  $\langle \mathit{TupleLiteral} \rangle$ || $\langle \mathit{ValueIdentifier} \rangle$ || $\langle \mathit{VariantLiteral} \rangle$ || $\langle \mathit{VoidLiteral} \rangle$

Together these two rules introduce the node class names of all values.

A.2.2 Declaration

The alternation rule for declaration, a category disjoint from argument and miscellaneous, closely mirrors that of binding:

$\langle \mathit{Declaration} \rangle$ ::= $\langle \mathit{TypeDeclaration} \rangle$ || $\langle \mathit{ValueDeclaration} \rangle$

Also, as with a binding, a type-declaration comes in only one form but a value-declaration comes in one of two forms:

> ⟨*ValueDeclaration*⟩ ::=
> ⟨*FixedValueDeclaration*⟩ ‖ ⟨*VariableValueDeclaration*⟩

In general, a declaration consists of an identifier and a type. It is similar to a binding in the sense that it introduces a named expression of some type. The difference is that the identifier of a declaration is bound to its expression dynamically, rather than statically. Hence a binding includes a static definition (i.e., an expression) but a declaration does not. Because their definitions are hidden, declarations support information hiding.

A.2.3 Miscellaneous

The final disjoint category of arbitrary is miscellaneous:

> ⟨*Miscellaneous*⟩ ::=
> ⟨*Empty*⟩ ‖ ⟨*AccumulationList*⟩ ‖ ⟨*ArgumentList*⟩ ‖ ⟨*ArrayList*⟩ ‖
> ⟨*BindingList*⟩ ‖ ⟨*ConditionalBranch*⟩ ‖ ⟨*ConditionalBranchList*⟩ ‖
> ⟨*IndexList*⟩ ‖ ⟨*IteratorElement*⟩ ‖ ⟨*IteratorList*⟩ ‖ ⟨*Signature*⟩ ‖
> ⟨*TypeBranchList*⟩ ‖ ⟨*TypeWhenBranch*⟩ ‖ ⟨*ValueBranchList*⟩ ‖
> ⟨*ValueWhenBranch*⟩ ‖ ⟨*VariantElement*⟩ ‖ ⟨*VariantList*⟩ ‖
> ⟨*WhenCondition*⟩

which introduces the names of Acer's remaining node classes. These node classes represent semantically incomplete constructs that are typically used as components of bindings, declarations, or expressions to represent complete constructs. Therefore, there is no semantic significance to grouping constructs in the category miscellaneous.

This completes the description of Acer's alternation rules and hence the overview of Acer's node categories and node classes. Before proceeding with detailed descriptions of Acer's various nodes, let us briefly consider general aspects of Acer's context-dependent relations.

A.3 Context-dependent relations

A fundamental principle guiding the design of Acer is that semantic objects should be represented as nodes and relations on nodes. Acer's nodes have just been introduced so consider now the relations defined on these nodes.

A.3.1 Scope

The primary means by which distant nodes in a program come to be related is through the *defining-occurrence* relation, which relates an identifier to the point in the program were it is defined. In context, an identifier can appear as either a defining-occurrence or as an *applied-occurrence*. An identifier is a defining-occurrence if it appears as the defined-identifier of a binding, declaration, iterator-element, type-when-branch, or value-when-branch, or if it appears as an element of an enumeration-type or option-type; otherwise, it is an applied-occurrence. The defining-occurrence of a defining-occurrence is of course that identifier itself but the defining-occurrence of an applied-occurrence is an identifier node determined according to scope rules, which we will now consider.

Acer's scope rules are defined in terms of two concepts: *name-layer* and *identifier-lookup*. A name-layer is a set of uniquely-spelled defining-occurrences and identifier-lookup is an algorithm for searching name-layers given a spelling and a starting node.

Name-layers are derived from nodes as follows. Each enumeration- and option-type has a name-layer containing its identifiers. Each argument-list, binding-list, dynamic-literal, record-literal, and tuple-literal has a name-layer containing the defined-identifiers of its bindings. Each dynamic-type, record-type, signature, and tuple-type has a name-layer containing the defined-identifiers of its declarations. Each iterator-list has a name-layer containing of the defined-identifiers of its iterator-elements. And each type-when-branch and value-when-branch has a name-layer containing its defined-identifier. In addition to these name-layers is *the global name-layer* (see A.9), which is associated with the unattached empty node and contains all globally visible identifiers.

Identifier-lookup, given a spelling and a start node x, proceeds as follows. At each step, the node-class of the parent of x and position of x with respect to its parent determines which name-layer(s) are searched and whether lookup should continue. For example, when x is a declaration in a signature, the name-layer of that signature is searched; if an appropriately spelled defining-occurrence is not found, identifier-lookup continues with the signature as x. For most node classes, no name-layers are searched and lookup continues with the parent of x as x. Thus lookup typically involves a rootward traversal of the tree with searching of name-layers only at particular nodes along the way.

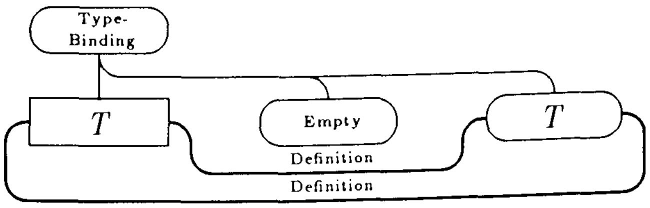
Name-layer searching is controlled by the following node classes: argument-list, binding-list, dyadic-method-call, dynamic-literal, dynamic-type, empty, function-call, function-literal, function-type, iteration, literal-selection, prefix-method-call, record-literal, record-type, reused-type-identifier, reused-value-identifier, signature, type-block, type-operator, type-selection, tuple-literal, tuple-type, unary-method-call, value-block, value-selection, variant-literal, and when-condition. For the sake of brevity, the details of how these node classes affect identifier-lookup are described as each node class is described in sections A.5 through A.33.

One notable aspect of Acer's scope rules is that scope is not affected by order of definition—a name-layer is after all a set—so forward reference to declarations and bindings is generally is acceptable, dependency analysis is used to determine invalid references (see A.7.2).

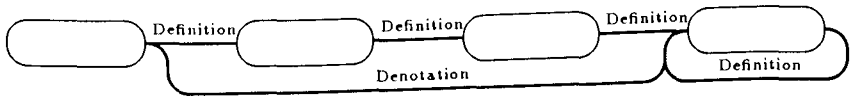
A.3.2 Definition and denotation

Because expressions are frequently used to denote other expressions (e.g., *pi* might be used to denote 3.14159), Acer provides the definition relation, which maps an expression node or empty node to the expression node or empty node that represents its definition. Most expressions, except for the following, are simply their own definition: accumulation, and-if-test, dyadic-method-call, empty, index, iteration, literal-selection, or-if-test, operator-call, prefix-method-call, reused-type-identifier, reused-value-identifier, type-block, type-denoter, type-designation, type-identifier, type-operator, unary-method-call, value-block, value-denoter, value-identifier, and value-selection. The details of the definition relation are described as each relevant node class is described.

Since the definition of an expression is an expression, which in turn may have as its definition yet another expression, there is the possibility of invalid circular definitions, e.g.,



However, in a correct program, repeated applications of the definition relation ultimately results in an expression that is its own definition.



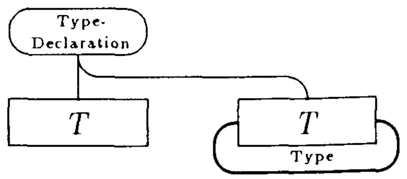
Thus Acer provides the denotation relation to map an expression node or empty node directly to this ultimate definition—this relation detects circular definitions and yields the error denotation in that case, e.g., either *error* or *Error* (see A.25), depending on whether the expression is a value or type.

Note that bindings have a definition by virtue of having a component by that name (see A.7).

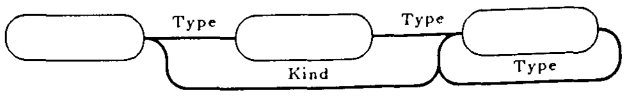
A.3.3 Type and kind

Because expressions are typed (e.g., 3.14159 has type *Real*), Acer provides the type relation, which maps an expression node or empty node to the type node or empty node that represents its type. The notion that values have types is quite conventional but the notion that types have types is not. It stems from the fact that Acer supports a three-level type system in which types are classified according to *kind*. The type of a value may be any class of type but the type of a type must be a restricted form of type called a *kind*. A type is a kind if it denotes either a concrete-type or a concrete type-operator (see A.19), that is, a type-operator with a body that is a kind. A kind is its own type.

Since the type of an expression is a type, which in turn may have as its type yet another type, there is the possibility of invalid circular types, e.g.,



However, in a correct Acer program two applications of the type relation must yield a kind:



Thus Acer provides the kind relation to map an expression node or empty node directly to this ultimate type—this relation detects when the type of a type is not a kind and yields the error type in that case, e.g., *Error* (see A.25).

Note that every binding and declaration also has a type by virtue of having a component by that name (see A.8).

A.3.4 Subtype

Acer defines a subtype relation on types to induce a lattice, that is, a subtype hierarchy. A type T is a subtype of a type T' if the denotation of T is a subtype of the denotation of T', or when T is not a kind, if the type of T is a subtype of T'. Subtype is defined in terms denotations and since the denotation of a type is either a type-operator, an abstract-type, or a concrete-type, subtype is specifically defined for only these. Types of different node classes are generally unrelated as subtypes. As with the other context-dependent relations, the details of the subtype relation are described as each relevant node class is described.

The subtype relation is an ordering relation (i.e., a partial order) so type equivalence is defined in terms of subtype—two types are equivalent when each is a subtype of the other. Also, the inverse of the subtype relation is referred to as the *supertype* relation, that is, T is a supertype of T' if and only if T' is a subtype of T.

Because the subtype relation is an ordering relation we may speak of the *maximal* type of a set of types. The maximal type of a set of types is any type that is a supertype of all the rest; the maximal type of an empty set is *Void* (see A.22). If a set of types does not have a maximal type then the maximal type is *Error* (see A.25). Note that special subtype rules apply for the types *Error* and *Raise* (see A.24.2).

This concludes the overview of Acer's primary context-dependent relations.

A.4 Lexical structure

This section presents a detailed description of Acer's lexical structure, i.e., its ASCII encoding. Every Acer program is represented as an abstract syntax tree, which can be encoded as stream of ASCII characters. The node structure of such a stream is determined by parsing, that is, by analyzing the stream of characters to determine its tokens (lexical analysis), and analyzing the stream of tokens to determine its nodes (syntax analysis).

A.4.1 The character set

Acer partitions the ASCII character set as follows:

```
⟨#LowercaseLetter⟩ ::=
    a || b || c || d || e || f || g || h || i || j || k || l || m ||
    n || o || p || q || r || s || t || u || v || w || x || y || z

⟨#UppercaseLetter⟩ ::=
    A || B || C || D || E || F || G || H || I || J || K || L || M ||
    N || O || P || Q || R || S || T || U || V || W || X || Y || Z || _

⟨#SymbolicLetter⟩ ::=
    ! || # || $ || & || * || + || - || / || < || = || > || ? || \ || ^ || | || ~
```

⟨*#Punctuation*⟩ ::=
`%` ‖ `'` ‖ `(` ‖ `)` ‖ `,` ‖ `.` ‖ `:` ‖ `;` ‖ `[` ‖ `]` ‖ `‘` ‖ `{` ‖ `}` ‖ `@`

⟨*#Digit*⟩ ::= **0** ‖ **1** ‖ **2** ‖ **3** ‖ **4** ‖ **5** ‖ **6** ‖ **7** ‖ **8** ‖ **9**

⟨*#DoubleQuote*⟩ ::= `"`

⟨*#Tab*⟩ ::= ASCII 9

⟨*#LineFeed*⟩ ::= ASCII 10

⟨*#CarriageReturn*⟩ ::= ASCII 13

⟨*#Space*⟩ ::= ASCII 32

⟨*#Other*⟩ ::= ASCII 0–8, 11–12, 14–31, 127

If an extended ASCII is used (such as IBM extended ASCII), the additional characters are, by default, included in other. However, if various characters from other produce desirable graphic characters, they may instead be included in one of the first four groups.

A.4.2 Token

A lexical analyzer for Acer recognizes five sorts of token: comment, identifier, keyword, punctuation, and lexical-literal. Each will be discussed in turn.

A.4.2.1 Comment

A comment-token begins with a '`%`' and terminates at the end of the line (i.e., the next carriage-return or line-feed):

⟨*Comment*⟩ ::=
`%` ⦃ ⟨*#LowercaseLetter*⟩ ‖ ⟨*#UppercaseLetter*⟩ ‖ ⟨*#SymbolicLetter*⟩ ‖
⟨*#Punctuation*⟩ ‖ ⟨*#Digit*⟩ ‖ ⟨*#DoubleQuote*⟩ ‖
⟨*#Tab*⟩ ‖ ⟨*#Space*⟩ ‖ ⟨*#Other*⟩ ⦄

A comment-token annotates the non-comment-token preceding it. Thus every non-comment-token has an associated annotation list consisting of the bodies of its trailing comment-tokens, each stripped of its leading '`%`'. A syntax analyzer, when accepting tokens, maps each token to a particular node called its owner, i.e., the node whose grammar rule requires the token's occurrence. When a token with annotations is accepted, the annotations are inserted at the end of the annotation list of that owner node.

Since every Acer node, with the possible exception of empty nodes (see A.5.1), owns at least one token in a lexical encoding, every Acer node can be annotated by annotating its tokens. Thus Acer's comment facility provides a concrete representation for nodes annotated with lines of text. Note that the annotation list of a node (or token) is not itself a node—it is a textual object consisting of lines of text.

Comments can be used in many different ways, for example, as documentation, as directives of various sorts, as cross-reference information, and so on. So that the role of each annotation (i.e., each comment line) is specified, the first character in an annotation (i.e., the character after the '%') determines its role.

Only the role of documentation is specified in this appendix: a space indicates documentation. The use of comments for other roles has not been fully explored. Nevertheless, in the future: a '$' should be used to indicate a compiler directive, a '%' should be used to indicate a formatting directive, and a '!' should be used to indicate an assertion.

A.4.2.2 Identifier

Acer provides two classes of identifier, one for denoting types and the other for denoting values. A type-identifier consists of an initial uppercase letter followed by zero or more lowercase letters, uppercase letters, and digits:

```
⟨TypeIdentifier⟩ ::=
    ⟨#UppercaseLetter⟩ ⦃ ⟨#LowercaseLetter⟩ ‖ ⟨#UppercaseLetter⟩ ‖
                         ⟨#Digit⟩ ⦄
```

A value-identifier is either *alphabetic* or *symbolic*, and hence consists of either an initial lowercase letter followed by zero or more lowercase letters, uppercase letters, and digits, or a sequence of one or more symbolic-letters:

```
⟨ValueIdentifier⟩ ::=
    ⟨#LowercaseLetter⟩ ⦃ ⟨#LowercaseLetter⟩ ‖ ⟨#UppercaseLetter⟩ ‖
                         ⟨#Digit⟩ ⦄ ‖
    ⟨#SymbolicLetter⟩ ⦃ ⟨#SymbolicLetter⟩ ⦄
```

Symbolic-letters in value-identifiers allow programmers to define their own operator symbols.

During parsing, each identifier-token gives rise to an identifier node containing the token's spelling, i.e., the owner node.

A.4.2.3 Keyword

The following tokens are used as keywords in Acer and cannot be used as identifiers:

Any	**andif**	**array**	**be**
becomes	**begin**	**Dynamic**	**dynamic**
do	**code**	**Enumeration**	**end**
else	**elsif**	**exception**	**Function**
finally	**for**	**function**	**if**
in	**inspect**	**is**	**isnot**
keep	**let**	**nothing**	**Operator**
Option	**of**	**ord**	**orif**
pointer	**Record**	**raise**	**record**
reference	**Then**	**Tuple**	**TYPE**
then	**try**	**trying**	**tuple**
Variant	**val**	**var**	**variant**
when	**with**		

The spellings of keyword-tokens are not stored in the syntax tree. Their sole purpose is to direct parsing (reading) and the association of annotations—they play no role in semantics.

A.4.2.4 Punctuation

The following tokens are used as punctuation in Acer:

```
% ' ( ) , . : ; [ ] ` { } @
```

Each punctuation character constitutes a token.

As with keyword-tokens, the spellings of punctuation-tokens are not stored in the syntax tree.

A.4.2.5 Lexical Literal

Acer provides lexical representations for integers, reals, characters and strings.

An integer-literal consists of an optional '-' and a sequence of one or more digits:

$$\langle \mathit{IntegerLiteral} \rangle ::= [\![\; - \;]\!] \; \langle \#\mathit{Digit} \rangle \; \{\!\{ \; \langle \#\mathit{Digit} \rangle \; \}\!\}$$

A real-literal consists of an optional '-', a sequence of one or more digits, a '.', another sequence of one or more digits, and an optional trailing power indication, which is an '**E**' or '**e**', an optional '-', and a sequence of one or more digits:

⟨*RealLiteral*⟩ ::=
⟦ - ⟧ ⟨*#Digit*⟩ ⦃ ⟨*#Digit*⟩ ⦄ . ⟨*#Digit*⟩ ⦃ ⟨*#Digit*⟩ ⦄
⟦ ⦅ **e** ‖ **E** ⦆ ⟦ - ⟧ ⟨*#Digit*⟩ ⦃ ⟨*#Digit*⟩ ⦄ ⟧

During lexical analysis of real-literals an '**e**' is automatically converted to an '**E**'.

A character-literal consists of a '‘', any character except a line-feed or carriage-return, and a '’':

⟨*CharacterLiteral*⟩ ::=
‘ ⦅ ⟨*#LowercaseLetter*⟩ ‖ ⟨*#UppercaseLetter*⟩ ‖ ⟨*#SymbolicLetter*⟩ ‖
⟨*#Punctuation*⟩ ‖ ⟨*#Digit*⟩ ‖ ⟨*#Space*⟩ ‖ ⟨*#DoubleQuote*⟩ ‖
⟨*#Other*⟩ ⦆ ’

A string-literal consists of a '"', zero or more characters other than line-feed, carriage-return or '"', and a '"':

⟨*StringLiteral*⟩ ::=
" ⦃ ⟨*#LowercaseLetter*⟩ ‖ ⟨*#UppercaseLetter*⟩ ‖ ⟨*#SymbolicLetter*⟩ ‖
⟨*#Punctuation*⟩ ‖ ⟨*#Digit*⟩ ‖ ⟨*#Space*⟩ ‖ ⟨*#Other*⟩ ⦄ "

During parsing, each lexical-token gives rise to a integer-literal, real-literal, character-literal, or string-literal node containing the token's spelling, i.e., the owner node. The spelling of a character- or string-literal node does not include the delimiting single or double quotation marks. Nevertheless, in the graphical representation these delimiting marks are illustrated as part of the spelling so as to distinguish character- and string-literals from identifiers.

A.4.3 Delimiter

To prevent ambiguity, adjacent tokens in an Acer program can and sometimes must be delimited by white-space (i.e., space, tab, carriage-return, or line-feed). Comments, punctuations, character-literals, and string-literals are self delimiting and therefore need no further delimitation. Symbolic value-identifiers need only be delimited from adjacent symbolic value-identifiers and from integer- and real-literals if the identifier is '–'. The remaining sorts of token—alphabetic value-identifier, type-identifier, keyword, integer-literal, and real-literal—must be delimited when adjacent.

A.4.4 Lexical alternatives

The viewing of nodes in this appendix, as has been the case throughout this thesis, is enhanced with the use of proportional fonts and type cues. The following conventions are followed:

- Comments are Roman.
- Keywords are **bold**.
- Alphabetic value-identifiers are *slanted*.
- Type-identifiers are *Italic*.
- Character-literals, string-literals, and symbolic value-identifiers are `typewriter`.

This concludes the description of Acer's ASCII encoding.

A.5 Special nodes

Recall that Acer's design is guided by the view that it is as important to specify how language constructs are represented and manipulated as it is to specify what they mean. For the manipulation of nodes, Acer provides four special node classes, namely empty, type-denoter, value-denoter, and arbitrary-list. Each will be discussed in turn.

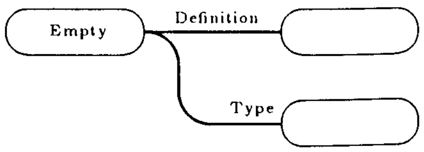
A.5.1 Empty

For representing a missing, optional, construction component, Acer provides empty, which is defined to be a childless construction:

⟨*Empty*⟩ ::= **nothing**

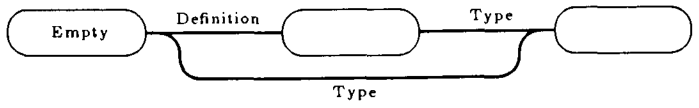
The following construction classes have optional components and may therefore contain empty nodes: array-literal, conditional, dynamic-inspection, fixed-value-binding, fixed-value-declaration, function-literal, iteration, keep-trying, raise, try, type-declaration, type-binding, type-when-branch, value-when-branch, variable-value-binding, variable-value-declaration, variant-inspection, and variant-type.

The context-dependent relations for an empty node are illustrated as

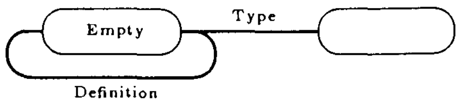


An empty node has a type and a definition, which are determined by the context in which it appears. Therefore, the relations are described as each of the above construction classes is described.

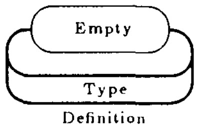
In general, if an empty node is not its own definition, its type is given by that of its definition:



If an empty node is its own definition, its type is determined by its context:



For an unattached empty node, and for an empty node appearing in an arbitrary-list, the context-dependent relations illustrated as



Such an empty node is its own type and its own definition.

A.5.1.1 Printing empty nodes

Empty nodes are usually invisible in a lexical encoding. However, in Acer an empty node can be treated as any other node by printing it as the keyword **nothing**. For example,

if *x* **then** *y* **end**

and

if *x* **then** *y* **else nothing end**

provide different views of the empty node representing the missing default-branch of the conditional (see A.23.3). When an empty node is visibly printed, any associated optional keywords are also printed, as is **else** above.

The reason for this unconventional treatment of empty nodes stems from the fact that every Acer node, except for an empty node, produces at least one token in a lexical encoding. Therefore, the above treatment of empty nodes eliminates the only exception to this general rule while still providing for the conventional lexical encoding in which empty nodes are invisible.

Empty nodes, like every other node, can be annotated. However, since annotations associate with a node by annotating its tokens, an annotated empty node must be printed as **nothing** to ensure that its annotations do not become reassociated with a different node.

A.5.1.2 The unattached empty node

The other use of empty nodes in Acer is to ensure that all nodes have a parent node, including even unattached nodes. Therefore, Acer specifies that there exist a special empty node, called *the unattached empty node*, which is the parent of all unattached nodes including itself. The unattached empty node is unique among nodes in that it is a parent and yet has no children.

A.5.2 Denoter

Acer provides denoters to facilitate the syntactic representation of semantic objects derived from existing nodes, that is, to represent types and definitions. To see how the need for denoters arises consider the tuple-literal

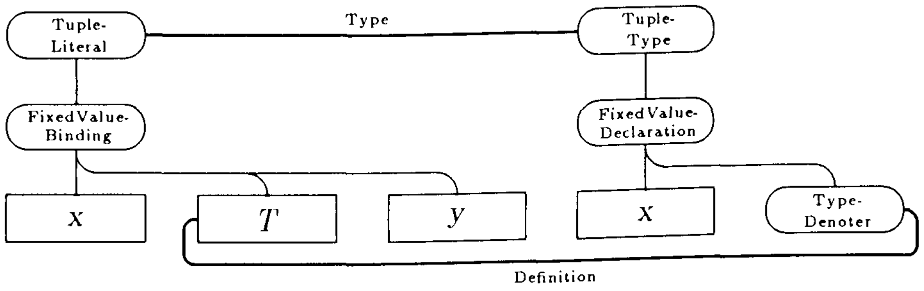
tuple let $x : T$ **be** y **end**

The type of this tuple-literal is

Tuple $x : T$ **end**

but since this derived tuple-type does not appear in the syntax tree containing the tuple-literal, the appropriate defining-occurrence of T cannot be determined.

Denoters circumvent this problem by a supporting references to nodes that are unreachable via context. For example, the above situation is solved using a type-denoter:



A denoter can be thought of as an anonymous identifier with a predetermined defining-occurrence. To be more precise, every denoter has a definition and stands in place of that definition.

Acer has two classes of denoter:

⟨*Denoter*⟩ ::= ⟨*TypeDenoter*⟩ || ⟨*ValueDenoter*⟩

where a type-denoter is defined as

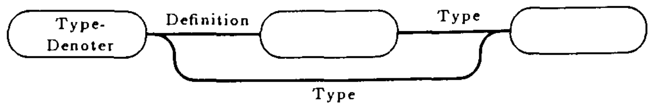
⟨*TypeDenoter*⟩ ::= `{[]}`

and a value-denoter is defined as

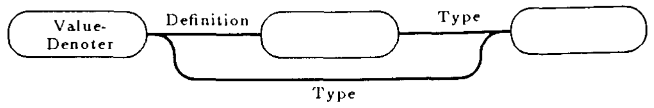
$\langle ValueDenoter \rangle$ ::= {()}

The definition of a type-denoter must be empty or a type and the definition of a value-denoter must be empty or a value. A denoter never has another denoter as its definition.

The context-dependent relations for a type-denoter are illustrated as



and those for a value-denoter are illustrated as



In general, a denoter has a definition and its type is given by the type of that definition.

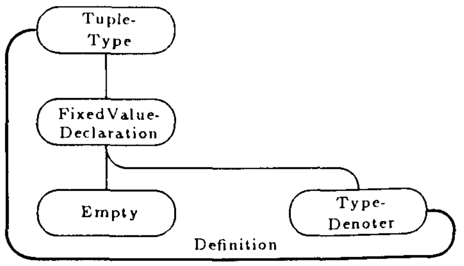
The graphical representation of a denoter illustrates its definition but usually nodes are viewed in terms of a lexical encoding. Therefore, to enhance the readability of a denoter's lexical encoding, the definition of a denoter may be printed within the denoter's brackets. Thus the above tuple-type could be printed as

Tuple x : {[]} **end**

or as

Tuple x : {[T]} **end**

Since a denoter can recursively contain itself, it is not always possible to print its definition. For example, a recursive tuple-type illustrated as



must be lexically encoded as

Tuple : {[]} **end**

since printing its denoter leads to an infinite regression. Also, because definitions can be shared, printing the definition of each denoter may result in a given node being printed an arbitrary number of times. In general, the definition of a denoter should only be printed if it is relatively small, for example, if it is an identifier.

A denoter should never appear in a user program since its definition cannot be determined via context—a denoter should appear only in a derived node, in which case its definition is determined so as to suit the role it plays in that node.

Because denoters should not appear in user programs, it would be reasonable for a parser to simply reject denoters. However, the following adhoc approach is chosen instead. A parser, when accepting a type- or value-denoter, assigns the unattached empty node as the denoter's definition, unless the denoter has a printed definition, in which case it assigns the parsed type or value as the denoter's definition. It is important to realize that in either case the original intent of the denoter is likely lost; in the first case an empty definition is assigned and in the second case a *copy* of the definition is assigned. Therefore, semantics will likely be altered by printing and parsing when denoters are involved. This is of no concern however since denoters should only appear in derived nodes, which are not typically parsed.

Essentially, a type-denoter is treated as if it were defined as

$$\langle \mathit{TypeDenoter} \rangle ::= \texttt{\{[} \ [\![\ \langle \mathit{Definition}{:}\mathit{Type} \rangle \ ]\!] \ \texttt{]\}}$$

and a value-denoter is treated as if it were defined as

$$\langle \mathit{ValueDenoter} \rangle ::= \texttt{\{(} \ [\![\ \langle \mathit{Definition}{:}\mathit{Value} \rangle \ ]\!] \ \texttt{)\}}$$

However, the definition component of a denoter is treated as a context-dependent relation, instead of as a context-free relation as implied by the above grammar rules. This is to ensure that the context-free relations specify a tree, instead of a more general graph.

A.5.2.1 Definition-copy

Denoters are closely involved in the notion of a *definition-copy*. Producing the definition-copy of a node involves recursively copying the tree rooted at that node, including the definition of each denoter, and substituting certain nodes with specified replacements. To see how the need for definition-copy arises, consider determining the type of the function-call

$$f\,(\,T,\, x)$$

where x has type T and f has type

Function (*Type*:: **Any**; *value*: *Type*) **Tuple** : *Type* **end end**

The type of the call is determined by copying the result-type of the function-type and substituting T in place of each occurrence of *Type*:

Tuple : T **end**

Since T is not visible in this newly derived tuple-type, a type-denoter must be used:

Tuple : $\{[T]\}$ **end**

where T is precisely the node T in the argument-list of the function-call.

In general, we shall speak of the definition-copy of node y with x' for x, where x and x' can be any combination of signature, argument-list, dynamic-type, tuple-type, or record-type. Producing such a definition-copy involves creating y', a copy of the tree rooted at y, in which each applied-occurrence of an identifier in the name-layer of x is replaced by a denoter with a definition that, depending of the node class of x', is either the corresponding expression in x' or the defined-identifier of the corresponding binding or declaration in x'. Note that when y is an empty node or denoter, definition-copy instead copies the definition of y.

When producing the definition-copy of a node, certain subnodes may be unaffected by substitution, that is, certain nodes contain neither applied-occurrences of identifiers in the name-layer of x nor applied-occurrences of identifiers that are copied. For such nodes, definition-copy can substitute a denoter to the original node in place of a copy of that node. This results in a more efficient representation since fewer nodes are used to represent the same object.

A.5.2.2 Definition-copy-at

Closely associated with the notion of a definition-copy is the notion of a *definition-copy-at*. The definition-copy-at of a node also results in the creation of a new unattached node that defines the same object as the original. However, for definition-copy, the new node is unattached and so denoters are used to refer to nodes not accessible via context, whereas for definition-copy-at, the new node is expressed in terms of the scope at some target node l. Thus the definition-copy-at of a node y at a location l involves producing a denoterless definition-copy of y expressed in terms of the scope at l.

That the definition-copy-at can generally be produced demonstrates that objects expressed using denoters can also be expressed without using denoters, given an appropriate scope. For example, consider the recursive tuple-type

Tuple : {[]} **end**

where the definition of the type-denoter is tuple-type itself. The definition-copy-at of this tuple-type at the unattached empty node results in

{**let** *Unnamed* **be Tuple** : *Unnamed* **end**; *Unnamed*}

Note that the definition-copy-at of a node y at a target node l cannot be produced for all possible targets because the objects used by y must be visible at l. Thus definition-copy-at fails when attempting to express a node 'outside of its scope.'

A.5.3 Arbitrary-list

The last of Acer's special nodes is the arbitrary-list, which is defined to be a list of arbitrary nodes by the list rule

⟨*ArbitraryList*⟩ ::= **arbitrary** ⦃ [⟨*Arbitrary*⟩] ⦄ **end**

An arbitrary-list allows arbitrary nodes to be grouped as a syntactic unit. It has no semantic significance and cannot appear in any Acer program. Arbitrary-lists, like denoters and the unattached empty node, are a feature provided for manipulating Acer programs, rather than for expressing Acer programs.

Note that the brackets delimiting elements in an arbitrary-list are necessary because the elements are essentially out of context. In Acer, element delimiters in lists are generally optional because it is always clear where one element ends and the next one begins. However, because an element in an arbitrary-list can be a partial construct (i.e., miscellaneous), delimitation is mandatory. For example, an arbitrary-list containing an array-list, which is normally the first component of an array-literal (see A.31), appears as

arbitrary [**array** 1 2 3] **end**

Without delimitation it would appear as

arbitrary array 1 2 3 **end**

which would look as if the terminating **end** of the arbitrary-list is missing.

This concludes the discussion of Acer's special node classes.

A.6 Identifier

For referring to objects by name, Acer provides identifiers:

⟨*Identifier*⟩ ::= ⟨*TypeIdentifier*⟩ ‖ ⟨*ValueIdentifier*⟩

where type-identifier is defined as

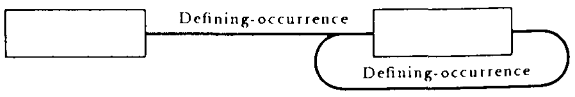
⟨*TypeIdentifier*⟩ ::=
⟨*#UppercaseLetter*⟩ ⦃ ⟨*#LowercaseLetter*⟩ ‖ ⟨*#UppercaseLetter*⟩ ‖ ⟨*#Digit*⟩ ⦄

and value-identifier is defined as

⟨*ValueIdentifier*⟩ ::=
⟨*#LowercaseLetter*⟩ ⦃ ⟨*#LowercaseLetter*⟩ ‖ ⟨*#UppercaseLetter*⟩ ‖ ⟨*#Digit*⟩ ⦄ ‖
⟨*#Symbol*⟩ ⦃ ⟨*#Symbol*⟩ ⦄

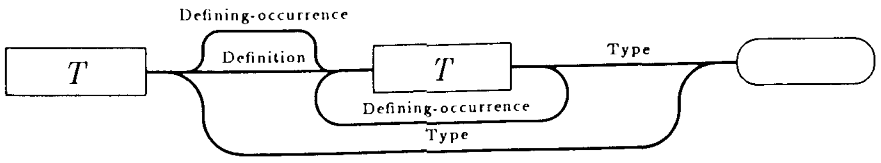
An identifier appears in context as either a *defining-occurrence* or an *applied-occurrence.* It is a defining-occurrence if it appears as the defined-identifier of a binding, declaration, iterator-element, type-when-branch, or value-when-branch, or if it appears as an element of an enumeration- or option-type, i.e., if it appears in a defining-occurrence context; it is an applied-occurrence otherwise.

Each identifier has a unique defining-occurrence. A defining-occurrence is its own defining-occurrence and an applied-occurrence has a defining-occurrence determined by identifier-lookup:

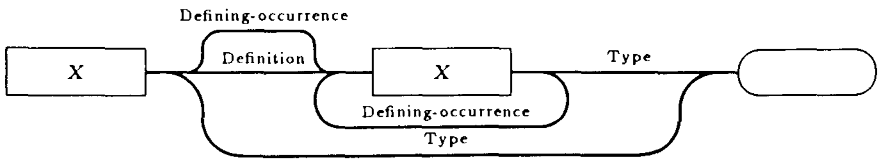


The context-dependent relations for a defining-occurrence are described as each defining-occurrence context is described. Those for an applied-occurrence are given below.

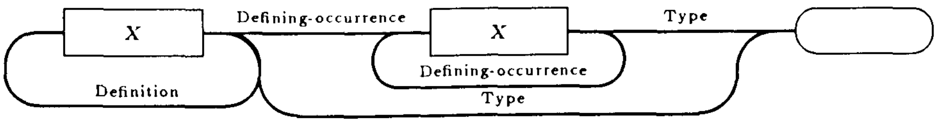
The context-dependent relations for an applied-occurrence of a type-identifier are illustrated as



Those for an applied-occurrence of a value-identifier are illustrated either as



or as



depending on whether it is a *fixed-identifier* or a ***variable-identifier***, respectively. A value-identifier is a variable-identifier if its defining-occurrence appears as the defined-identifier of a variable-value-binding or a variable-value-declaration; it is a fixed-identifier otherwise.

In general, the type of an applied-occurrence is given by that of its defining-occurrence. Also, the definition of an applied-occurrence is given directly by its defining-occurrence, except for a variable-identifier, which act as its own definition. A variable-identifier acts as its own definition because it can be modified by assignment (see A.30.4) and so potentially denotes a different object each time it is evaluated; a fixed-identifier or type-identifier, on the other hand, must denote a single object throughout its scope and so the identity of that object is represented by the unique defining-occurrence.

The subtype rules for type-identifiers are described with respect to the subtype rules for abstract-types (see A.20).

A.6.1 Reused-identifier

In most statically-scoped languages there is the problem of name hiding due to the nested reuse of names. For example, the outer definition of x in

{**let** x **be** 10; {x * {**let** x **be** 20; {x + x}}}}

is not visible in the inner block, which also introduces an x.

For referring to identifiers hidden by reuse, Acer provides reused-identifiers:

⟨*ReusedIdentifier*⟩ ::=
  ⟨*ReusedTypeIdentifier*⟩ || ⟨*ReusedValueIdentifier*⟩

A reused-type-identifier consists of an identifier and a depth-indicator:

⟨*ReusedTypeIdentifier*⟩ ::=
  ⟨*Identifier*:*TypeIdentifier*⟩ ' [⟨*DepthIndicator*:*IntegerLiteral*⟩]

as does a reused-value-identifier:

> ⟨*ReusedValueIdentifier*⟩ ::=
> ⟨*Identifier*: *ValueIdentifier*⟩ ' [⟨*DepthIndicator*:*IntegerLiteral*⟩]

The depth-indicator must be non-negative.

In general, the context-dependent relations for a reused-identifier are the same as those for its identifier. Thus a reused-identifier is treated as an applied-occurrence that has a defining-occurrence, a definition, and a type. Also, a reused-value-identifier is considered to be either a fixed-identifier or variable-identifier depending on whether its identifier is a fixed-identifier or variable-identifier.

A.6.2 Scope

Normally, when identifier-lookup begins at a regular identifier, searching terminates when a matching defining-occurrence is found. However, when lookup begins at the identifier of a reused-identifier, searching terminates when the $n+1$ matching defining-occurrence is found, where n is value of the depth-indicator. This way an identifier in an outer scope that is reused in an inner scope can still be referenced in the inner scope. Thus the above example could be expressed as

> {**let** *x* **be** 10; {**let** *x* **be** 20; {*x*'[1] * {*x* + *x*}}}}

Without reused-identifiers it would not be possible to provide the definition-copy-at facility since the scope of an object could have holes where the object is not visible.

A.7 Binding

For introducing named expressions, Acer provides bindings:

> ⟨*Binding*⟩ ::= ⟨*TypeBinding*⟩ ‖ ⟨*ValueBinding*⟩

Type-bindings come in only one form but value-bindings come in one of two forms:

> ⟨*ValueBinding*⟩ ::= ⟨*FixedValueBinding*⟩ ‖ ⟨*VariableValueBinding*⟩

Type-binding is defined as

> ⟨*TypeBinding*⟩ ::=
> **let** ⟨*DefinedIdentifier*:*TypeIdentifier*⟩ ⟦ : : ⟨:*Type*⟩ ⟧ **be**
> ⟨*Definition*:*Type*⟩

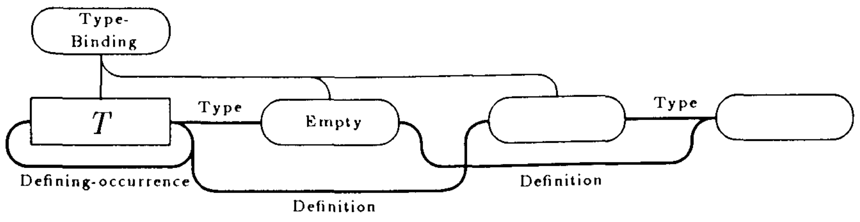
fixed-value-binding is defined as

> ⟨*FixedValueBinding*⟩ ::=
> **let** ⟨*DefinedIdentifier*:*ValueIdentifier*⟩ ⟦ : ⟨:*Type*⟩ ⟧ **be**
> ⟨*Definition*: *Value*⟩

and variable-value-binding is defined as

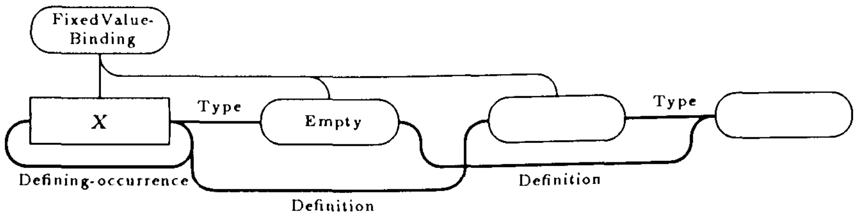
> ⟨*VariableValueBinding*⟩ ::=
> **let var** ⟦ ⟨*DefinedIdentifier*:*ValueIdentifier*⟩ ⟧ ⟦ : ⟨:*Type*⟩ ⟧ **be**
> ⟨*Definition*: *Value*⟩

The context-dependent relations for a type-binding are illustrated as

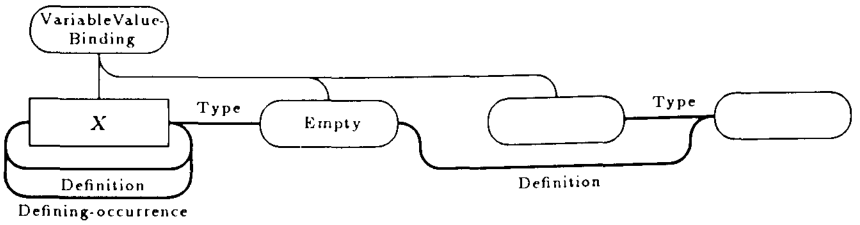


The type of a type-binding must be a kind.

Similarly, the context-dependent relations for a fixed-value-binding are illustrated as

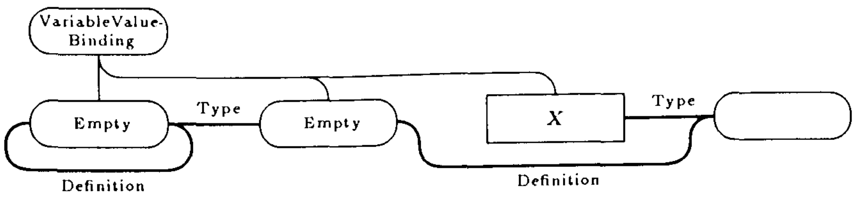


But the context-dependent relations for a variable-value-binding are slightly different:



The defined-identifier of a variable-value-binding is its own definition because it is a variable-identifier (see A.6).

The defined-identifier of a variable-value-binding is optional, but the same context-dependent relations apply:



Anonymous variable-value-bindings are used to specify anonymous, variable, data-structure components; anonymous, fixed, data-structure components are specified simply using expressions.

In general, a binding introduces a defined-identifier as the name for its definition. Also, a binding optionally indicates the type of its definition; if a type is indicated, the type of the definition must be a subtype of that type. We shall now consider the role bindings play in scoping, which is followed by a discussion of how bindings are evaluated.

A.7.1 Scope

According to Acer's grammar, the parent of a binding must be of class arbitrary-list, argument-list, tuple-literal, binding-list, dynamic-literal, empty, or record-literal. The behavior of identifier-lookup with respect to a binding is determined by this class.

If the node class of a binding's parent is arbitrary-list or empty, identifier-lookup searches the name-layer consisting of just the binding's defined-identifier; lookup continues only if the defining-occurrence is not found.

If the node class of a binding's parent is one of the remaining possible classes, no search is performed and lookup continues with the parent. But in this case, the binding contributes its defined-identifier to the name-layer of its parent and lookup searches this name-layer when it reaches the parent. Hence, regardless of a binding's context, its defined-identifier is visible within it.

A.7.2 Dependency analysis

Acer's scope rules permit pathological bindings such as

```
let x be x
```

but the denotation relation detects these errors. Other situations, such as the binding-list (see A.12)

```
{let x be y;
 let y be {x + 1};
```

are erroneous because a correct evaluation is not possible. However, similar situations such as

```
{let x be tuple y end;
 let y be tuple x end;
```

are not erroneous because, as we shall see, a correct evaluation is possible. Erroneous dependencies are detected by *dependency analysis.*

Dependency analysis is primarily concerned with values, in particular, with evaluation order of bindings during the initialization of a binding-list (see A.12), an argument-list (see A.15), or an aggregate-literal (see A.14). (Types are not evaluated and all anomalous type dependencies are detected by the definition and denotation relations.) For flexibility Acer specifies that bindings may be evaluated in any order that does access uninitialized bindings. Thus programmers should not assume a particular evaluation order. Nevertheless, Acer also specifies that bindings must appear in an order that can be correctly evaluated left-to-right. Therefore, an implementation of Acer may choose to evaluate bindings either left-to-right or in some other correct order.

Dependency analysis is based on the notion of *direct references* and *indirect references.* A value-binding b is said to directly reference a value-binding b' if b *contains* an applied-occurrence of the defined-identifier of b'. (Containment includes applied-occurrences recursively contained by denoters.) A value-binding b is said to indirectly reference a value-binding b' if b directly references b' or any value-binding b'' directly referenced by b indirectly references b', i.e., the indirect references of b are the transitive closure of its direct references.

Using these notions, Acer defines a *delayed-occurrence* to be an applied-occurrence of a value-identifier that occurs before or within the binding that introduces it, or before or within a binding indirectly referenced by the binding that introduces it. A delayed-occurrence, then, is simply an applied-occurrence that occurs before its definition is complete. For example, in the binding-list

```
{let f be function () y.z end;
 let x be f ();
 let y be tuple let z be 10 end;
```

the applied-occurrence of f in the binding for x is a delayed-occurrence. (In fact, it is an invalid delayed-occurrence, as we shall see in a moment.) Note that the applied-occurrence of f is not a forward reference in the conventional sense.

Formally, a delayed-occurrence i is valid if the following two conditions hold for i and b, the binding that introduces i. First, the definition of b must denote a literal. And second, for every x enclosing i but not enclosing b, either x must denote a literal or x must be enclosed by a function-literal or type that does not also enclose b.

The use of delayed-occurrences is quite restricted. To understand why delayed-occurrences can be supported at all one must realize that a literal is either a constant (i.e., a character-literal, integer-literal, literal-selection, real-literal, reused-value-identifier, string-literal, value-identifier, or void-literal) or a *constructor* (i.e., an array-literal, dynamic-literal, exception-literal, function-literal, record-literal, reference-literal, tuple-literal, or variant-literal). A delayed-occurrence that references a constant is clearly valid and a delayed-occurrence that references a constructor is valid because a constructor can be allocated in advance and initialized later. Hence, a delayed-occurrence is supported as a reference to either a constant or an uninitialized data structure.

To prevent access to components of uninitialized data structures, delayed-occurrences are restricted to occur only within types and constructors. However, delayed-occurrences are permitted anywhere within a function-literal's body, even within non-literal expressions (as with $y.x$ above), because a function-literal is a constructor and any value referenced in its body will not be accessed until the function is called; the restrictions on delayed-occurrences prevent such calls until all values referenced by the function are fully defined. For example, the call to f above is invalid precisely because the applied-occurrence f is a delayed-occurrence enclosed by a non-literal value, the function-call $f()$. A delayed-occurrence enclosed by a type is permitted because types are not evaluated and hence values enclosed by types are not evaluated.

A.8 Declaration

For introducing named, hidden expressions, known only to have a type that is a subtype of an indicated type, Acer provides declarations:

$$\langle \mathit{Declaration} \rangle ::= \langle \mathit{TypeDeclaration} \rangle \; \| \; \langle \mathit{ValueDeclaration} \rangle$$

For each class of binding, there is a corresponding class of declaration. Hence, there is one class of type-declaration and there are two classes of value-declaration:

$$\begin{aligned}&\langle \mathit{ValueDeclaration} \rangle ::= \\ &\quad \langle \mathit{FixedValueDeclaration} \rangle \; \| \; \langle \mathit{VariableValueDeclaration} \rangle\end{aligned}$$

Every declaration consists of an optional defined-identifier and a type. Type-declaration is defined as

$$\langle \mathit{TypeDeclaration} \rangle ::= [\![\; \langle \mathit{DefinedIdentifier{:}TypeIdentifier} \rangle \;]\!] \; :: \; \langle {:}\mathit{Type} \rangle$$

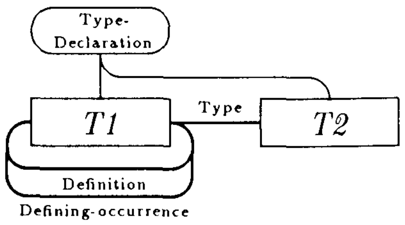
Fixed-value-declaration is defined as

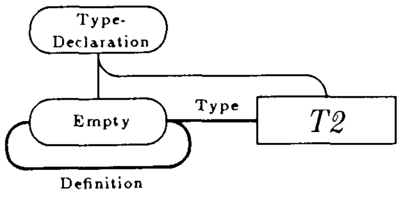
⟨*FixedValueDeclaration*⟩ ::=
[[⟨*DefinedIdentifier*: *ValueIdentifier*⟩]] : ⟨:*Type*⟩

And variable-value-declaration is defined as

⟨*VariableValueDeclaration*⟩ ::=
var [[⟨*DefinedIdentifier*: *ValueIdentifier*⟩]] : ⟨:*Type*⟩

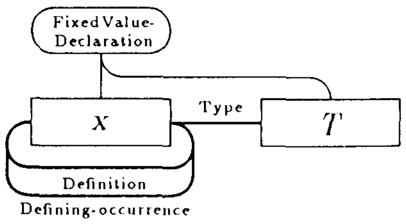
The context-dependent relations for type-declarations are illustrated as

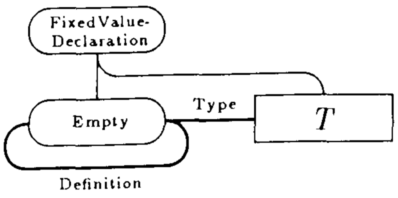




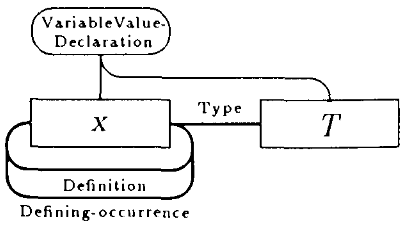
The type of a type-declaration must be a kind because the type of a type must be a kind. A type-identifier introduced by a type-declaration is considered to be an abstract-type and so the subtype rules for such an identifier are described with respect to the subtype rules for abstract-types (see A.20).

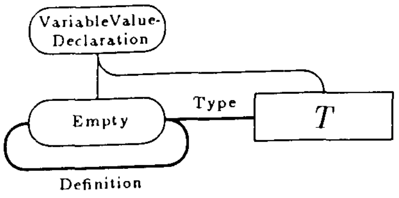
The context-dependent relations for fixed-value-declarations are analogous to those for type-declarations:





As are those for a variable-value-declaration:





In general, a declaration introduces a defined-identifier as the name for a hidden expression; only the type of this expression is given. A declaration specifies the type of a parameter or data-structure component and hence its definition is determined dynamically, not statically. The defined-identifier of a declaration is known only to denote an expression with a type that is a subtype of the indicated type and so it stands as its own definition. The defined-identifier of a declaration is optional so that anonymous parameters or data-structure components can be specified.

Consider now the scope rules for declarations, which closely mirror those for bindings.

A.8.1 Scope

According to Acer's grammar, the parent of a declaration must be of class arbitrary-list, dynamic-type, empty, record-type, signature, or tuple-type. The behavior of identifier-lookup with respect to a declaration is determined by this class.

If the node class of a declaration's parent is arbitrary-list or empty, identifier-lookup searches the name-layer consisting of just the declaration's defined-identifier; lookup proceeds only if the defining-occurrence is not found.

If the node class of a declaration's parent is one of the remaining possible classes, no search is performed and lookup continues with the parent. But in this case, the declaration contributes its defined-identifier to the name-layer of its parent and lookup searches this name-layer when it reaches the parent. Hence, regardless of a declaration's context, its defined-identifier is visible within it.

A.8.2 Deriving a declaration from an argument

A declaration can be derived from an argument (a binding or expression) to represent the argument's type while hiding the argument's definition. Expressions derive anonymous declarations and bindings derive named declarations, except for variable-value-bindings with an empty defined-identifier, which derive anonymous declarations instead. For example, the arguments

> **let** *T1* :: *T2* **be** *T3*
> **let** *x* : *T* **be** *y*
> **let var** *x* : *T* **be** *y*
> *T1*
> *x*
> **let var be** *x*

respectively derive the declarations

> *T1* :: {[*T2*]}
> *x* : {[*T*]}
> **var** *x* : {[*T*]}
> :: {[*T2*]}
> : {[*T*]}
> **var** : {[*T*]}

The defined-identifier of a derived declaration is determined by the defined-identifier of the argument, if it has one, and the type is determined by the definition-copy of the argument's type with the defined-identifiers of the declarations for the defined-identifier of the bindings. The notion of derived declarations is used to specify the type of an aggregate-literal (see A.14) and to specify the way in which a signature is derived from an argument-list (see A.15).

A.9 The global name-layer

Earlier it was mentioned that the unattached empty node has associated with it the global name-layer. This name-layer is searched when identifier-lookup reaches the unattached empty node, i.e., when lookup's search node is unattached. If lookup fails to find the matching defining-occurrence in the global name-layer, it terminates and yields the unattached empty node. Hence, the defining-occurrence of an undefined applied-occurrence is given by the unattached empty node.

A defining-occurrence is introduced into the global name-layer in one of two ways, depending on whether it is a type or a value. Each will be considered in turn.

A.9.1 Global value-identifier

A global value-identifier is introduced through compilation. The simplest way to introduce a global value is to compile a fixed-value-binding, which has the following effects. First, a translation used to compute the binding's definition (value), is produced and stored. And second, a declaration of the value is produced and stored. Presumably every Acer environment will have access to a file system in which it can permanently store nodes, translations, and other information.

The declaration produced by compilation is a fixed-value-declaration and consists of a defined-identifier that is a copy of that of the binding, and a type that is a definition-copy-at of the type of the binding at the unattached empty node, i.e., a definition-copy-at with global scope. Storing the declaration has the effect of including its defined-identifier in the global-name-layer.

For example, the binding

let x **be tuple** x **end**

can be compiled to produce the declaration

x : {**let** *Unnamed* **be Tuple** : *Unnamed* **end**; *Unnamed*}

The binding for x can also explicitly indicate the type of its derived declaration, e.g., the binding

let x : {**let** T **be Tuple** : T **end**; T} **be tuple** x **end**

can be compiled to produce the equivalent declaration

x : {**let** T **be Tuple** : T **end**; T}

After compiling either of the above two bindings for x, a declaration containing a defining-occurrence spelled x becomes globally visible. An unattached identifier x will have as its defining-occurrence the defined-identifier of that declaration.

Because each global value is introduced in the form of a declaration, it is not possible to statically determine its definition. Its definition, which appears in the corresponding binding, remains hidden. In fact, a corresponding binding need not even exist if a translation that computes the value of a declaration can be produced in some way other than by compiling a binding, perhaps using an assembler.

Essentially, a global value is treated as a top-level parameter. Its definition (i.e., its implementation) can be changed as long as its definition's type is a subtype of the type indicated by its declaration. In this way, Acer supports information hiding and smart re-compilation. (Smart recompilation is supported by the fact that if a previously compiled binding is recompiled to produce a declaration with an equivalent type then bindings that reference that declaration need not be recompiled.)

A.9.1.1 Mutually dependent values

Mutually-dependent, global value-identifiers are simultaneously introduced by compiling a binding-list (see A.12)—such a binding-list may not contain variable-value-bindings. The effect of compiling a binding-list is similar to that of compiling just a fixed-value-binding. First, a translation used to compute the bindings' definitions, is produced and stored. And second, for each fixed-value-binding in the binding-list, a declaration of its value is produced, just a before, and stored. Storing these declarations has the effect of including the defined-identifiers in the global-name-layer. (The type-bindings do not become visible and can be used only in within the binding-list.)

For example, compiling the binding-list

{**let** x **be tuple** 1 y **end**;
 let y **be tuple** 1.0 x **end**;

produces the declarations

```
x : {let Unnamed be
        Tuple  : Integer;  : Tuple  :Real;  : Unnamed end end;
    Unnamed}
y : {let Unnamed be
        Tuple  : Real;  : Tuple  :Integer;  : Unnamed end end;
    Unnamed}
```

The defined-identifiers of the declarations (x and y) are included in the global name-layer.

A.9.2 Global type-identifiers

A global type-identifier is introduced by *storing* a type-binding. Storing a type-binding makes it permanently available and includes its defined-identifier in the global name-layer. (A fixed-value-binding or a valid binding-list can be stored too but this does not affect scope, as it does for a type-binding or for a fixed-value-declaration.)

A type-binding is not compiled to produced a declaration, as with a fixed-value-binding, because a type-declaration is useless unless it is quantified (see A.20.3), that is, unless values of that type are also declared. An unattached type-binding is not automatically included in the global name-layer because multiple unattached bindings for a given spelling can exist; the one that is stored is singled out to have its defined-identifier in the global name-layer.

Global type-identifiers are no more than a convenience since each applied-occurrence of such an identifier can be replaced by a definition-copy-at of its binding's definition at the location of the applied-occurrence. In other words, global types are not hidden but global values are.

This completes the description how identifier's are introduced with global scope.

A.10 Expression

How bindings and declarations are used to specify and make visible expressions has now been described. But how expressions themselves are formed has not yet been described. The remaining sections of this appendix do just that.

Recall that types and values are collectively known as expressions:

$$\langle \mathit{Expression} \rangle ::= \langle \mathit{Type} \rangle \;\|\; \langle \mathit{Value} \rangle$$

Type is defined as

$\langle$*Type*$\rangle$ ::=
$\langle$*AbstractType*$\rangle$ ‖ $\langle$*ConcreteType*$\rangle$ ‖ $\langle$*ReusedTypeIdentifier*$\rangle$ ‖
$\langle$*TypeBlock*$\rangle$ ‖ $\langle$*TypeDenoter*$\rangle$ ‖ $\langle$*TypeDesignation*$\rangle$ ‖
$\langle$*TypeOperator*$\rangle$

where abstract-type is defined as

$\langle$*AbstractType*$\rangle$ ::=
$\langle$*OperatorCall*$\rangle$ ‖ $\langle$*TypeIdentifier*$\rangle$ ‖ $\langle$*TypeSelection*$\rangle$

and a concrete-type is defined as

$\langle$*ConcreteType*$\rangle$ ::=
$\langle$*AnyType*$\rangle$ ‖ $\langle$*DynamicType*$\rangle$ ‖ $\langle$*EnumerationType*$\rangle$ ‖
$\langle$*FunctionType*$\rangle$ ‖ $\langle$*OptionType*$\rangle$ ‖ $\langle$*RecordType*$\rangle$ ‖
$\langle$*TupleType*$\rangle$ ‖ $\langle$*VariantType*$\rangle$

Value is defined as

$\langle$*Value*$\rangle$ ::=
$\langle$*Accumulation*$\rangle$ ‖ $\langle$*AndIfTest*$\rangle$ ‖ $\langle$*Assignment*$\rangle$ ‖ $\langle$*CodePatch*$\rangle$ ‖
$\langle$*CompoundValue*$\rangle$ ‖ $\langle$*Conditional*$\rangle$ ‖ $\langle$*Dereference*$\rangle$ ‖
$\langle$*DyadicMethodCall*$\rangle$ ‖ $\langle$*DynamicInspection*$\rangle$ ‖ $\langle$*FunctionCall*$\rangle$ ‖
$\langle$*Index*$\rangle$ ‖ $\langle$*IsTest*$\rangle$ ‖ $\langle$*IsNotTest*$\rangle$ ‖ $\langle$*Iteration*$\rangle$ ‖ $\langle$*KeepTrying*$\rangle$ ‖
$\langle$*Literal*$\rangle$ ‖ $\langle$*OrdCall*$\rangle$ ‖ $\langle$*OrIfTest*$\rangle$ ‖ $\langle$*PointerCall*$\rangle$ ‖
$\langle$*PrefixMethodCall*$\rangle$ ‖ $\langle$*Raise*$\rangle$ ‖ $\langle$*Try*$\rangle$ ‖ $\langle$*TryFinally*$\rangle$ ‖
$\langle$*UnaryMethodCall*$\rangle$ ‖ $\langle$*ValCall*$\rangle$ ‖ $\langle$*ValueBlock*$\rangle$ ‖ $\langle$*ValueDenoter*$\rangle$ ‖
$\langle$*ValueSelection*$\rangle$ ‖ $\langle$*VariantInspection*$\rangle$

where literal is defined as

$\langle$*Literal*$\rangle$ ::=
$\langle$*ArrayLiteral*$\rangle$ ‖ $\langle$*CharacterLiteral*$\rangle$ ‖ $\langle$*DynamicLiteral*$\rangle$ ‖
$\langle$*ExceptionLiteral*$\rangle$ ‖ $\langle$*FunctionLiteral*$\rangle$ ‖ $\langle$*IntegerLiteral*$\rangle$ ‖
$\langle$*LiteralSelection*$\rangle$ ‖ $\langle$*RealLiteral*$\rangle$ ‖ $\langle$*RecordLiteral*$\rangle$ ‖
$\langle$*ReferenceLiteral*$\rangle$ ‖ $\langle$*ReusedValueIdentifier*$\rangle$ ‖ $\langle$*StringLiteral*$\rangle$ ‖
$\langle$*TupleLiteral*$\rangle$ ‖ $\langle$*ValueIdentifier*$\rangle$ ‖ $\langle$*VariantLiteral*$\rangle$ ‖ $\langle$*VoidLiteral*$\rangle$

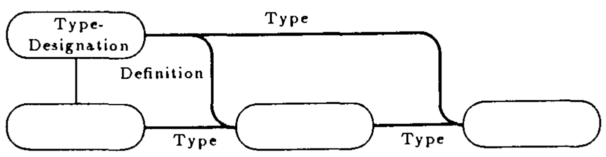
These are the various expression classes that are described in the sections that remain.

A.11 Type-designation

For denoting the type of an expression using just the expression itself, Acer provides a type-designation:

$\langle$*TypeDesignation*$\rangle$ ::= **TYPE** ($\langle$*:Expression*$\rangle$)

The context-dependent relations for a type-designation are illustrated as



Since a type-designation is a type, its expression is not evaluated. Subtyping is not defined on type-designations because a type-designation always denotes some other class of type.

A.12 Block

For introducing bindings local to an expression context, Acer provides blocks. A block is either a type-block or a value-block:

$$\langle \mathit{Block} \rangle ::= \langle \mathit{TypeBlock} \rangle \parallel \langle \mathit{ValueBlock} \rangle$$

A type-block consists of bindings and a body:

$$\langle \mathit{TypeBlock} \rangle ::= \langle \mathit{Bindings{:}BindingList} \rangle\ \langle \mathit{Body{:}Type} \rangle\ \}$$

as does a value-block:

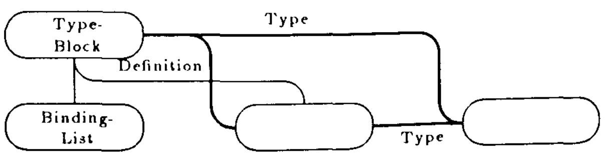
$$\langle \mathit{ValueBlock} \rangle ::= \langle \mathit{Bindings{:}BindingList} \rangle\ \langle \mathit{Body{:}Value} \rangle\ \}$$

And a binding-list is defined as

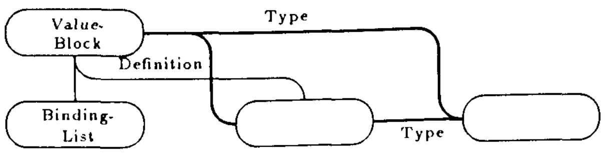
$$\langle \mathit{BindingList} \rangle ::= \{\ \{\![\ \langle \mathit{Binding} \rangle\ ]\!\}^{[\,;\,]}\ [\![\ ;\ ]\!]$$

The bindings of a type-block may not contain value-bindings.

The context-dependent relations for a type-block are illustrated as



Those for a value-block are the same:



A.12.1 Scope

The name-layer of the bindings of a block is searched either when lookup, starting in the body, reaches the block, or when lookup, starting in the binding-list, reaches the binding-list. In other words, the bindings in a binding-list are visible in that list regardless of whether the list is contained by a block; if it is contained by a block then the bindings are visible there too.

A.12.2 Evaluation

When a value-block is evaluated, its binding-list is evaluated first and then its body is evaluated in the context of those bindings. A value-block yields the value yielded by its body. As discussed in section A.7.2, even though a binding-list may be evaluated in any order that does not result in invalid delayed-occurrences, a left-to-right order must be one such correct order.

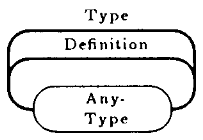
A type-block is not evaluated since it is a type. Subtype is not defined of type-blocks since a type-block always denotes some other class of type.

A.13 Any

For denoting the root type of the type lattice, Acer provides an any-type, which is a childless construction:

$\langle AnyType \rangle$ ::= **Any**

The context-dependent relations for an any-type are illustrated as



Every type is a subtype of every any-type.

A.14 Aggregate

For constructing aggregate data structures, Acer provides three closely related data structuring mechanisms: tuples, records, and dynamics. A tuple-value is constructed by a tuple-literal:

⟨*TupleLiteral*⟩ ::= **tuple** ⦃ ⟨*Argument*⟩ ⦄$^{[\,,\,]}$ **end**

which has a tuple-type:

⟨*TupleType*⟩ ::= **Tuple** ⦃ ⟨*Declaration*⟩ ⦄$^{[\,;\,]}$ **end**

A record-value is constructed by a record-literal:

⟨*RecordLiteral*⟩ ::= **record** ⦃ ⟨*Binding*⟩ ⦄$^{[\,,\,]}$ **end**

which has a record-type:

⟨*RecordType*⟩ ::= **Record** ⦃ ⟨*Declaration*⟩ ⦄$^{[\,;\,]}$ **end**

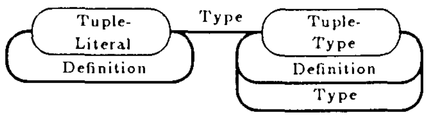
And a dynamic-value is constructed by a dynamic-literal:

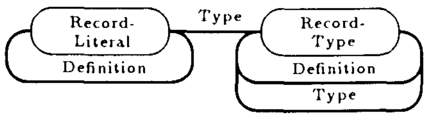
⟨*DynamicLiteral*⟩ ::= **dynamic** ⦃ ⟨*Argument*⟩ ⦄$^{[\,,\,]}$ **end**

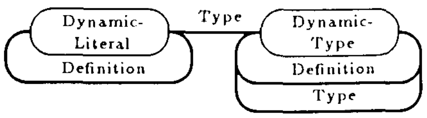
which as a dynamic-type:

⟨*DynamicType*⟩ ::= **Dynamic** ⦃ ⟨*Declaration*⟩ ⦄$^{[\,;\,]}$ **end**

The context-dependent relations for these constructs are illustrated as







For the non-empty case, each argument in the aggregate-literal derives a corresponding declaration (see A.8.2) in the aggregate-type.

Unlike tuples, which have ordered components, records have unordered components. For this reason, every component of a record must have a name. Hence, a record-literal may not contain anonymous variable-value-bindings and a record-type may not contain anonymous declarations.

Dynamics are used to provide for dynamic type checking; at run-time the first component of a dynamic (i.e., its tag) contains a representation of a type. Not every type can be used as the tag of a dynamic, however. Only a *closed-type* is permitted. A type is a closed-type if it is possible to produce a definition-copy-at of that type at the unattached empty node, that is, if it can be expressed with global scope. (Actually, abstract-types as closed-types are not completely safe because the definition (implementation) of an abstract-type can change from one run of a program to the next. But many basic types such as *Integer* (see A.26) are provided as abstract-types and certainly these abstract-types must be supported.)

Because the first component of a dynamic must be a closed-type, the first argument of a dynamic-literal must be a close-type or a type-binding that defines a closed-type. Also, the first declaration of a dynamic-type must be a type-declaration with a type that is a closed-type.

Operations for copying, reading and writing dynamics are provided by the module *dynamics*, which is visible as

```
dynamics :
  Tuple
    error : Exception ( Void)
    copy : Function ( : Dynamic Type :: Any;  : Type end)
               Dynamic Type :: Any; value : Type end
             end
    input : Function (filePath : String)
               Dynamic Type :: Any; value : Type end
              end
    output : Function (filePath : String
                        : Dynamic Type :: Any;  : Type end)
                Void
              end
  end
```

A.14.1 Scope

Each aggregate-literal has a name-layer containing the defined-identifiers of its bindings, which is searched when identifier-lookup, starting in the literal, reaches the literal. Similarly, each aggregate-type has a name-layer containing the defined-identifiers of its declarations, which is searched when identifier-lookup, starting in the type, reaches the type.

A.14.2 Evaluation

When an aggregate-literal is evaluated its arguments are evaluated in any order that does not access uninitialized components. Dependency analysis (see A.7.2) is used to validate the order in which arguments appear in the aggregate-literal.

An aggregate-literal is a constructor and hence its storage may be allocated well in advance of the evaluation of the literal. In this way, mutually-referential data structures can be constructed.

A.14.3 Subtype

An aggregate-type A is a subtype of an aggregate-type A' of the same class if for each declaration d' in A', there exists a *corresponding* declaration d in A such that:

- d and d' are the same class of declaration.
- The defined-identifiers of d and d' have the same spelling or at least one of them is empty.
- The definition-copy of the type of d with A' for A is either equivalent to the type of d', if d and d' are variable-value-declarations, or is a subtype of the type of d' otherwise.

Corresponding declarations are determined positionally for tuple- and dynamic-types, and by name for record-types. Note that a subtype A can have extra declarations, that is, declarations that do not correspond to any declaration in the supertype A'.

A.14.4 Component selection

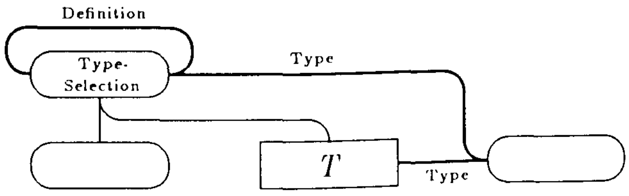
The components of an aggregate can be accessed using either type-selection:

⟨*TypeSelection*⟩ ::= ⟨*Base*:*Value*⟩ . ⟨*Selector*:*TypeIdentifier*⟩

or value-selection:

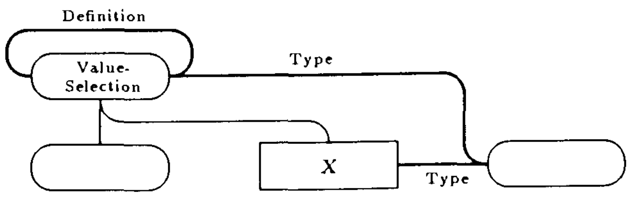
⟨*ValueSelection*⟩ ::= ⟨*Base*:*Value*⟩ . ⟨*Selector*:*ValueIdentifier*⟩

The context-dependent relations for a type-selection are illustrated as



The base of a type-selection must be a quantifier (see A.20.3) and must have a kind of class tuple-, record-, or dynamic-type.

Similarly, the context-dependent relations for a *concrete* value-selection are illustrated as



If the kind of the base of a value-selection is an aggregate-type containing a declaration of an identifier with the same spelling as the selector then the value-selection is called a *concrete* value-selection and the above relations apply. Otherwise, the value-selection is called an *abstract* value-selection and the relations described in section A.21.4 apply.

In general, the type of a type- or value-selection is not given directly by the type of its selector but is given by the definition-copy of the type of the selector with the following substitutions: each applied-occurrence of an identifier appearing in the name-layer of the kind of the base (an aggregate-type) is replaced by a type- or value-selection that has a denoter to the base as its base and has the applied-occurrence as its selector. This is called *resolving* the type because it adds to the type the information about its quantifier (see A.20.3). Thus for a tuple declared as

t : **Tuple** *Type* :: **Any**; *value* :: *Type* **end**

the type of *t.value* is *t.Type*, rather than simply *Type*.

A.14.4.1 Scope

When identifier-lookup begins at the selector of a type-selection, only the name-layer of the kind of the base of that type-selection is searched; lookup then terminates regardless of whether a defining-occurrence is found. Similarly, when identifier-lookup begins at the selector of a concrete value-selection, only the name-layer of the kind of the base of that value-selection is searched.

A.14.4.2 Evaluation

Since types are not evaluated, the base of a type-selection is not evaluated. When a concrete value-selection is evaluated, its base is evaluated and the named component is selected and yielded.

A.14.5 Dynamic-inspection

A dynamic-value can be *narrowed* to make its components accessible as the components of a tuple. This is done using a dynamic-inspection:

⟨*DynamicInspection*⟩ ::=
  inspect ⟨*Selector*: *Value*⟩ ⟨*Branches*: *TypeBranchList*⟩
   ⟦ **else** ⟨*DefaultBranch*: *Value*⟩ ⟧ **end**

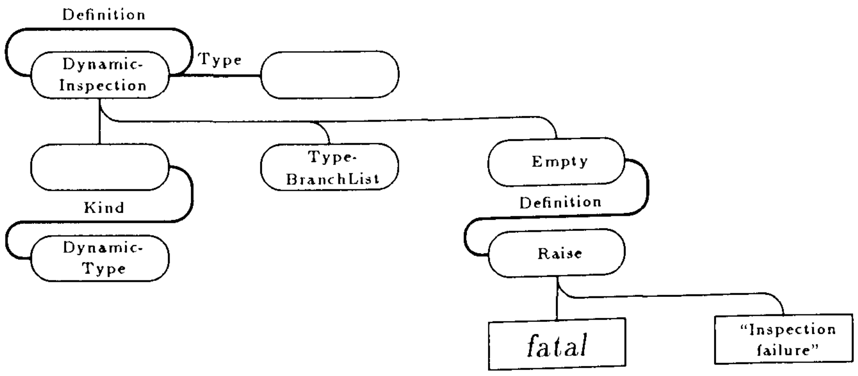
where type-branch-list is defined as

⟨*TypeBranchList*⟩ ::= **Then** ⦃ ⟨*TypeWhenBranch*⟩ ⦄$^{[\,;\,]}$

and type-when-branch is defined as

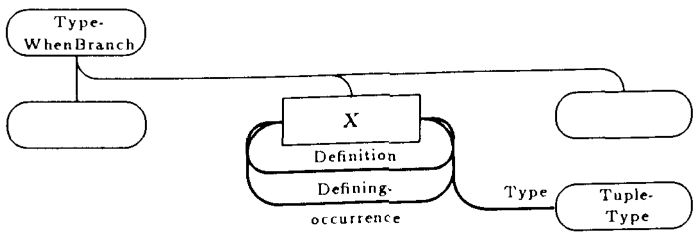
⟨*TypeWhenBranch*⟩ ::=
  when ⟨*Condition*: *Type*⟩ ⟦ **with** ⟨*DefinedIdentifier*: *ValueIdentifier*⟩ ⟧
   then ⟨*Consequent*: *Value*⟩

The context-dependent relations for a dynamic-inspection are illustrated as



The type of the consequent of each branch and the type of the default-branch must be such that one is a supertype of all the others; that type, the maximal type, is the inspection's type. The kind of the selector must be of class dynamic-type, and the condition of each branch must be a subtype of the type of the first declaration of that dynamic-type. No branch may have a condition that is a subtype of the condition of a branch that precedes it; this would be an unreachable branch.

The context-dependent relations for a type-when-branch are illustrated as



They are identical when the defined-identifier is empty, except that an empty node does not have a defining-occurrence. The type of the defined-identifier is derived by determining T, the kind of the selector of the immediately enclosing dynamic-inspection. From T, a dynamic-type, a tuple-type is derived by producing T', a definition-copy of T with no substitutions but with the following changes: T' is changed from a dynamic-type to a tuple-type; the initial declaration is changed from a type-declaration to an anonymous fixed-value-declaration of type any-type; and each applied-occurrence of the defined-identifier of the initial type-declaration is replaced by a type-denoter to the type-when-branch's condition.

For example, when inspecting a value of kind

Dynamic *Type* :: *OtherType*; *value*: *Type* **end**

using a type-when-branch of the form

when T **with** x **then** x.*value*

the defined-identifier x has type

Tuple :**Any**; *value*: $\{[T]\}$ **end**

where T is the condition of the type-when-branch. Notice how the *value* component of the dynamic can be accessed as the *value* component of x. Notice too that the initial anonymous component of the tuple contains the type-tag of the dynamic but because it has type any-type it is essentially useless. (If a metaprogramming system for Acer were implemented in Acer, the type for this first component could indicate that it is a node. It could then be manipulated using the metaprogramming system.)

A.14.5.1 Scope

The only effect a dynamic-inspection has on scope is when identifier-lookup reaches a type-when-branch from its consequent. The name-layer of the type-when-branch is then searched and lookup continues if a defining-occurrence is not found. The name-layer of a type-when-branch contains just its defined-identifier.

A.14.5.2 Evaluation

When a dynamic-inspection is evaluated, the selector is evaluated to yield a dynamic. The type-tag of this dynamic is selected to yield T, which is used to determine the appropriate branch to evaluate. If there is no branch with a condition that is a supertype of T then the default-branch is evaluated to yield the result—evaluating an empty default-branch raises an exception (see A.24.2). Otherwise, the first branch with a condition that is a supertype of T is evaluated to yield the result.

When the type-when-branch of a dynamic-inspection is evaluated, the dynamic yielded by the selector is bound to the defined-identifier and the consequent is evaluated to yield the result.

A.15 Argument-list and signature

Argument-lists and signatures are incomplete constructs that are used to specify parts of expressions and their types. Argument-list is defined as

$$\langle ArgumentList \rangle ::= (\ \{\!\!\{\ \langle Argument \rangle\ \}\!\!\}^{[\ ,\ ]}$$

and signature is defined as

$$\langle Signature \rangle ::= (\ \{\!\!\{\ \langle Declaration \rangle\ \}\!\!\}^{[\ ;\ ]}\ )$$

A signature can be derived from an argument-list in the same way that a tuple-type is derived from a tuple-literal (see A.14).

A.15.1 Scope

When identifier-lookup reaches an argument-list its name-layer is searched. Similarly, when identifier-lookup reaches a signature its name-layer is searched.

A.15.2 Subtype

The subtype relation for signatures, also called the ***subsignature*** relation, is the same as the subtype relation for tuple-types (see A.14.1).

A.15.3 Conformance

Given an argument-list a and a signature s we may ask whether a ***conforms*** to s. An argument-list a conforms to a signature s if they have the same length and if for each declaration d at position n in s, there exists a corresponding expression or binding b at position n in a such that:

- *b* and *d* have corresponding node classes, that is, if *d* is a type-declaration, *b* is a type or type-binding; if *d* is a fixed-value-declaration, *b* is a value or fixed-value-binding; and if *d* is a variable-value-declaration, *b* is a variable-value-binding.
- The defined-identifiers of *d* and *b* (if it has one) have the same spelling or at least one of them is empty.
- The definition-copy of the type of *d* with *a* for *s* is either equivalent to the type of *b*, if *d* is a variable-value-declaration, or is a supertype of the type of *b* otherwise.

A.15.4 Evaluation

A signature is like a type and is not evaluated. An argument-list is like a binding-list and is evaluated in any order that does not access uninitialized bindings—a left-to-right order must be a correct evaluation order (see A.7.2).

A.16 Enumeration and option

For representing a finite number of unstructured data elements, Acer provides enumerations and options. An enumeration- or option-value is denoted using a literal-selection:

⟨*LiteralSelection*⟩ ::= ⟨*Base*:*Type*⟩ . ⟨*Selector*:*ValueIdentifier*⟩

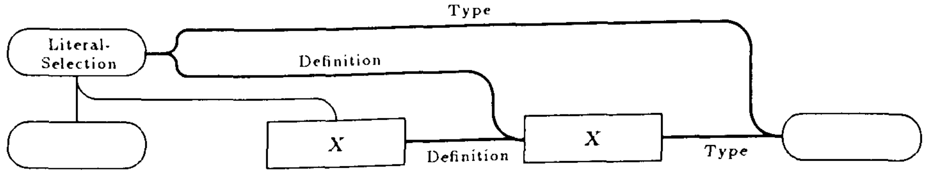
which has either an enumeration-type:

⟨*EnumerationType*⟩ ::= **Enumeration** ⦃ ⟨*ValueIdentifier*⟩ ⦄$^{[\,,\,]}$ **end**

or an option type:

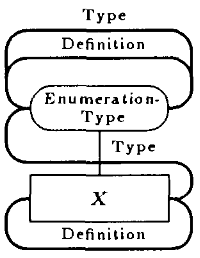
⟨*OptionType*⟩ ::= **Option** ⦃ ⟨*ValueIdentifier*⟩ ⦄$^{[\,,\,]}$ **end**

The context-dependent relations for literal-selection are illustrated as



The base of a literal-selection must denote an option- or enumeration-type that contains the defining-occurrence of its selector.

The context-dependent relations for enumeration-type are illustrated as



The same relations apply for option-type.

A.16.1 Scope

An enumeration- or option-type does not affect identifier-lookup but it does define a name-layer consisting of its identifiers.

A literal-selection affects identifier-lookup as follows. When lookup reaches a literal-selection from its selector, the name-layer of the denotation of the base is searched and lookup terminates, successful or not.

A.16.2 Subtype

An enumeration-type E is a subtype of an enumeration-type E' if for each identifier i at the nth position in E, there is a corresponding identifier i' with the same spelling at the nth position in E'.

An option-type O is a subtype of an option-type O' if for each identifier i in O there is an identifier i' with the same spelling in O'.

The ordering of identifiers is significant for enumeration-types but not for option-types.

A.16.3 Evaluation

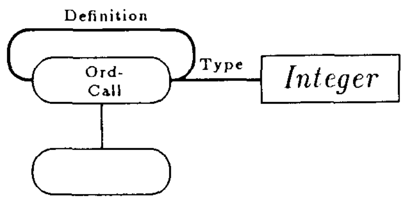
When a literal-selection is evaluated, its denoted value is yielded. A literal-selection is a constant.

A.16.4 Conversion

For determining the ordinal position of an enumeration-value, Acer provides an ord-call:

$\langle \mathit{OrdCall} \rangle ::=$ **ord** ($\langle \mathit{Base}\!:\!\mathit{Value} \rangle$)

The context-dependent relations for ord-call are illustrated as



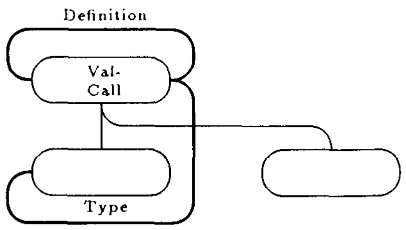
The base must be an enumeration-value.

When an ord-call is evaluated, its base is evaluated first. The value yielded by the base is converted to an *Integer*. The ordinal position of the first enumeration-value is 0 and successive values have increasing ordinal positions.

For determining an enumeration-value given an enumeration-type and an ordinal position, Acer provides a val-call:

⟨*ValCall*⟩ ::= **val** (⟨*BaseType*:*Type*⟩ ⟦ , ⟧ ⟨*Ordinal*:*Value*⟩)

The context-dependent relations for a val-call are illustrated as



The base-type must denote an enumeration-type and the ordinal must be an *Integer*.

When a val-call is evaluated, the ordinal is evaluated to yield an *Integer*, which is converted to an enumeration-value and yielded. If the ordinal is out of range, the following exception is raised:

raise *fatal* **with** `"Val-call out of range."` **end**

A.17 Variant

For constructing varying data structures, Acer provides variants. A variant-value is constructed by a variant-literal:

⟨*VariantLiteral*⟩ ::=
  variant ⟨*Tag*:*ValueIdentifier*⟩ **of** ⟨*BaseType*:*Type*⟩
    with ⟨*Arguments*:*ArgumentList*⟩)

which has a variant-type:

⟨*VariantType*⟩ ::=
  Variant ⟨*Tag*:*Type*⟩ ⟨*Variants*: *VariantList*⟩
   ⟦ **else** ⟨*Default*:*Signature*⟩ ⟧ **end**

where variant-list is defined as

⟨*VariantList*⟩ ::= **of** ⦃ ⟨*VariantElement*⟩ ⦄$^{[\,;\,]}$

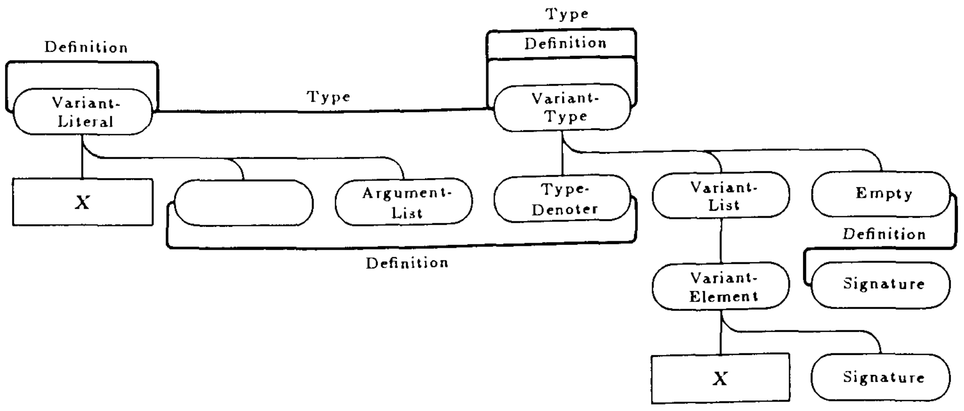
variant-element is defined as

⟨*VariantElement*⟩ ::=
  ⟨*Condition*: *WhenCondition*⟩ **then** ⟨*Consequent*:*Signature*⟩

and when-condition is defined as

⟨*WhenCondition*⟩ ::= **when** ⦃ ⟨*Value*⟩ ⦄$^{[\,,\,]}$

The context-dependent relations for these constructs are illustrated as



The base-type of a variant-literal must denote an enumeration- or option-type containing a defining-occurrence for the tag. The type of a variant-literal is a variant-type derived as follows: the tag is the literal's base-type, the default is empty, and the variant-list has a single variant-element, its condition is a copy of the literal's tag and its consequent is the signature derived from the literal's argument-list (see A.15).

The tag of a variant-type must denote an enumeration- or option-type T. Each value appearing in the when-condition of a variant-element must be a value-identifier defined in T. The when-conditions of a variant-type may contain at most one applied-occurrence of each identifier in T. Hence, for each identifier i in T, there is a corresponding signature in the variant-type—either the signature of the variant-element containing an applied-occurrence of i or the default signature.

A.17.1 Scope

When identifier-lookup reaches a variant-literal from its tag, the name-layer of the denotation of its base-type is searched. Lookup then terminates, successful or not.

When identifier-lookup reaches the when-condition of a variant-element, the name-layer of the denotation of the tag of the immediately enclosing variant-type is searched. Lookup then terminates.

A.17.2 Evaluation

When a variant-literal is evaluated, its arguments are evaluated in any order that does not access uninitialized components (see A.7.2). A variant-literal is a constructor and hence its storage may be allocated well in advance of the evaluation of the literal.

A.17.3 Subtype

A variant-type V with a tag denoting T is a subtype of a variant-type V' with a tag denoting T' if the following hold:

- T is a subtype of T'.
- And for each identifier i in T and i' in T', where i and i' have the same spelling, the signature s associated with i in V is a subsignature (see A.15.2) of the signature s' associated with i' in V'.

A.17.4 Variant-inspection

A variant-value must be *narrowed* to make its components accessible. This is done using a variant-inspection:

```
⟨VariantInspection⟩ ::=
    inspect ⟨Selector: Value⟩ ⟨Branches: ValueBranchList⟩
      [[ else ⟨DefaultBranch: Value⟩ ]] end
```

where value-branch-list is defined as

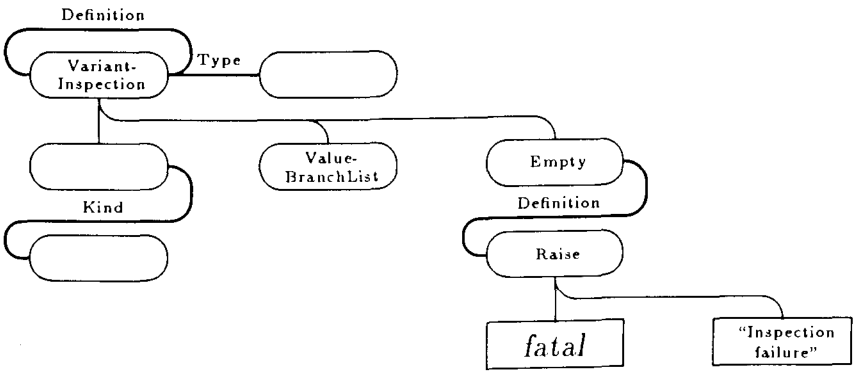
```
⟨ValueBranchList⟩ ::= then {[ ⟨ValueWhenBranch⟩ ]}^[ ; ]
```

and value-when-branch is defined as

⟨*ValueWhenBranch*⟩ ::=
⟨*Condition*: *WhenCondition*⟩
[[**with** ⟨*DefinedIdentifier*: *ValueIdentifier*⟩]] **then**
⟨*Consequent*: *Value*⟩

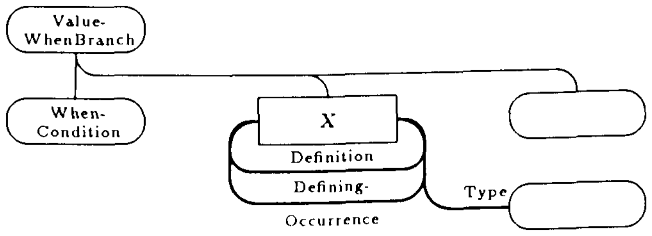
A variant-inspection can also be used as a multi-way branch based on an enumeration- or option-value.

The context-dependent relations for a variant-inspection are illustrated as



The type of the consequent of each branch and the type of the default-branch must be such that one is a supertype of all the others; that type, the maximal-type, is the inspection's type. The kind of the selector must denote either an enumeration- or option-type E or a variant-type with a tag that denotes an enumeration- or option-type E. The conditions of a variant-inspection may contain at most one applied-occurrence of each identifier in E.

The context-dependent relations for a value-when-branch in the branches of a variant-inspection are illustrated as



They are identical for an empty defined-identifier, except that an empty node does not have a defining-occurrence. The type of the defined-identifier depends on the kind of the selector T.

If T is a variant-type then the type of the defined-identifier of a value-when-branch is derived follows. The set of signatures associated with the identifiers in the condition must be such that one signature s is a supersignature (see A.15.2) of all the others. From s a tuple-type is derived by producing T', a definition-copy of s with no substitutions but with the following changes: T' is changed from a signature to a tuple-type; and an anonymous fixed-value-declaration, with a type given by the variant-type's tag, is inserted as the first element. For example, the type of *v* in

> **when** *id* **with** *v* **then** *v.x*

where the selector has type

```
Variant {[Enumeration id end]} of
  when id then (T :: Any; x : T)
 end
```

is

> **Tuple** :{[**Enumeration** *id* **end**]}; T :: **Any**; x : T **end**

If T is an enumeration- or option-type then the type of the defined-identifier is derived as follows. For an enumeration-type, the type of the defined-identifier is a copy of T that includes only those identifiers with ordinal positions no bigger than that of the identifier with the largest ordinal position in the condition. For an option-type, the type of the defined-identifier is an option-type that includes only those identifiers appearing in the condition.

A.17.4.1 Scope

When identifier-lookup reaches a when-condition of a variant-inspection, one of two things happens. If T, the kind of the selector, is a variant-type then the name-layer of the tag of that variant-type is searched and lookup terminates, successful or not. Otherwise, T must denote an enumeration- or option-type; the name-layer of T is then searched and lookup terminates, successful or not.

A.17.4.2 Evaluation

When a variant-inspection is evaluated the selector is evaluated first. If it is a variant then its tag is selected to yield i. Otherwise, it is an enumeration or option which directly yields i. From i the branch with an applied-occurrence of i is determined. If no applied-occurrence appears, the default-branch is evaluated to yield the result—evaluating an empty default-branch raises an exception (see A.24.2). Otherwise, the appropriate branch is evaluated to yield the result.

When the value-when-branch of a variant-inspection is evaluated, the value yielded by the selector is bound to the defined-identifier and the consequent is evaluated to yield the result.

A.18 Function

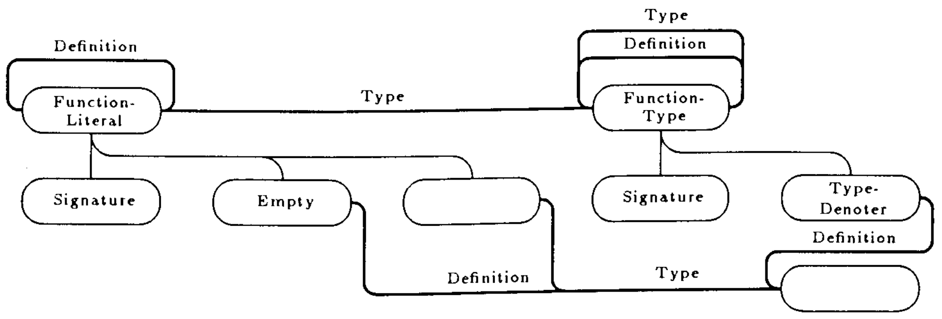
For expressing a value in terms of a parameterized expression, Acer provides functions. A function-value is constructed by a function-literal:

⟨*FunctionLiteral*⟩ ::=
    function ⟨:*Signature*⟩ ⟦ : ⟨*ResultType*:*Type*⟩ ⟧ ⟨*Body*:*Value*⟩ **end**

which has a function-type:

⟨*FunctionType*⟩ ::= **Function** ⟨:*Signature*⟩ ⟨*ResultType*:*Type*⟩ **end**

The context-dependent relations for these constructs are illustrated as



The type of a function-literal's body must be a subtype of the result-type. The signature of a function-literal or function-type may not contain variable-value-declarations.

A.18.1 Scope

When identifier-lookup reaches a function-literal from either its result-type or its body, the name-layer of the signature is searched. Similarly, when identifier-lookup reaches a function-type from its result-type, the name-layer of the signature is searched.

A.18.2 Subtype

A function-type F is a subtype of a function-type F' if:

- The signature of F' is a subtype of the signature (see A.15.2) of F and they have the same length.
- The definition-copy of the result-type of F with the signature of F' for the signature of F is a subtype of the result-type of F'.

A.18.3 Evaluation

When a function-literal is evaluated, a closure of the values used in the literal is computed. That closure, along with the instructions for evaluating the literal's body, represent the function-value.

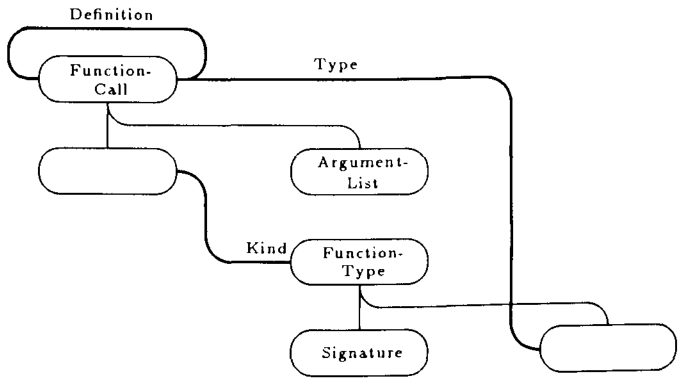
A function-literal is a constructor and hence its storage may be allocated well in advance of the evaluation of the literal. In this way recursive functions can be constructed.

A.18.4 Function-call

For invoking a function, Acer provides a function-call:

$\langle$*FunctionCall*$\rangle$ ::= $\langle$*Function*: *Value*$\rangle$ $\langle$*Arguments*:*ArgumentList*$\rangle$)

The context-dependent relations for a function-call are illustrated as



In general, the type of a function-call is derived by producing the definition-copy of the function-type's result-type with the argument-list for the signature. The argument-list must conform (see A.15.3) to the signature.

A.18.4.1 Scope

When identifier-lookup reaches a function-call from its function, the name-layer of the argument-list is searched.

A.18.4.2 Evaluation

When a function-call is evaluated, the arguments are evaluated (see A.7.2) and then the function is evaluated to yield the result. Evaluating a function involves evaluating its body in the context of its closure and the arguments supplied by the call.

A.19 Type-operator

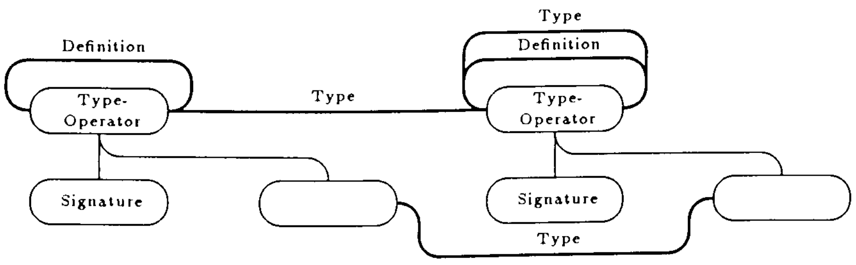
For expressing a type in terms of a parameterized expression, Acer provides type-operators:

⟨*TypeOperator*⟩ ::= **Operator** ⟨:*Signature*⟩ ⟨*Body*:*Type*⟩ **end**

There are no values with type-operators as their types.

The context-dependent relations for a type-operator depend on whether it is *abstract* or *concrete*. A type-operator is a concrete type-operator if its body denotes either a concrete-type or a concrete type-operator. Conversely, a type-operator is an abstract type-operator if its body denotes either an abstract-type or an abstract type-operator. The distinction between the two lies in the fact that a concrete type-operator is a kind but an abstract type-operator is not.

The context-dependent relations for type-operators are illustrated as



An abstract type-operator has a concrete type-operator as its type and a concrete type-operator is its own type. The definition of a type-operator is *Error* if the type-operator appears in the denotation of any node in its body, or if it appears in the denotation of any node in those denotations, and so on. In other words, the definition of a type-operator is *Error* if it is recursive. The signature of a type-operator may not contain value-declarations.

A.19.1 Scope

When identifier-lookup reaches a type-operator from its body, the name-layer of the signature is searched.

A.19.2 Subtype

A type-operator T is a subtype of a type-operator T' if:

- The signature of T' is a subsignature (see A.15.2) of the signature of T and they have the same length.

- The definition-copy of the body of T with the signature of T' for the signature of T is a subtype of the body of T'.

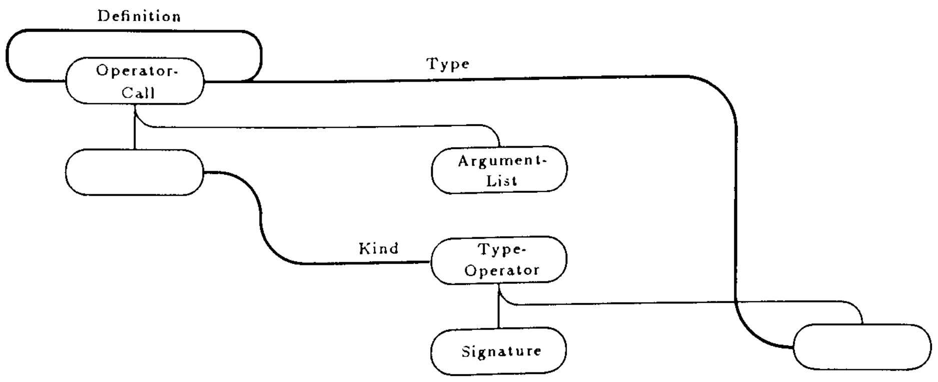
A.19.3 Operator-call

For invoking a type-operator, Acer provides an operator-call:

⟨*OperatorCall*⟩ ::= ⟨*Operator:Type*⟩ ⟨*Arguments:ArgumentList*⟩)

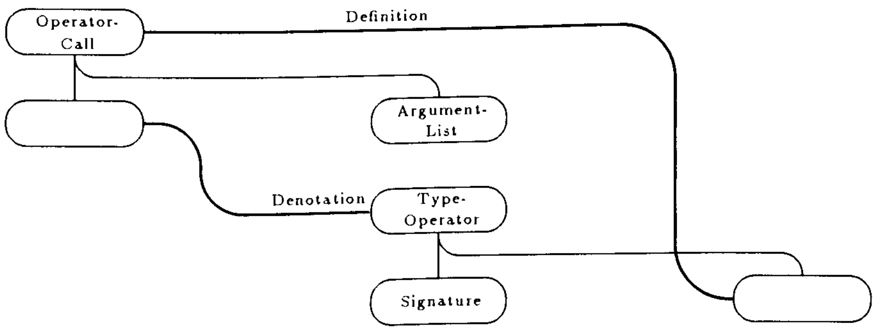
The kind of the operator must denote a type-operator and the argument-list must conform to the signature of that type-operator.

The context-dependent relations for an operator-call are illustrated as



In general, the type is derived by producing a definition-copy of the body of the type-operator with the argument-list of the operator-call for the signature of the type-operator.

An operator-call is its own definition only if its operator does not denote a type-operator, that is, if the operator denotes an abstract-type of kind type-operator. If the operator denotes a type-operator, the definition relation is instead illustrated as



In general, the definition is derived from the type-operator in the same way that the type is derived above. Because an operator-call with an operator that denotes an abstract-type is its own definition, subtype is defined on it as we shall see in the following section.

A.20 Abstract-type

Every type T denoting an abstract-type A (a type-identifier, type-selection, or operator-call) has an *abstract-name* I, an *abstract-base* B, and a *quantifier* Q. Each will be considered in the subsections that follow.

A.20.1 Abstract-name

The abstract-name I of an abstract-type A is determined as follows:

- If A is a type-identifier, I is A itself.
- If A is a type-selection, I is the defining-occurrence of the selector of A.
- If A is an operator-call, I is the abstract-name of the operator of A.

An abstract-name is a type-identifier that occurs as the defined-identifier of a type-declaration in a signature or aggregate-type.

A.20.2 Abstract-base

The abstract-base B of an abstract-type A is determined as follows:

- If A is a type-identifier or type-selection, B is A itself.
- If A is an operator-call, B is the abstract-base of the operator of A.

An abstract-base is a type-identifier or type-selection.

A.20.3 Quantifier

The quantifier Q of an abstract-type A is determined as follows:

- If A is a type-identifier, Q is the parent of the parent of A.
- If A is a type-selection, Q is the base of A.
- If A is an operator-call, Q is the quantifier of the operator of A.

A quantifier is a signature or aggregate-type, if B is a type-identifier; it is a value with an aggregate-type as its kind, if B is a type-selection.

To prevent an abstract-type from being used outside the scope of its quantifier, and to ensure that the quantifier denotes the same value throughout its scope, Acer enforces the following restriction. For every value x with a type T that denotes an abstract-type, the quantifier Q of T must be a *valid quantifier* with respect to x. A quantifier Q is valid with respect to x if:

- Q is a signature or aggregate-type,
- Q is a fixed-identifier that does not have a defining-occurrence enclosed by x,
- Q is a concrete value-selection with a selector that is a fixed-identifier and a base that is a valid quantifier with respect to x,
- or Q is a value-denoter with a definition that is a valid quantifier with respect to x.

If the quantifier of the type of x is not valid with respect to x then x is said to have an *invalid abstract-type*. For example, the above restriction dictates that the selection

> **tuple let** T **be** *Integer*; **let** x: T **be** 0 **end**.x

has an invalid abstract-type

> {(**tuple let** T **be** *Integer*; **let** x: T **be** 0 **end**)}.T

because a tuple-literal is not a valid quantifier. Also, the block

> {**let** t **be tuple let** T **be** *Integer*; **let** x: T **be** 0 **end**; t.x}

has an invalid abstract-type:

> {(t)}.T

because its contains the defining-occurrence of its type's quantifier.

Quantifiers are used to define subtyping, and to define Acer's various forms of method-calling. Subtyping will be considered first.

A.20.4 Subtype

Subtype for abstract-types is defined as follows. A type-identifier or type-selection A is a subtype of a type-identifier of type-selection A' if

- the abstract-names of A and A' have the same spelling,

- and the quantifier of A is *equivalent* to the quantifier of A'.

An operator-call A is a subtype of an operator call A' if

- A and A' have the same number of arguments,
- each argument at position n in A is equivalent to the corresponding argument at position n in A',
- and the operator of A is a subtype of the operator of A'.

Equivalence for quantifiers is defined as follows. A quantifier denoting Q is equivalent to a quantifier denoting Q' if

- Q and Q' are the same node,
- or Q and Q' are value-selections with bases that are equivalent and selectors that are spelled the same.

Notice that equivalence is defined on values as well as types. (The notion of static value-equivalence could be extended. For example, two different integer-literal nodes could be considered equivalent if they have the same spelling. This does not affect subtyping however and will not be considered further.)

A.21 Method

As a convenient short-hand notation for function-calls Acer provides various classes of method-call. To support method-calls, Acer defines the notion of a *method* as follows. Every value x with a type that denotes an abstract-type A potentially has methods, that is, values associated with the quantifier Q of A.

If Q is a signature or aggregate-type, the names of the methods of x appear in the same name-layer as the abstract-name I of A. For example, in the function-literal

function (T :: **Any**; x : T; m: **Function** (: T) T **end**) m(x) **end**

x has a method named m.

If Q is a value, the names of the methods of x appear in the name-layer of the kind of Q. For example, if t is globally declared as

t: **Tuple** T : **Any**; x : T; m: **Function** (: T) T **end end** then the value-selection $t.x$ has a method named m.

Hence, a value x has a method named m if an identifier with that spelling can be found in the quantifier of its type. This method is denoted in context either as the identifier m, if Q is a signature or aggregate-type, or as the value-selection $Q.m$, if Q is a value. This identifier or value-selection is called the ***denoted method*** and is used to define the behavior of method-calls.

A.21.1 Prefix-method-call

The most general form of method-call is the prefix-method-call, which consists of a method-name and arguments:

⟨*PrefixMethodCall*⟩ ::=
⟨*MethodName*:*ValueIdentifier*⟩ . ⟨*Arguments*:*ArgumentList*⟩)

Special scope rules apply for method-names.

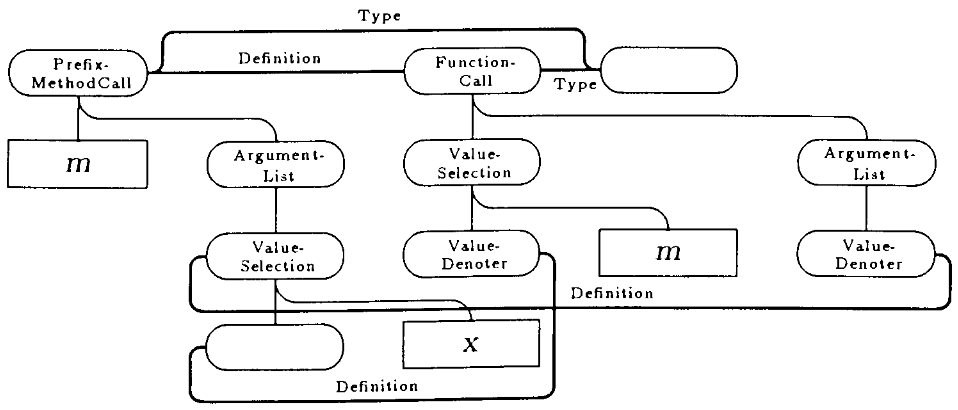
A.21.1.1 Scope

When identifier-lookup reaches a prefix-method-call from its method-name, a ***method-search*** is initiated. A method-search, given a series of arguments and the spelling s, considers each successive value-argument x with an abstract-type A having a quantifier Q. If Q is a signature or aggregate-type, the name-layer of Q is searched, and if Q is a value, the name-layer of the kind of Q is searched. Hence, the first argument x with a method named s determines the defining-occurrence of the method-name. This argument is called the ***method-owner***.

For the method-name of a prefix-method-call, method-search is applied to the argument-list. Lookup terminates after the method-search, regardless of whether a method is found. The unattached empty node is returned if no method is found.

A.21.1.2 Type and definition

The context-dependent relations for a prefix-method-call are illustrated as



In general, the definition of a prefix-method-call is a function-call derived as follows. The argument-list of the function-call is a definition-copy of that of the method-call, and the function is the denoted method of the method-owner. Also, additional arguments are inserted if the type of the method-owner denotes an operator-call A. In this case, definition-copies of the arguments of A are inserted as the first arguments of the function-call. Furthermore, if the operator of A denotes an operator-call, the arguments of that operator-call are also copied and inserted, and so on until the abstract-base is reached.

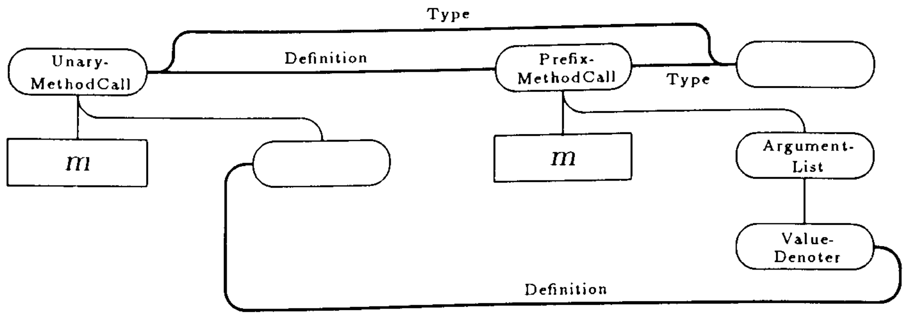
Each of Acer's remaining method-call classes is defined in terms of a prefix-method-call.

A.21.2 Unary-method-call

A unary-method-call consists of a method-name and an operand:

⟨*UnaryMethodCall*⟩ ::=
  { ⟨*MethodName*: *ValueIdentifier*⟩ ⟨*Operand*: *Value*⟩ }

The context-dependent relations for a unary-method-call are illustrated as



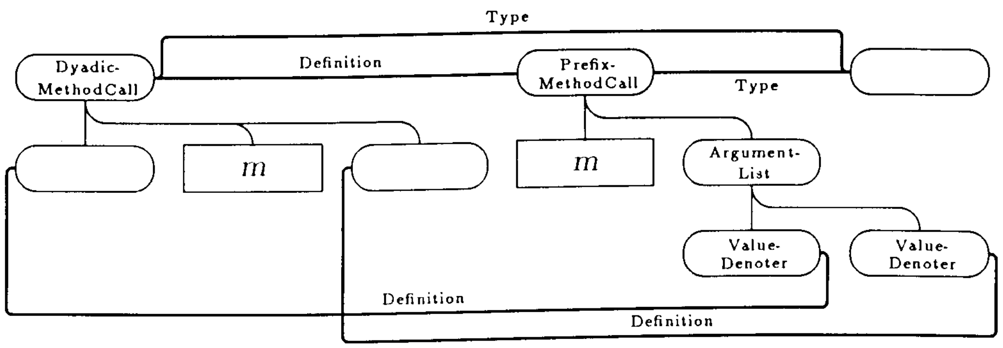
The defining-occurrence of the method-name is determined just it is for the prefix-method-call.

A.21.3 Dyadic-method-call

A dyadic-method-call consists of a first-operand, a method-name, and a second-operand:

> ⟨*DyadicMethodCall*⟩ ::=
> { ⟨*FirstOperand*: *Value*⟩ ⟨*MethodName*: *ValueIdentifier*⟩
> ⟨*SecondOperand*: *Value*⟩ }

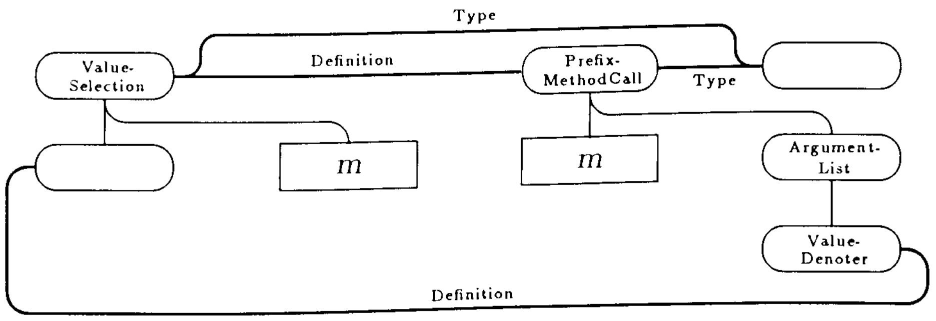
The context-dependent relations for a dyadic-method-call are illustrated as



The defining-occurrence of the method-name is determined just it is for the prefix-method-call.

A.21.4 Abstract value-selection

If the kind of the base of a value-selection x is not a aggregate-type T, or the selector does not occur in the name-layer of T then x is called an abstract value-selection. In this case the context-dependent relations are illustrated as



An abstract value-selection is treated as a unary-method-call. However, if the kind of the definition is a reference- or pointer-type (see A.30) then the definition is instead the dereference of the above definition. The defining-occurrence of the selector of an abstract value-selection is determined as it is for the prefix-method-call.

A.21.5 Index

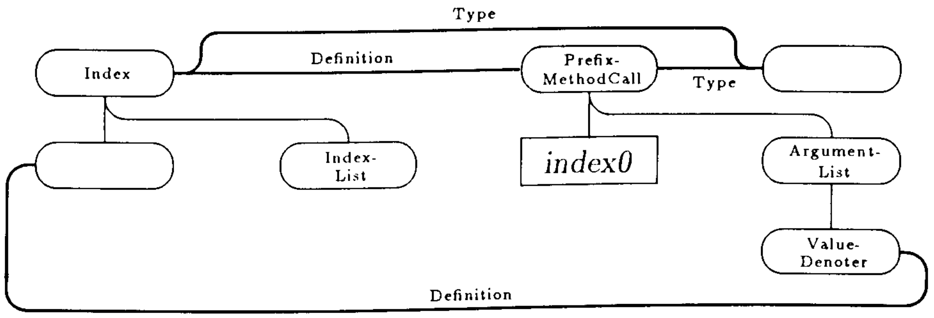
An index consists of a base and indices:

⟨*Index*⟩ ::= ⟨*Base*: *Value*⟩ ⟨*Indices*: *IndexList*⟩]

where index-list is defined as

⟨*IndexList*⟩ ::= [⦃ ⟨*Value*⟩ ⦄$^{[\,,\,]}$

The context-dependent relations for an index are illustrated as



The method-name of an index's definition is of the form *indexn*, where n is the length of the indices of x. If the index-list is not empty then definition-copies of the indices are inserted after the initial argument in the prefix-method-call's argument-list. Just as for the definition of an abstract value-selection, if the kind of the definition of an index is a reference- or pointer-type then the definition is instead a dereference of the above definition.

This completes the description of Acer's method-calls.

A.22 Void

For representing the type of expressions evaluated for side-effect, Acer provides the type *Void*, which is visible in the global name-layer as

let *Void* **be** *void.Type*

The module *void* is visible as

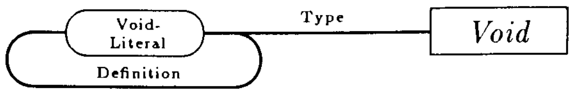
```
void:
  Tuple
    Type::Any
  end
```

Any type equivalent to *Void* shall be called a void-type.

There is only a single value of type *Void*. It can be denoted by a void-literal:

⟨*VoidLiteral*⟩ ::= {}

The context-dependent relations for a void-literal are illustrated as



A.23 Boolean

For representing Boolean truth values, Acer provides the type *Boolean*, which is visible in the global name-layer as

```
let Boolean be boolean.Type
```

The module *boolean* is visible as

```
boolean:
  Tuple
    Type :: Enumeration false, true end
    not : Function ( : Type) Type end
    and : Function ( : Type;  : Type) Type end
    or : Function ( : Type;  : Type) Type end
    all : Function ( : Type) Accumulator(Type, Type) end
    some : Function ( : Type) Accumulator(Type, Type) end
    < : Function ( : Type;  : Type) Type end
    <= : Function ( : Type;  : Type) Type end
    > : Function ( : Type;  : Type) Type end
    >= : Function ( : Type;  : Type) Type end
    = : Function ( : Type;  : Type) Type end
    # : Function ( : Type;  : Type) Type end
  end
```

The *Boolean* truth values are denoted as *boolean.false* and *boolean.true*. However, the global declarations

false : *Boolean*

and

true : *Boolean*

are provided as synonyms.

Acer has several constructs involving Booleans.

A.23.1 Identity

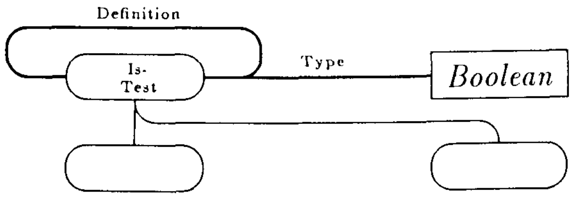
For testing identity (bitwise equality), Acer provides an is-test, consisting of a first-operand and a second-operand:

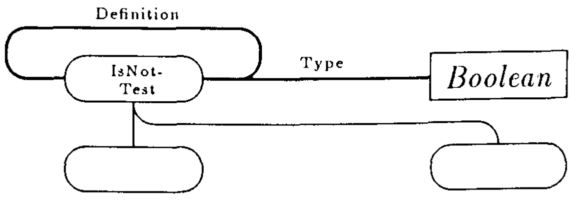
$\langle \mathit{IsTest} \rangle$ ::= { $\langle \mathit{FirstOperand}\colon \mathit{Value} \rangle$ **is** $\langle \mathit{SecondOperand}\colon \mathit{Value} \rangle$ }

as well as an is-not-test:

$\langle \mathit{IsNotTest} \rangle$ ::=
{ $\langle \mathit{FirstOperand}\colon \mathit{Value} \rangle$ **isnot** $\langle \mathit{SecondOperand}\colon \mathit{Value} \rangle$ }

The context-dependent relations for these constructs are illustrated as





The types of the two operands must be such that one is a subtype of the other.

A.23.2 Evaluation

When an is-test or an is-not-test is evaluated, its operands are evaluated in arbitrary order. The values yielded are tested for identity to yield the appropriate result.

A.23.3 Conditional

For supporting branching on *Boolean* values, Acer provides a conditional, which consists of branches and an optional default-branch:

$\langle \mathit{Conditional} \rangle$ ::=
$\langle \mathit{Branches}\colon \mathit{ConditionalBranchList} \rangle$
$[\![$ **else** $\langle \mathit{DefaultBranch}\colon \mathit{Value} \rangle$ $]\!]$ **end**

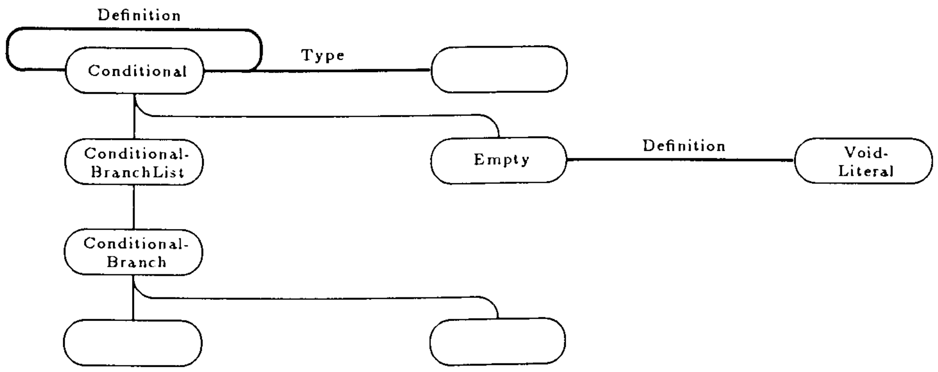
where conditional-branch-list is defined as

$\langle$*ConditionalBranchList*$\rangle$ ::= **if** ⦃ $\langle$*ConditionalBranch*$\rangle$ ⦄$^{\textbf{elsif}}$

and conditional-branch is defined as

$\langle$*ConditionalBranch*$\rangle$ ::= $\langle$*Condition*: *Value*$\rangle$ **then** $\langle$*Consequent*: *Value*$\rangle$

The context-dependent relations for a conditional are illustrated as



The type of the condition of a conditional-branch must be *Boolean.* The types of the consequents of the branches and the type of the default-branch must be such that one is a supertype of all the others; that type, the maximal type, is the conditional's type.

A.23.3.1 Evaluation

When a conditional is evaluated, each successive condition of the branches is evaluated until for some branch *b* the condition yields *true*. The consequent of *b* is then evaluated to yield the result. If no condition yields *true*, the default-branch is evaluated to yield the result.

A.23.4 Shortcircuit evaluation

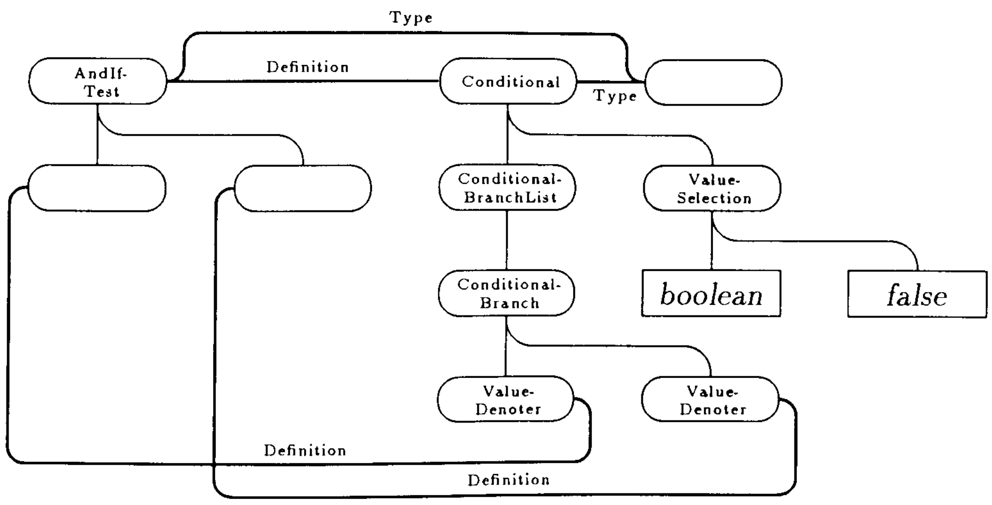
For supporting short-circuit Boolean evaluation, Acer provides the and-if-test and the or-if-test. An and-if-test consists of a first-operand and a second-operand:

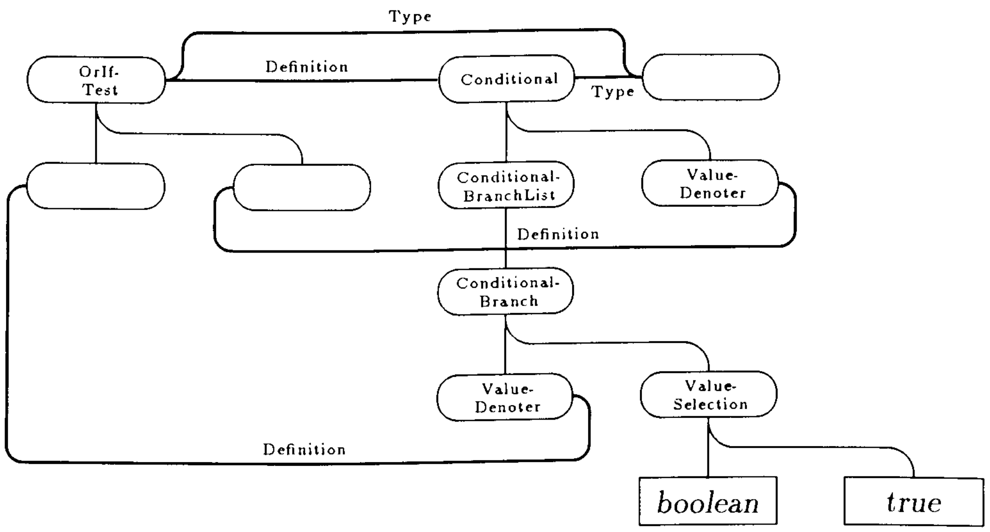
$\langle$*AndIfTest*$\rangle$ ::=
{ $\langle$*FirstOperand*: *Value*$\rangle$ **andif** $\langle$*SecondOperand*: *Value*$\rangle$ }

as does and or-if-test:

$\langle$*OrIfTest*$\rangle$ ::= { $\langle$*FirstOperand*: *Value*$\rangle$ **orif** $\langle$*SecondOperand*: *Value*$\rangle$ }

The context-dependent relations for these constructs are illustrated as





Thus each of these constructs is defined as the appropriate conditional, which evaluates the second argument only when necessary.

Note that {*all true*} ([x, *y*]) is equivalent to {*x* **andif** *y*} and that {*some true*} ([x, *y*]) is equivalent to {*x* **orif** *y*}. These accumulators can thus be used as iterated versions of **andif** and **orif**.

A.24 Exception

For interrupting normal sequential evaluation, Acer provides exceptions. To support exceptions, the type *Exception* is visible in the global-name-layer as

let *Exception* **be** *exceptions.Type* and the module *exceptions* is visible as

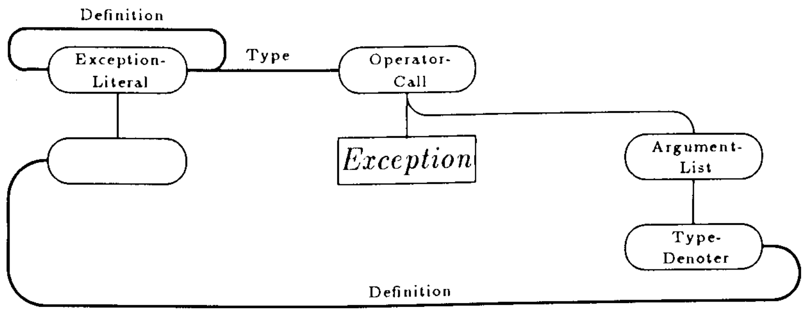
> *exceptions*:
> **Tuple**
> *Type* :: **Operator** (*BaseType* :: **Any**) **Any end**
> *Raise* :: **Any**
> *Error* :: **Any**
> **end**

Any type equivalent to *Exception* (*T*) is called an exception-type with base-type *T*.

An exception-value is constructed by an exception-literal:

⟨*ExceptionLiteral*⟩ ::= **exception** (⟨*BaseType*:*Type*⟩)

The context-dependent relations for an exception-literal are illustrated as



A.24.0.1 Evaluation

When an exception-literal is evaluated, the constructed exception-value is yielded. An exception-literal is a constructor and hence its storage may be allocated well in advance of the evaluation of the literal.

A.24.1 Standard exceptions

Acer provides two standard exceptions, *exit* and *fatal*, which are visible in the global name-layer as

> *exit* : *Exception* (*Void*)
> *fatal* : *Exception* (*String*)

The *exit* exception is provided simply as a convenience but the *fatal* expection is special; when a program is terminated by raising *fatal*, the associated-value of *fatal*, which must be a *String* (see A.29), is printed as an error message.

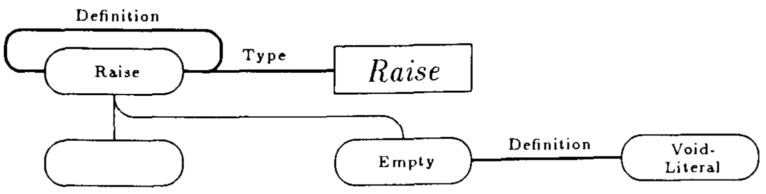
Consider now how exceptions are raised.

A.24.2 Raise

An exception is raised by a raise, which consists an exception-value and an optional associated-value:

> ⟨*Raise*⟩ ::=
> **raise** ⟨*ExceptionValue*: *Value*⟩
> ⟦ **with** ⟨*AssociatedValue*: *Value*⟩ ⟧ **end**

The context-dependent relations for a raise are illustrated as



The type of the exception-value must denote an exception-type and the type of the associated-value must be a subtype of the base-type.

A.24.2.1 Subtype

The type *Raise* is visible in the global name-layer as

> **let** *Raise* **be** *exceptions.Raise*

It is a special type because it cannot cause type conflicts. It is considered equivalent to every other type. However, when determining the maximal type of a set of types, as for the branches of a conditional, it is considered the maximal type only if every type denotes *Raise*. In other words, an expression has type *Raise* only if it *always* raises an exception.

A.24.2.2 Evaluation

A raise never yields a value. When a raise is evaluated, the exception-value is evaluated first and then normal evaluation is suspended. Then the exception, with its associated-value, propagates back through the dynamic evaluation chain until it is trapped by a suitable handler.

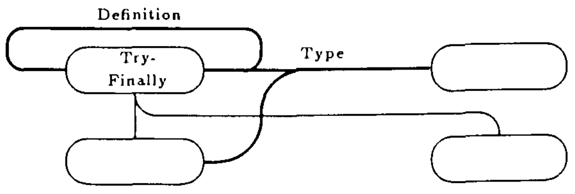
Acer provides three constructs for trapping exceptions, namely try-finally, try, and keep-trying. Each will be considered in turn.

A.24.3 Try-finally

For specifying evaluations to be carried out regardless of whether an exception is raised, Acer provides the try-finally, which consists of a body and a final-action:

⟨*TryFinally*⟩ ::= **try** ⟨*Body*: *Value*⟩ **finally** ⟨*FinalAction*: *Value*⟩ **end**

The context-dependent relations for a try-finally are illustrated as



A.24.3.1 Evaluation

When a try-finally is evaluated, its body is evaluated first and its final-action is evaluated next, even if the body raises an exception. Then, if neither the body nor the final action raise an exception, the try-finally yields the value yielded by its body. Otherwise, either the body, the final-action, or both raise an exception. If the body raises an exception but the final-action does not, the try-finally raises the exception raised by its body. Otherwise, if both the body and the final-action raise exception, or just the final-action raises an exception, the try-finally raises the exception raised by the final-action. In any case, the value yielded by the final-action is discarded.

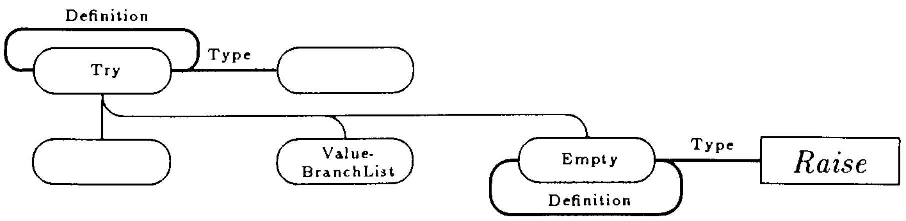
A.24.4 Try

For trapping exceptions raised by the evaluation of an expression, Acer provides the try, which consists of a body, branches, and an optional default-branch:

⟨*Try*⟩ ::=
  try ⟨*Body*: *Value*⟩ ⟨*Branches*: *ValueBranchList*⟩
    ⟦ **else** ⟨*DefaultBranch*: *Value*⟩ ⟧ **end**

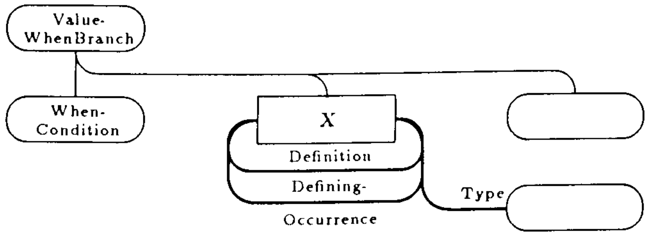
A value-branch-list has the same form as for a variant-inspection (see A.17.4) but the semantics of a value-when-branch in a try is different from the semantics of a value-when-branch in a variant-inspection.

The context-dependent relations for a try are illustrated as



The types of the consequents of the branches, the type of the body, and the type of the default-branch must be such that one is a supertype of all the others; that type, the maximal type, is the try's type.

The context-dependent relations for a value-when-branch in the branches of a try are illustrated as



They are identical for an empty defined-identifier, except that an empty node does not have a defining-occurrence. The maximal type of the types of values in the condition must denote an exception-type; the base-type of that type is the type of the defined-identifier.

A.24.4.1 Evaluation

When a try is evaluated, the body is evaluated first. If the body does not raise an exception, the value yielded by the body is yielded. Otherwise, the raised exception is compared with the value yielded by each successive value in the conditions of the branches. If a match is found, the associated-value of the exception is bound to the defined-identifier of the corresponding branch and the consequent is evaluated to yield the result of the try. If not match is found, the default-branch is evaluated to yield the result of the try. Evaluating an empty default-branch reraises the exception raised by the body.

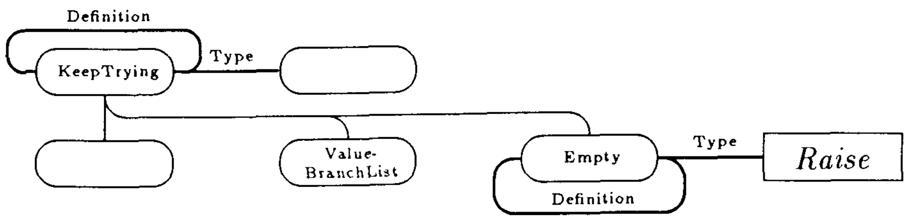
A.24.5 Keep-trying

For specifying evaluations to be carried out repeatedly, Acer provides the keep-trying, which consists of a body, branches, and an optional default-branch:

⟨*KeepTrying*⟩ ::=
  keep trying ⟨*Body*: *Value*⟩ ⟨*Branches*: *ValueBranchList*⟩
    [[**else** ⟨*DefaultBranch*: *Value*⟩]] **end**

A keep-trying is Acer's only looping construct, other than high-level iteration.

The context-dependent relations for a keep-trying are illustrated as



The types of the consequents of the branches, and the type of the default-branch must be such that one is a supertype of all the others; that type is the keep-trying's type.

The context-dependent relations for a value-when-branch in the branches of a keep-trying are the same as those for a try.

A.24.5.1 Evaluation

When a keep-trying is evaluated, the body is evaluated repeatedly until an exception is raised. Thus the value(s) yielded by the body are discarded each time. When an exception is finally raised, it is compared with the value yielded by each successive value in the conditions of the branches. If a match is found, the associated-value of the exception is bound to the defined-identifier of the corresponding branch and the consequent is evaluated to yield the result of the keep-trying. If not match is found, the default-branch is evaluated to yield the result of the keep-trying. Evaluating an empty default-branch reraises the exception raised by the body.

A.25 Error

For representing the denotations of erroneous types and values, Acer provides the special type *Error* and the special value *error*. The type *Error* is visible in the global name-layer as

let *Error* **be** *exceptions.Error*

(The module *exceptions* was introduced in section A.24.) The value *error*, the one value of type *Error*, is visible in the global name-layer as

error : *Error*

A.25.1 Subtype

The type *Error*, like the type *Raise*, it is considered equivalent to every other type. However, when determining the maximal type of a set of types, as for the branches of a conditional, it is considered the maximal type if any type denotes *Error*. (This is unlike the type *Raise*, which is considered the maximal type only if every type denotes *Raise*.)

A.25.2 Evaluation

When a value that denotes *error* is evaluated, the *fatal* exception is raised as

raise *fatal* **with** "error" **end**

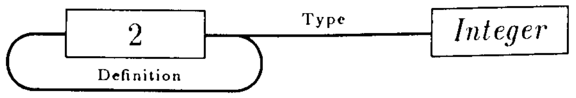
Thus although it is invalid for a program to contain expressions that denote *error* or *Error*, a program containing such expressions can nevertheless be evaluated, although likely not with the intended effect.

A.26 Integer

For representing integers, Acer provides integer-literals:

$\langle IntegerLiteral\rangle ::= \langle\#Digit\rangle \; \{\!\![\; \langle\#Digit\rangle \;]\!\!\}$

The context-dependent relations for an integer-literal are illustrated as



The type *Integer* is visible as

let *Integer* **be** *integer.Type*

and the module *integer* is visible as

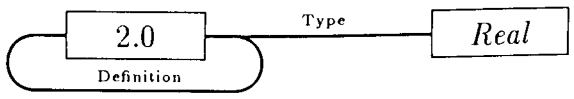
```
integer:
  Tuple
    Type :: Any
    error : Exception (Void)
    ~ : Function ( : Type) Type end
    abs : Function ( : Type) Type end
    + : Function ( : Type; : Type) Type end
    - : Function ( : Type; : Type) Type end
    * : Function ( : Type; : Type) Type end
    mod : Function ( : Type; : Type) Type end
    div : Function ( : Type; : Type) Type end
    < : Function ( : Type; : Type) Boolean end
    <= : Function ( : Type; : Type) Boolean end
    > : Function ( : Type; : Type) Boolean end
    >= : Function ( : Type; : Type) Boolean end
    = : Function ( : Type; : Type) Boolean end
    # : Function ( : Type; : Type) Boolean end
   end
```

A.27 Real

For representing reals, Acer provides real-literals:

⟨*RealLiteral*⟩ ::=
  ⟦ - ⟧ ⟨*#Digit*⟩ ⦃ ⟨*#Digit*⟩ ⦄ . ⟨*#Digit*⟩ ⦃ ⟨*#Digit*⟩ ⦄
    ⟦ ⦅ **e** ‖ **E** ⦆ ⟦ - ⟧ ⟨*#Digit*⟩ ⦃ ⟨*#Digit*⟩ ⦄ ⟧

The context-dependent relations for a real-literal are illustrated as



The type *Real* is visible as

let *Real* **be** *real.Type*

and the module *real* is visible as

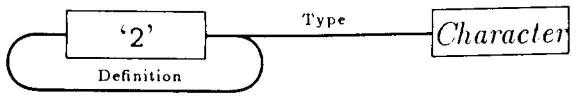
```
real:
  Tuple
    Type :: Any
    error : Exception ( Void)
    ~ : Function ( : Type) Type end
    abs : Function ( : Type) Type end
    + : Function ( : Type;  : Type) Type end
    - : Function ( : Type;  : Type) Type end
    * : Function ( : Type;  : Type) Type end
    / : Function ( : Type;  : Type) Type end
    ^ : Function ( : Type;  : Type) Type end
    < : Function ( : Type;  : Type) Boolean end
    <= : Function ( : Type;  : Type) Boolean end
    > : Function ( : Type;  : Type) Boolean end
    >= : Function ( : Type;  : Type) Boolean end
    = : Function ( : Type;  : Type) Boolean end
    # : Function ( : Type;  : Type) Boolean end
  end
```

A.28 Character

For representing characters, Acer provides character-literals:

⟨*CharacterLiteral*⟩ ::=
  ' ⟦ ⟨*#LowercaseLetter*⟩ || ⟨*#UppercaseLetter*⟩ || ⟨*#SymbolicLetter*⟩ ||
    ⟨*#Punctuation*⟩ || ⟨*#Digit*⟩ || ⟨*#Space*⟩ || ⟨*#DoubleQuote*⟩ ||
    ⟨*#Other*⟩ ⟧ '

The context-dependent relations for a character-literal are illustrated as



The type *Character* is visible as

let *Character* **be** *character.Type*

and the module *character* is visible as

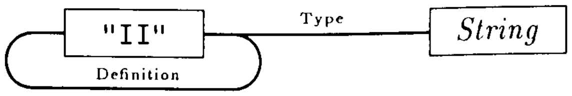
character :
  Tuple
    Type :: **Any**
    < : **Function** (: *Type*; : *Type*) *Boolean* **end**
    <= : **Function** (: *Type*; : *Type*) *Boolean* **end**
    > : **Function** (: *Type*; : *Type*) *Boolean* **end**
    >= : **Function** (: *Type*; : *Type*) *Boolean* **end**
    = : **Function** (: *Type*; : *Type*) *Boolean* **end**
    # : **Function** (: *Type*; : *Type*) *Boolean* **end**
  end

A.29 String

For representing strings, Acer provides string-literals:

⟨*StringLiteral*⟩ ::=
  " ⦃ ⟨*#LowercaseLetter*⟩ ‖ ⟨*#UppercaseLetter*⟩ ‖ ⟨*#SymbolicLetter*⟩ ‖
    ⟨*#Punctuation*⟩ ‖ ⟨*#Digit*⟩ ‖ ⟨*#Space*⟩ ‖ ⟨*#Other*⟩ ⦄ "

The context-dependent relations for a string-literal are illustrated as



The type *String* is visible as

let *String* **be** *string.Type*

and the module *string* is visible as

string :
  Tuple
    Type :: **Any**
    error : *Exception* (*Void*)
    length : **Function** (: *BaseType*) *Integer* **end**
    index1 : **Function** (: *Type*; : *Integer*) *Character* **end**
    substring : **Function** (: *Type*; : *Integer*; : *Integer*) *Type* **end**
    + : **Function** (: *Type*; : *Type*) *Type* **end**
    < : **Function** (: *Type*; : *Type*) *Boolean* **end**
    <= : **Function** (: *Type*; : *Type*) *Boolean* **end**
    > : **Function** (: *Type*; : *Type*) *Boolean* **end**
    >= : **Function** (: *Type*; : *Type*) *Boolean* **end**
    = : **Function** (: *Type*; : *Type*) *Boolean* **end**
    # : **Function** (: *Type*; : *Type*) *Boolean* **end**
  end

A.30 Locations and side-effects

For modeling updatable locations, Acer provides two similar notions, references and pointers. Whereas references provide high-level support by modeling updatable locations in terms of fetch and store functions, pointers provide low-level support by modeling updatable locations in terms of memory addressing.

A.30.1 Reference

For determining a reference to a location, Acer provides a reference-literal:

⟨*ReferenceLiteral*⟩ ::= **reference** (⟨*Base*: *Value*⟩)

which has a reference-type. The type *Reference* is visible as

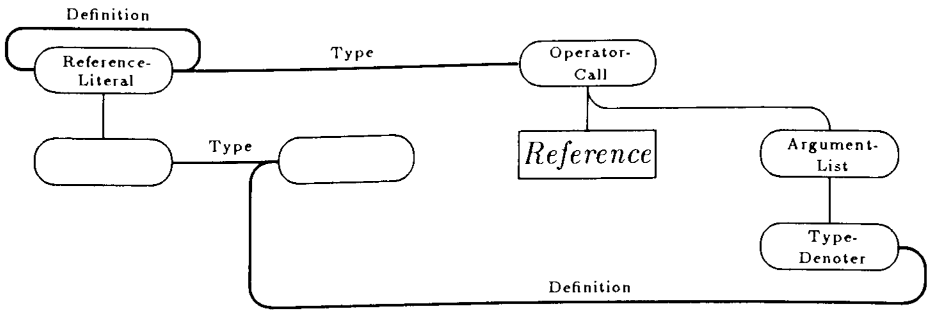
let *Reference* **be** *references.Type*

and the module *references* is visible as

```
references:
  Tuple
    Type :: Operator (BaseType :: Any) Any end
    new :Function (BaseType :: Any;  : BaseType)
           Type (BaseType)
          end
    create:
      Function
         (BaseType :: Any
          fetch: Function () BaseType end
          store: Function ( : BaseType) Void end)
        Type (BaseType)
       end
   end
```

Any type equivalent to *Reference*(*T*) is called a reference-type with base-type *T*.

The context-dependent relations for a reference-literal are illustrated as



The base of a reference-literal must be either a dereference, a variable-identifier, a value-selection with a variable-identifier as its selector, an index that denotes a dereference, an abstract value-selection that denotes a dereference, or a denoter with one of the above as its definition. With the enforcement of these restrictions, the location referenced by the base of a reference-literal can always be determined.

A.30.1.1 Evaluation

When a reference-literal is evaluated, its base is evaluated to yield the referenced location. A reference-literal is a constructor and hence its storage may be allocated well in advance of the evaluation of the literal. Hence, recursive references can be constructed.

A reference-literal yields a reference to an existing location but there are two ways in which references to a new location can be constructed. First, a new reference can be constructed by calling *references.new* with a type and a value of that type. And second, a new reference can be constructed by calling *references.create* with a fetch function and a store function.

Regardless of how a reference is constructed, a reference consists of two functions: a fetch function, which is called to yield the referenced value; and a store function, which is called to update the referenced value. When a reference is constructed by a reference-literal or by *references.new*, the fetch and store functions are created implicitly but when a reference is constructed by references.create, the fetch and store functions are provided explicitly.

A.30.2 Pointer

For determining a pointer to a location, Acer provides a pointer-call:

⟨*PointerCall*⟩ ::= **pointer** (⟨*Base*: *Value*⟩)

which has a pointer-type. The type *Pointer* is visible as

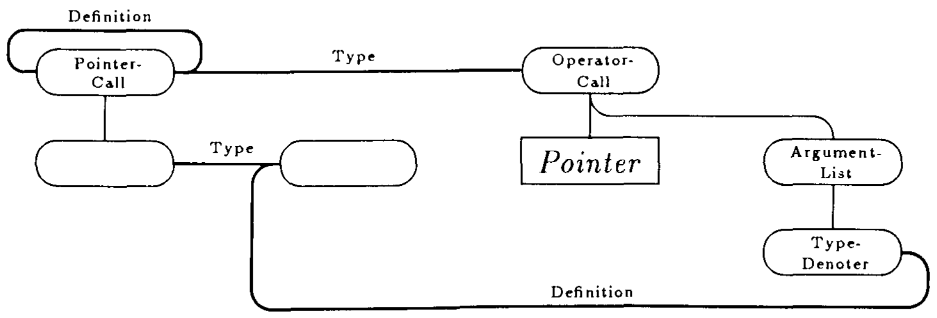
let *Pointer* **be** *pointers.Type*

and the module *pointers* is visible as

```
pointers:
  Tuple
    Type :: Operator (BaseType :: Any) Any end
    new :Function (BaseType :: Any;  : BaseType)
              Type (BaseType)
            end
  end
```

Any type equivalent to *Pointer* (T) is called a pointer-type with base-type T.

The context-dependent relations for a pointer-call are illustrated as



The base of a pointer-call must be either a dereference with a base of type pointer-type, a variable-identifier, a concrete value-selection with a variable-identifier as its selector, an index that denotes a dereference with a base of type pointer-type, an abstract value-selection that denotes a dereference with a base of type pointer-type, or a value-denoter with one of the above as its definition. The base of a pointer call may also be either a fixed value-identifier, a concrete value-selection with a fixed value-identifier as its selector, or a denoter with one of these as its definition, but in this case it is considered unsafe.

A.30.2.1 Evaluation

When a pointer-call is evaluated, the base is evaluated to yield its address. This address is yielded as the result of the call. Unlike a reference-literal, a pointer-call is not a constructor.

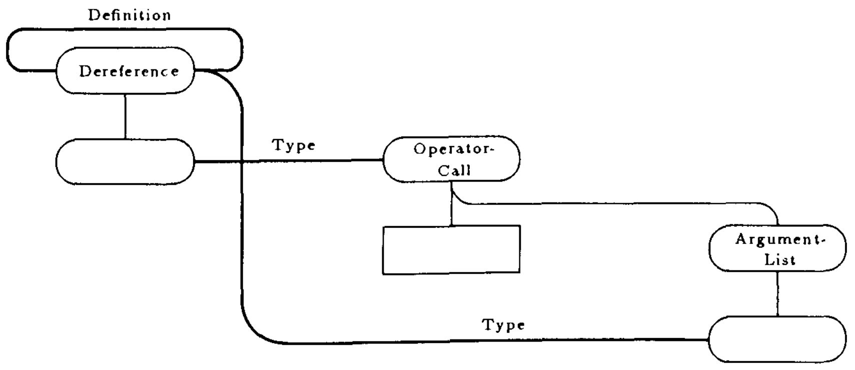
A pointer-call yields a pointer to an existing location but there is also a way in which a pointer to a new location can be constructed, by calling *pointers.new* with a type and a value of that type.

A.30.3 Dereference

For accessing references and pointers, Acer provides the dereference:

$\langle Dereference \rangle ::= \langle Base\colon Value \rangle$ @

The context-dependent relations for a dereference are illustrated as



The type of the base of a dereference must denote a pointer- or reference-type; the base-type of that type is the type of the dereference.

A.30.3.1 Evaluation

When a dereference is evaluated one of two things happens, depending on whether the base is reference or pointer. If the base is a pointer, the target location is accessed to yield the result of the dereference. Otherwise, if the base is a reference, the reference's fetch function is called to yield the result of the dereference.

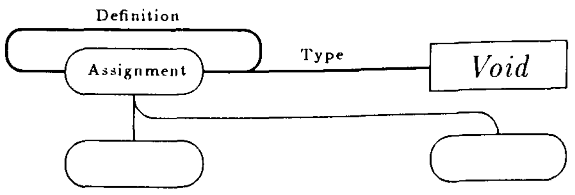
A dereference can also be used as the destination of an assignment, in which case, as we shall see, its evaluation is carried out quite differently.

A.30.4 Assignment

For modifying the contents of a location, Acer provides the assignment, which consists of a destination and a source:

⟨*Assignment*⟩ ::= { ⟨*Destination*: *Value*⟩ **becomes** ⟨*Source*: *Value*⟩ }

The context-dependent relations for an assignment are illustrated as



The type of the source must be a subtype of the type of the destination. The destination of an assignment, like the base of a reference-literal, must be a dereference, a variable-identifier, a concrete value-selection with a variable-identifier as its selector, an index that denotes a dereference, an abstract value-selection that denotes a dereference, or a value-denoter with one of the above as its definition. In addition, the destination of an assignment, like the base of a pointer-call, may also be a fixed-identifier, a concrete-value-selection with a selector that is a fixed-identifier, or a value-denoter with one of these as its definition, but in this case the assignment is unsafe.

A.30.4.1 Evaluation

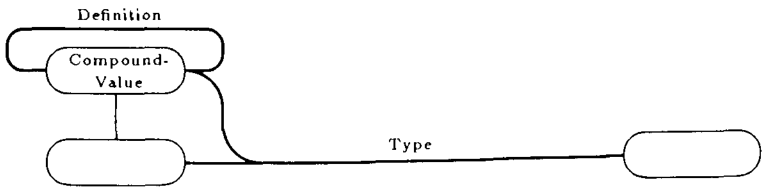
When an assignment is evaluated, the source and destination are evaluated in arbitrary order; the destination is evaluated to yield a reference or pointer to a location and the source is evaluated to yield the value to be stored at that location. When the destination yields a reference (i.e., when the destination denotes a dereferenced reference), the reference's store function is called with the value yielded by the source. Otherwise, the destination yields a pointer and the location addressed by that pointer is updated with the value yielded by the source.

A.30.5 Compound-value

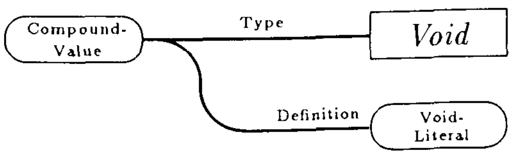
For specifying sequential evaluation, Acer provides a compound-value:

$\langle \textit{CompoundValue} \rangle ::= \textbf{begin} \ \{\!\!\{ \ \langle \textit{Value} \rangle \ \}\!\!\}^{\textbf{[;]}} \ \textbf{end}$

The context-dependent relations for a compound-value are illustrated as



The type of the last value is the type of the compound-value. The type of an empty compound-value is *Void*:



A.30.5.1 Evaluation

When a compound-value is evaluated, each successive value is evaluated in order. The value yielded by the last value is yielded by the compound-value. An empty compound-value yields the void-value.

A.31 Array

For constructing data structures with an arbitrary number of components, Acer provides the array-literal:

$\langle$*ArrayLiteral*$\rangle$ ::= $\langle$*Elements*:*ArrayList*$\rangle$ [[**of** $\langle$*BaseType*:*Type*$\rangle$]] **end**

where array-list is defined as:

$\langle$*ArrayList*$\rangle$ ::= **array** {[$\langle$*Value*$\rangle$]}$^{[\,,\,]}$

An array-literal has an array-type. The type *Array* is visible as

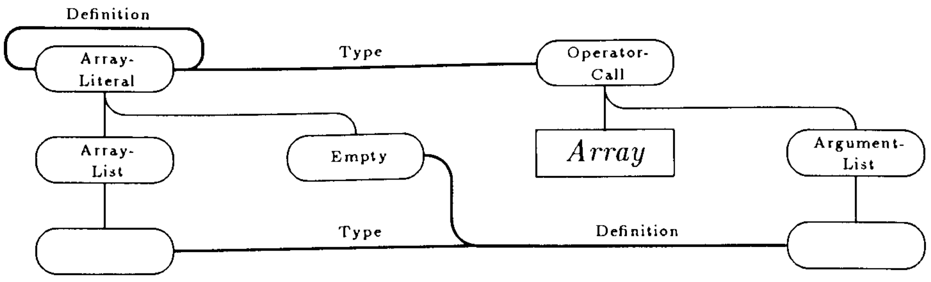
let *Array* **be** *arrays.Type*

and the module *arrays* is visible as

```
arrays:
  Tuple
    Type :: Operator (BaseType :: Any) Any end
    error : Exception (Void)
    length : Function (BaseType :: Any;  : Type (BaseType))
                Integer
              end
    index1 :
      Function (BaseType :: Any;  : Type (BaseType);  : Integer)
        Pointer (BaseType)
      end
    new : Function (BaseType :: Any;  : BaseType;  : Integer)
             Type (BaseType)
           end
  end
```

Any type equivalent to *Array*(T) shall be called an array-type with base-type T.

The context-dependent relations for an array-literal are illustrated as



The type of an array-literal is an array-type with the literal's base-type as its base-type. The types of the values in the elements must be such that one is a supertype of all the others; that type, the maximal type, is the definition of the empty base-type. If the base-type is not empty, the base-type must be a supertype of the maximal type. If the array-list is empty the maximal type is *Void*.

A.31.1 Evaluation

When an array-literal is evaluated, its elements are evaluated in arbitrary order. An array-literal is a constructor and hence its storage may be allocated well in advance of the evaluation of the literal.

A.32 Iterator and accumulator

For supporting high-level iteration and accumulation, Acer provides the notion of iterators and accumulators. An iterator is a sequence producer and an accumulator is a sequence consumer.

A.32.1 Iterator

To support iterators the type *Iterator* is visible as

```
let Iterator be
   Operator (BaseType :: Any)
      Tuple
         done : Exception (Void)
         produce : Function () BaseType end
         terminate : Function () Void end
      end
   end
```

Thus, an iterator is any value x with a type that is a subtype of

Iterator (T)

where T is the base-type.

An iterator produces a sequence of values through repeated calls to its *produce* function. When the iterator's sequence of values is exhausted, a call to *produce* raises its *done* exception. The iterator is then terminated with a call to its *terminate* function. The iterator may be terminated before the sequence is exhausted by calling its *terminate* function early.

A.32.2 Accumulator

To support accumulators the type *Accumulator* is visible as

```
let Accumulator be
  Operator (BaseType :: Any;  ResultType :: Any)
    Tuple
      done : Exception ( Void)
      consume : Function ( : BaseType) Void end
      terminate : Function () ResultType end
     end
   end
```

Thus, an accumulator is any value x with a type that is a subtype of

Accumulator$(B,\ R)$

where B is the base-type and R is the result-type. An accumulator consumes a sequence of values through repeated calls to its *consume* function. When the accumulation is complete, a call to *consume* will raise its *done* exception. An accumulator raises *done* only if it does not wish to consume more values. An accumulator is terminated by a call to its *terminate* function, which yields the result of the accumulation. The *terminate* function can be called either because the accumulator has raised *done* or because the sequence of values has been exhausted.

A declaration of the accumulator *discard* is visible in the global name-layer. Its corresponding binding is defined as

```
let discard :Accumulator(Any, Void) be
  tuple
    exception ( Void)
    function ( : Any) {} end
    function () {} end
   end
```

A.32.3 Iteration

Iterators and accumulators are used in iterations, which consist of iterators, an optional filter, an optional accumulator, and a body:

```
⟨Iteration⟩ ::=
   ⟨Iterators: IteratorList⟩ [[ andif ⟨Filter: Value⟩ ]] do
     [[ ⟨Accumulator: Value⟩ ]] ⟨Body: Value⟩ end
```

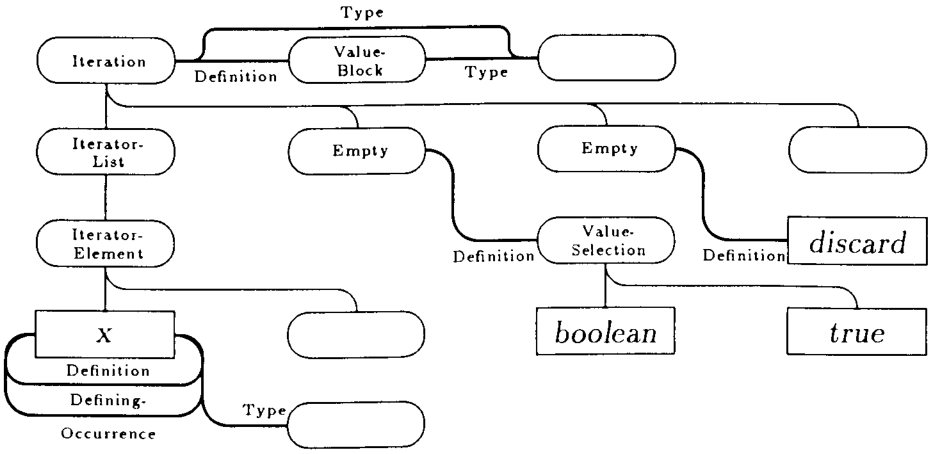
An iterator-list is a list of iterator-elements:

⟨*IteratorList*⟩ ::= **for** ⟦ ⟨*IteratorElement*⟩ ⟧$^{[\,;\,]}$

and an iterator-element consists of a defined-identifier and an iterator:

⟨*IteratorElement*⟩ ::=
⟨*DefinedIdentifier*: *ValueIdentifier*⟩ **in** ⟨*Iterator*: *Value*⟩

The context-dependent relations for an iteration are illustrated as



The iterator of an iterator-element must have a type that is a subtype of

Iterator (T)

and the type of the defined-identifier is T. The filter must be of type *Boolean*. The accumulator must have a type that is a subtype of

Accumulator (B, R)

The body must have a type that is a subtype of B. And the iteration has type R.

The definition of an iteration is derived so that an iteration of the form

for *i* **in** *x*; *j* **in** *y* **andif** *f* (*i*, *j*) **do** *z* *g* (*i*, *j*) **end**

is equivalent to

```
{let i1 be x;  let i2 be y;  let a be z
 keep trying
    {let i be i1.produce();  let j be i2.produce()
     if f (i, j) then a.consume(g (i, j)) end}
   then
    when i1.done, i2.done, a.done then
     begin
       i1.terminate();  i2.terminate();  a.terminate()
      end
  end}
```

Thus the definition of an iteration is a value-block with the appropriate bindings and body.

A.32.3.1 Scope

An iterator-list has a name-layer containing the defined-identifier of each of its iterator-elements. When identifier-lookup starts in the filter or body of an iteration, the name-layer of the iterator-list is searched.

A.32.4 Accumulation

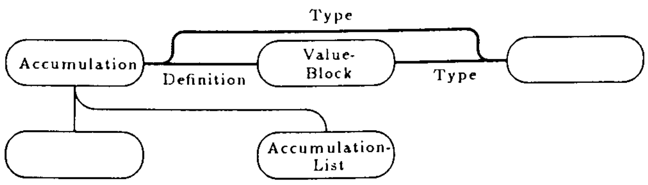
Accumulators are also used in accumulations, which consist of an accumulator and an accumulation-list:

⟨*Accumulation*⟩ ::= ⟨*Accumulator*: *Value*⟩ ⟨*Elements*: *AccumulationList*⟩])

where an accumulation-list is a list of values:

⟨*AccumulationList*⟩ ::= ([{[⟨*Value*⟩]}**[,]**

The context-dependent relations for an accumulation are illustrated as



The accumulator's type must be a subtype of

Accumulator(B, R)

The type of each element in the accumulation-list must be a subtype of B. And the type of the accumulation is R.

The definition of an accumulation is derived so that an accumulation of the form

a([x, y])

is equivalent to

```
{let a1 be a
 try begin
        a1.consume(x)
        a1.consume(y)
        a1.terminate()
      end
   then when a1.done then a1.terminate()
  end}
```

Thus the definition of an accumulation is a value-block with the appropriate bindings and body.

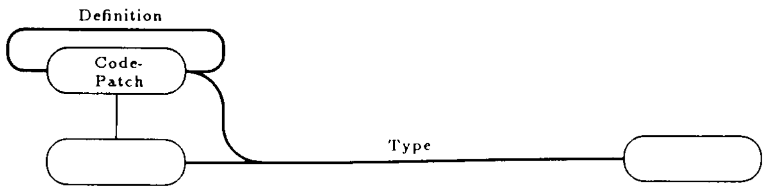
A.33 Code-patch

For representing machine dependent evaluations, Acer provides the code-patch:

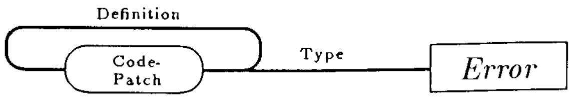
⟨*CodePatch*⟩ ::= **code** ⦃ ⟨*Expression*⟩ ⦄$^{[\,;\,]}$ **end**

The first expression must be a type.

The context-dependent relations for a code-patch are illustrated as



The first expression is the code-patch's type. The type of an empty code-patch is *Error*:



The trailing expressions of a code-patch (i.e., every expression except the first) are used to represent the data and instructions of some particular machine. Typically, the final expression is used to specify the effective-address of the value to be yielded, and the expressions between the first and the last are used to specify the machine instructions to be executed.

Because the trailing expressions of a code-patch are interpreted in an implementation dependent manner, the normal context-dependent relations for expressions do not apply for the trailing expressions. Such expressions simply reuse Acer's existing syntax to denote something else entirely. The advantage of introducing machine dependencies in this way is that Acer provides the same context-free syntax for all implementations and yet also provides a flexible syntax for expressing the data and instructions of a particular machine. For example, the code-patch

code *Integer*; *move*(*d1*, *d0*); *d0* **end**

might be used to move the contents of register *d1* to register *d0* and to yield the final value of *d0* as an *Integer* result. And the code-patch

code *Pointer*(**Any**); *A7* **end**

might be used to yield the value of the stack-register *A7* as a pointer.

A.34 The grammar

A.34.1 Declaration and binding

⟨*FixedValueBinding*⟩ ::=
  let ⟨*DefinedIdentifier*:*ValueIdentifier*⟩ ⟦ : ⟨:*Type*⟩ ⟧ **be** ⟨*Definition*:*Value*⟩

⟨*FixedValueDeclaration*⟩ ::= ⟦ ⟨*DefinedIdentifier*:*ValueIdentifier*⟩ ⟧ : ⟨:*Type*⟩

⟨*TypeBinding*⟩ ::=
  let ⟨*DefinedIdentifier*:*TypeIdentifier*⟩ ⟦ : : ⟨:*Type*⟩ ⟧ **be** ⟨*Definition*:*Type*⟩

⟨*TypeDeclaration*⟩ ::= ⟦ ⟨*DefinedIdentifier*:*TypeIdentifier*⟩ ⟧ : : ⟨:*Type*⟩

⟨*VariableValueBinding*⟩ ::=
  let var ⟦ ⟨*DefinedIdentifier*:*ValueIdentifier*⟩ ⟧ ⟦ : ⟨:*Type*⟩ ⟧ **be** ⟨*Definition*:*Value*⟩

⟨*VariableValueDeclaration*⟩ ::= **var** ⟦ ⟨*DefinedIdentifier*:*ValueIdentifier*⟩ ⟧ : ⟨:*Type*⟩

A.34.2 Type

⟨*Type*⟩ ::=
  ⟨*AbstractType*⟩ ‖ ⟨*ConcreteType*⟩ ‖ ⟨*ReusedTypeIdentifier*⟩ ‖ ⟨*TypeBlock*⟩ ‖
  ⟨*TypeDenoter*⟩ ‖ ⟨*TypeDesignation*⟩ ‖ ⟨*TypeOperator*⟩

⟨*AbstractType*⟩ ::=
  ⟨*OperatorCall*⟩ ‖ ⟨*TypeIdentifier*⟩ ‖ ⟨*TypeSelection*⟩

⟨*ConcreteType*⟩ ::=
  ⟨*AnyType*⟩ ‖ ⟨*DynamicType*⟩ ‖ ⟨*EnumerationType*⟩ ‖ ⟨*FunctionType*⟩ ‖
  ⟨*OptionType*⟩ ‖ ⟨*RecordType*⟩ ‖ ⟨*TupleType*⟩ ‖ ⟨*VariantType*⟩

⟨*AnyType*⟩ ::= **Any**

⟨*DynamicType*⟩ ::= **Dynamic** ⦃ ⟨*Declaration*⟩ ⦄$^{[\,;\,]}$ **end**

⟨*EnumerationType*⟩ ::= **Enumeration** ⦃ ⟨*ValueIdentifier*⟩ ⦄$^{[\,,\,]}$ **end**

⟨*FunctionType*⟩ ::= **Function** ⟨:*Signature*⟩ ⟨*ResultType*:*Type*⟩ **end**

⟨*OperatorCall*⟩ ::= ⟨*Operator:Type*⟩ ⟨*Arguments:ArgumentList*⟩ `)`

⟨*OptionType*⟩ ::= **Option** ⦃ ⟨*ValueIdentifier*⟩ $⦄^{[\,,\,]}$ **end**

⟨*RecordType*⟩ ::= **Record** ⦃ ⟨*Declaration*⟩ $⦄^{[\,;\,]}$ **end**

⟨*ReusedTypeIdentifier*⟩ ::= ⟨*Identifier:TypeIdentifier*⟩ `'` `[` ⟨*DepthIndicator:IntegerLiteral*⟩ `]`

⟨*TupleType*⟩ ::= **Tuple** ⦃ ⟨*Declaration*⟩ $⦄^{[\,;\,]}$ **end**

⟨*TypeBlock*⟩ ::= ⟨*Bindings:BindingList*⟩ ⟨*Body:Type*⟩ `}`

⟨*TypeDenoter*⟩ ::= `{[]}`

⟨*TypeDesignation*⟩ ::= **TYPE** `(` ⟨*:Expression*⟩ `)`

⟨*TypeOperator*⟩ ::= **Operator** ⟨*:Signature*⟩ ⟨*Body:Type*⟩ **end**

⟨*TypeSelection*⟩ ::= ⟨*Base:Value*⟩ `.` ⟨*Selector:TypeIdentifier*⟩

⟨*VariantType*⟩ ::=
  Variant ⟨*Tag:Type*⟩ ⟨*Variants:VariantList*⟩ ⟦ **else** ⟨*Default:Signature*⟩ ⟧ **end**

A.34.3 Value

⟨*Value*⟩ ::=
  ⟨*Accumulation*⟩ ‖ ⟨*AndIfTest*⟩ ‖ ⟨*Assignment*⟩ ‖ ⟨*CodePatch*⟩ ‖ ⟨*CompoundValue*⟩ ‖
  ⟨*Conditional*⟩ ‖ ⟨*Dereference*⟩ ‖ ⟨*DyadicMethodCall*⟩ ‖ ⟨*DynamicInspection*⟩ ‖
  ⟨*FunctionCall*⟩ ‖ ⟨*Index*⟩ ‖ ⟨*IsTest*⟩ ‖ ⟨*IsNotTest*⟩ ‖ ⟨*Iteration*⟩ ‖ ⟨*KeepTrying*⟩ ‖
  ⟨*Literal*⟩ ‖ ⟨*OrdCall*⟩ ‖ ⟨*OrIfTest*⟩ ‖ ⟨*PointerCall*⟩ ‖ ⟨*PrefixMethodCall*⟩ ‖ ⟨*Raise*⟩ ‖
  ⟨*Try*⟩ ‖ ⟨*TryFinally*⟩ ‖ ⟨*UnaryMethodCall*⟩ ‖ ⟨*ValCall*⟩ ‖ ⟨*ValueBlock*⟩ ‖
  ⟨*ValueDenoter*⟩ ‖ ⟨*ValueSelection*⟩ ‖ ⟨*VariantInspection*⟩

⟨*Literal*⟩ ::=
  ⟨*ArrayLiteral*⟩ ‖ ⟨*CharacterLiteral*⟩ ‖ ⟨*DynamicLiteral*⟩ ‖ ⟨*ExceptionLiteral*⟩ ‖
  ⟨*FunctionLiteral*⟩ ‖ ⟨*IntegerLiteral*⟩ ‖ ⟨*LiteralSelection*⟩ ‖ ⟨*RealLiteral*⟩ ‖
  ⟨*RecordLiteral*⟩ ‖ ⟨*ReferenceLiteral*⟩ ‖ ⟨*ReusedValueIdentifier*⟩ ‖ ⟨*StringLiteral*⟩ ‖
  ⟨*TupleLiteral*⟩ ‖ ⟨*ValueIdentifier*⟩ ‖ ⟨*VariantLiteral*⟩ ‖ ⟨*VoidLiteral*⟩

⟨*Accumulation*⟩ ::= ⟨*Accumulator*: *Value*⟩ ⟨*Elements*:*AccumulationList*⟩])

⟨*AndIfTest*⟩ ::= { ⟨*FirstOperand*: *Value*⟩ **andif** ⟨*SecondOperand*: *Value*⟩ }

⟨*ArrayLiteral*⟩ ::= ⟨*Elements*:*ArrayList*⟩ [[**of** ⟨*BaseType*: *Type*⟩]] **end**

⟨*Assignment*⟩ ::= { ⟨*Destination*: *Value*⟩ **becomes** ⟨*Source*: *Value*⟩ }

⟨*CodePatch*⟩ ::= **code** {[⟨*Expression*⟩]}$^{[\,;\,]}$ **end**

⟨*CompoundValue*⟩ ::= **begin** {[⟨*Value*⟩]}$^{[\,;\,]}$ **end**

⟨*Conditional*⟩ ::= ⟨*Branches*:*ConditionalBranchList*⟩ [[**else** ⟨*DefaultBranch*: *Value*⟩]] **end**

⟨*Dereference*⟩ ::= ⟨*Base*: *Value*⟩ @

⟨*DyadicMethodCall*⟩ ::=
  { ⟨*FirstOperand*: *Value*⟩ ⟨*MethodName*: *ValueIdentifier*⟩ ⟨*SecondOperand*: *Value*⟩ }

⟨*DynamicInspection*⟩ ::=
  inspect ⟨*Selector*: *Value*⟩ ⟨*Branches*: *TypeBranchList*⟩
    [[**else** ⟨*DefaultBranch*: *Value*⟩]] **end**

⟨*DynamicLiteral*⟩ ::= **dynamic** {[⟨*Argument*⟩]}$^{[\,,\,]}$ **end**

⟨*ExceptionLiteral*⟩ ::= **exception** (⟨*BaseType*: *Type*⟩)

⟨*FunctionCall*⟩ ::= ⟨*Function*: *Value*⟩ ⟨*Arguments*:*ArgumentList*⟩)

⟨*FunctionLiteral*⟩ ::= **function** ⟨:*Signature*⟩ [[: ⟨*ResultType*: *Type*⟩]] ⟨*Body*: *Value*⟩ **end**

⟨*Index*⟩ ::= ⟨*Base*: *Value*⟩ ⟨*Indices*:*IndexList*⟩]

⟨*IsNotTest*⟩ ::= { ⟨*FirstOperand*: *Value*⟩ **isnot** ⟨*SecondOperand*: *Value*⟩ }

⟨*IsTest*⟩ ::= { ⟨*FirstOperand*: *Value*⟩ **is** ⟨*SecondOperand*: *Value*⟩ }

⟨*Iteration*⟩ ::=
  ⟨*Iterators*:*IteratorList*⟩ [[**andif** ⟨*Filter*: *Value*⟩]] **do**
    [[⟨*Accumulator*: *Value*⟩]] ⟨*Body*: *Value*⟩ **end**

⟨*KeepTrying*⟩ ::=
  keep trying ⟨*Body*: *Value*⟩ ⟨*Branches*: *ValueBranchList*⟩
    ⟦ **else** ⟨*DefaultBranch*: *Value*⟩ ⟧ **end**

⟨*LiteralSelection*⟩ ::= ⟨*Base*: *Type*⟩ . ⟨*Selector*: *ValueIdentifier*⟩

⟨*OrdCall*⟩ ::= **ord** (⟨*Base*: *Value*⟩)

⟨*OrIfTest*⟩ ::= { ⟨*FirstOperand*: *Value*⟩ **orif** ⟨*SecondOperand*: *Value*⟩ }

⟨*PointerCall*⟩ ::= **pointer** (⟨*Base*: *Value*⟩)

⟨*PrefixMethodCall*⟩ ::= ⟨*MethodName*: *ValueIdentifier*⟩ . ⟨*Arguments*: *ArgumentList*⟩)

⟨*Raise*⟩ ::= **raise** ⟨*ExceptionValue*: *Value*⟩ ⟦ **with** ⟨*AssociatedValue*: *Value*⟩ ⟧ **end**

⟨*RecordLiteral*⟩ ::= **record** ⦃ ⟨*Binding*⟩ ⦄$^{[\,,\,]}$ **end**

⟨*ReferenceLiteral*⟩ ::= **reference** (⟨*Base*: *Value*⟩)

⟨*ReusedValueIdentifier*⟩ ::= ⟨*Identifier*: *ValueIdentifier*⟩ ' [⟨*DepthIndicator*: *IntegerLiteral*⟩]

⟨*Try*⟩ ::=
  try ⟨*Body*: *Value*⟩ ⟨*Branches*: *ValueBranchList*⟩ ⟦ **else** ⟨*DefaultBranch*: *Value*⟩ ⟧ **end**

⟨*TryFinally*⟩ ::= **try** ⟨*Body*: *Value*⟩ **finally** ⟨*FinalAction*: *Value*⟩ **end**

⟨*TupleLiteral*⟩ ::= **tuple** ⦃ ⟨*Argument*⟩ ⦄$^{[\,,\,]}$ **end**

⟨*UnaryMethodCall*⟩ ::= { ⟨*MethodName*: *ValueIdentifier*⟩ ⟨*Operand*: *Value*⟩ }

⟨*ValueBlock*⟩ ::= ⟨*Bindings*: *BindingList*⟩ ⟨*Body*: *Value*⟩ }

⟨*ValCall*⟩ ::= **val** (⟨*BaseType*: *Type*⟩ ⟦ , ⟧ ⟨*Ordinal*: *Value*⟩)

⟨*ValueDenoter*⟩ ::= {()}

⟨*ValueSelection*⟩ ::= ⟨*Base*: *Value*⟩ . ⟨*Selector*: *ValueIdentifier*⟩

⟨*VariantInspection*⟩ ::=
  inspect ⟨*Selector*: *Value*⟩ ⟨*Branches*: *ValueBranchList*⟩
    ⟦ **else** ⟨*DefaultBranch*: *Value*⟩ ⟧ **end**

⟨*VariantLiteral*⟩ ::=
  variant ⟨*Tag*: *ValueIdentifier*⟩ **of** ⟨*BaseType*: *Type*⟩ **with** ⟨*Arguments*: *ArgumentList*⟩)

⟨*VoidLiteral*⟩ ::= {}

A.34.4 Miscellaneous

⟨*Miscellaneous*⟩ ::=
⟨*Empty*⟩ ‖ ⟨*AccumulationList*⟩ ‖ ⟨*ArgumentList*⟩ ‖ ⟨*ArrayList*⟩ ‖ ⟨*BindingList*⟩ ‖ ⟨*ConditionalBranch*⟩ ‖ ⟨*ConditionalBranchList*⟩ ‖ ⟨*IndexList*⟩ ‖ ⟨*IteratorElement*⟩ ‖ ⟨*IteratorList*⟩ ‖ ⟨*Signature*⟩ ‖ ⟨*TypeBranchList*⟩ ‖ ⟨*TypeWhenBranch*⟩ ‖ ⟨*ValueBranchList*⟩ ‖ ⟨*ValueWhenBranch*⟩ ‖ ⟨*VariantElement*⟩ ‖ ⟨*VariantList*⟩ ‖ ⟨*WhenCondition*⟩

⟨*AccumulationList*⟩ ::= `([` ⦃ ⟨*Value*⟩ ⦄$^{[\,,\,]}$

⟨*ArbitraryList*⟩ ::= **arbitrary** ⦃ `[` ⟨*Arbitrary*⟩ `]` ⦄ **end**

⟨*ArgumentList*⟩ ::= `(` ⦃ ⟨*Argument*⟩ ⦄$^{[\,,\,]}$

⟨*ArrayList*⟩ ::= **array** ⦃ ⟨*Value*⟩ ⦄$^{[\,,\,]}$

⟨*BindingList*⟩ ::= `{` ⦃ ⟨*Binding*⟩ ⦄$^{[\,;\,]}$ ⟦ ; ⟧

⟨*ConditionalBranch*⟩ ::= ⟨*Condition*: *Value*⟩ **then** ⟨*Consequent*: *Value*⟩

⟨*ConditionalBranchList*⟩ ::= **if** ⦃ ⟨*ConditionalBranch*⟩ ⦄$^{\textbf{elsif}}$

⟨*Empty*⟩ ::= **nothing**

⟨*IndexList*⟩ ::= `[` ⦃ ⟨*Value*⟩ ⦄$^{[\,,\,]}$

⟨*IteratorElement*⟩ ::= ⟨*DefinedIdentifier*: *ValueIdentifier*⟩ **in** ⟨*Iterator*: *Value*⟩

⟨*IteratorList*⟩ ::= **for** ⦃ ⟨*IteratorElement*⟩ ⦄$^{[\,;\,]}$

⟨*Signature*⟩ ::= `(` ⦃ ⟨*Declaration*⟩ ⦄$^{[\,;\,]}$ `)`

⟨*TypeBranchList*⟩ ::= **Then** ⦃ ⟨*TypeWhenBranch*⟩ ⦄$^{[\,;\,]}$

⟨*TypeWhenBranch*⟩ ::=
when ⟨*Condition*: *Type*⟩ ⟦ **with** ⟨*DefinedIdentifier*: *ValueIdentifier*⟩ ⟧
then ⟨*Consequent*: *Value*⟩

⟨*ValueBranchList*⟩ ::= **then** ⦃ ⟨*ValueWhenBranch*⟩ ⦄$^{[\,;\,]}$

⟨*ValueWhenBranch*⟩ ::=
    ⟨*Condition: WhenCondition*⟩ ⟦ **with** ⟨*DefinedIdentifier: ValueIdentifier*⟩ ⟧
        then ⟨*Consequent: Value*⟩

⟨*VariantElement*⟩ ::= ⟨*Condition: WhenCondition*⟩ **then** ⟨*Consequent: Signature*⟩

⟨*VariantList*⟩ ::= **of** {[⟨*VariantElement*⟩]}$^{[\,;\,]}$

⟨*WhenCondition*⟩ ::= **when** {[⟨*Value*⟩]}$^{[\,,\,]}$

A.34.5 Generic

⟨*Arbitrary*⟩ ::= ⟨*Argument*⟩ ‖ ⟨*Declaration*⟩ ‖ ⟨*Miscellaneous*⟩

⟨*Argument*⟩ ::= ⟨*Binding*⟩ ‖ ⟨*Expression*⟩

⟨*Binding*⟩ ::= ⟨*TypeBinding*⟩ ‖ ⟨*ValueBinding*⟩

⟨*Block*⟩ ::= ⟨*TypeBlock*⟩ ‖ ⟨*ValueBlock*⟩

⟨*Declaration*⟩ ::= ⟨*TypeDeclaration*⟩ ‖ ⟨*ValueDeclaration*⟩

⟨*Denoter*⟩ ::= ⟨*TypeDenoter*⟩ ‖ ⟨*ValueDenoter*⟩

⟨*Expression*⟩ ::= ⟨*Type*⟩ ‖ ⟨*Value*⟩

⟨*Identifier*⟩ ::= ⟨*TypeIdentifier*⟩ ‖ ⟨*ValueIdentifier*⟩

⟨*ReusedIdentifier*⟩ ::= ⟨*ReusedTypeIdentifier*⟩ ‖ ⟨*ReusedValueIdentifier*⟩

⟨*ValueDeclaration*⟩ ::= ⟨*FixedValueDeclaration*⟩ ‖ ⟨*VariableValueDeclaration*⟩

⟨*ValueBinding*⟩ ::= ⟨*FixedValueBinding*⟩ ‖ ⟨*VariableValueBinding*⟩

A.34.6 Lexeme

⟨*CharacterLiteral*⟩ ::=
‘ ⦅ ⟨*#LowercaseLetter*⟩ ‖ ⟨*#UppercaseLetter*⟩ ‖ ⟨*#SymbolicLetter*⟩ ‖
⟨*#Punctuation*⟩ ‖ ⟨*#Digit*⟩ ‖ ⟨*#Space*⟩ ‖ ⟨*#DoubleQuote*⟩ ‖ ⟨*#Other*⟩ ⦆ ’

⟨*IntegerLiteral*⟩ ::= ⟦ - ⟧ ⟨*#Digit*⟩ ⦃ ⟨*#Digit*⟩ ⦄

⟨*RealLiteral*⟩ ::=
⟦ - ⟧ ⟨*#Digit*⟩ ⦃ ⟨*#Digit*⟩ ⦄ . ⟨*#Digit*⟩ ⦃ ⟨*#Digit*⟩ ⦄
⟦ ⦅ **e** ‖ **E** ⦆ ⟦ - ⟧ ⟨*#Digit*⟩ ⦃ ⟨*#Digit*⟩ ⦄ ⟧

⟨*StringLiteral*⟩ ::=
" ⦃ ⟨*#LowercaseLetter*⟩ ‖ ⟨*#UppercaseLetter*⟩ ‖ ⟨*#SymbolicLetter*⟩ ‖
⟨*#Punctuation*⟩ ‖ ⟨*#Digit*⟩ ‖ ⟨*#Space*⟩ ‖ ⟨*#Other*⟩ ⦄ "

⟨*TypeIdentifier*⟩ ::=
⟨*#UppercaseLetter*⟩ ⦃ ⟨*#LowercaseLetter*⟩ ‖ ⟨*#UppercaseLetter*⟩ ‖ ⟨*#Digit*⟩ ⦄

⟨*ValueIdentifier*⟩ ::=
⟨*#LowercaseLetter*⟩ ⦃ ⟨*#LowercaseLetter*⟩ ‖ ⟨*#UppercaseLetter*⟩ ‖ ⟨*#Digit*⟩ ⦄ ‖
⟨*#SymbolicLetter*⟩ ⦃ ⟨*#SymbolicLetter*⟩ ⦄

A.34.7 Comment

⟨*Comment*⟩ ::=
% ⦃ ⟨*#LowercaseLetter*⟩ ‖ ⟨*#UppercaseLetter*⟩ ‖ ⟨*#SymbolicLetter*⟩ ‖
⟨*#Punctuation*⟩ ‖ ⟨*#Digit*⟩ ‖ ⟨*#DoubleQuote*⟩ ‖ ⟨*#Tab*⟩ ‖ ⟨*#Space*⟩ ‖ ⟨*#Other*⟩ ⦄

A.34.8 Character description

⟨*#LowercaseLetter*⟩ ::=
a ‖ **b** ‖ **c** ‖ **d** ‖ **e** ‖ **f** ‖ **g** ‖ **h** ‖ **i** ‖ **j** ‖ **k** ‖ **l** ‖ **m** ‖
n ‖ **o** ‖ **p** ‖ **q** ‖ **r** ‖ **s** ‖ **t** ‖ **u** ‖ **v** ‖ **w** ‖ **x** ‖ **y** ‖ **z**

⟨*#UppercaseLetter*⟩ ::=
A ‖ **B** ‖ **C** ‖ **D** ‖ **E** ‖ **F** ‖ **G** ‖ **H** ‖ **I** ‖ **J** ‖ **K** ‖ **L** ‖ **M** ‖
N ‖ **O** ‖ **P** ‖ **Q** ‖ **R** ‖ **S** ‖ **T** ‖ **U** ‖ **V** ‖ **W** ‖ **X** ‖ **Y** ‖ **Z** ‖ _

⟨*#SymbolicLetter*⟩ ::=
`!` ∥ `#` ∥ `$` ∥ `&` ∥ `*` ∥ `+` ∥ `-` ∥ `/` ∥ `<` ∥ `=` ∥ `>` ∥ `?` ∥ `\` ∥ `^` ∥ `|` ∥ `~`

⟨*#Punctuation*⟩ ::=
`%` ∥ `'` ∥ `(` ∥ `)` ∥ `,` ∥ `.` ∥ `:` ∥ `;` ∥ `[` ∥ `]` ∥ `` ` `` ∥ `{` ∥ `}` ∥ `@`

⟨*#Digit*⟩ ::= **0** ∥ **1** ∥ **2** ∥ **3** ∥ **4** ∥ **5** ∥ **6** ∥ **7** ∥ **8** ∥ **9**

⟨*#DoubleQuote*⟩ ::= `"`

⟨*#Tab*⟩ ::= ASCII 9

⟨*#LineFeed*⟩ ::= ASCII 10

⟨*#CarriageReturn*⟩ ::= ASCII 13

⟨*#Space*⟩ ::= ASCII 32

⟨*#Other*⟩ ::= ASCII 0–8, 11–12, 14–31, 127

Appendix B

The PCAcer manual

This appendix describes PCAcer, a mouse-driven interactive programming environment that supports Acer on an MS-DOS system. PCAcer is designed to operate a three button mouse and a VGA monitor; color text mode, supporting 80 characters by 50 lines, is used.

The appendix includes the following:

- Introductory information and conventions.
- A description of the screen.
- A general description of windows.
- How to edit with each type of window.
- How to work with multiple views.
- How to store and compile programs.
- How to query the semantics attributes.

A familiarity with the basics of MS-DOS is assumed.

B.1 Introduction

Throughout this manual the following conventions are used:

Keys to be pressed are shown as

$\boxed{\text{X}}$ $\boxed{\text{x}}$ $\boxed{\text{space}}$ $\boxed{\text{esc}}$ $\boxed{\text{enter}}$

Mouse buttons to be pressed are shown as

$\boxed{\blacksquare\ \square\ \square}$ $\boxed{\square\ \blacksquare\ \square}$ $\boxed{\square\ \square\ \blacksquare}$

A mouse button can either be *clicked*, i.e., pressed and released, or *held*, i.e., pressed and kept down.

Keys and mouse buttons to be shifted, controlled, or altered when pressed are shown as

$\boxed{\text{shift}}\!\dashv\!\boxed{\blacksquare\ \square\ \square}$ $\boxed{\text{ctrl}}\!\dashv\!\boxed{\text{C}}$ $\boxed{\text{alt}}\!\dashv\!\boxed{\text{X}}$

Because PCAcer makes heavy use of color (e.g., for selecting and highlighting) it is difficult to illustrate how an actual screen will look. A verbal description will therefore have to be adequate.

The sixteen text colors provided by a VGA monitor are named as follows: black, blue, green, cyan, red, magenta, brown, light gray, dark gray, light blue, light green, light cyan, light red, light magenta, yellow, and white; only the first eight of these can be used as background colors.

B.2 Parts of the screen

When the PCAcer environment starts, it creates a screen with:

- the main menu on at the top;
- a mouse cursor;
- and a work area that displays various windows.

The main menu is shown in black text on a white background and any part of the work area not covered by a window is shown in black. The cursor blinks and takes on the text color at its position.

Clicking $\boxed{\blacksquare\ \square\ \square}$ while the cursor is on a main menu command invokes that command. For example, to quit click $\boxed{\blacksquare\ \square\ \square}$ on the *quit* command. Upon quitting, PCAcer stores the contents and positions of all windows (except, as we shall see later, for windows that own attributes). PCAcer reloads this information the next time it runs.

The other main menu commands will be described as they become relevant.

B.3 Node windows and text windows

A window comprises a frame, which consists of four edges and four corners, and a body, which views a portion of an object. There are two types of window, text and node. A text window views lines of characters and a node window views an unparsed node. Both of these are viewed as textual objects.

Every window has a selection, the particular focus of interest. For text, the selection is a character or range of characters, and for node, the selection is a node or range of nodes. Other than the nature of the selection, the different types of window are similar.

B.3.1 The banner

The banner, i.e., the top edge, of a text window appears as:

[5a8c] Empty   (1,1)§(1,1)   1×1   1↔78   1↕47

and the banner of a node window appears as:

[5a8c] Empty   [5a8c] Empty   1×1   1↔78   1↕47

In either case, the banner is displayed in cyan text on a blue background (except for the banner of the selected window, which is shown in yellow text). The information on the banner is interpreted as follows:

Every window has an associated *owner* node, which is indicated at the start of the banner as the node's hexadecimal identity number (i.e., its segment address) and the node's class, e.g., [518c] Empty. Similarly, the object viewed by each window has a maximum line length and a specific number of lines, e.g., 1×1, and the portion in view is given as a range of columns, e.g., 1↔78, and a range of lines, e.g., 1↕47.

The difference between the banners of the two types of window naturally lies in how the selection is specified. For text windows, the selection is given as a start coordinate and an end coordinate, e.g., (1,1)§(1,1). But for node windows, the selection point is given as a node, e.g., [518c] Empty, or as range of nodes, e.g., [519c] BindingList[2↔3]. (Note that a range of nodes is given in terms of a parent node and the positions of the selected children.

B.3.2 The body

The body of a window has a cyan background. The selection is highlighted by a white background. For text windows, text is shown in blue, and for node windows, text is shown in various colors, e.g., keywords are black, type-identifiers are magenta, value-identifiers and value lexemes are light blue, comments are red, and denoters are brown. A node window presents a more readable view of programs.

B.3.3 The view indicators

The left and bottom edge of a window graphically indicate the view position and the view portion relative to the object as a whole. For example, if the entire object is in view, both the left and bottom edge are single line borders. However, if only half the lines are visible, only half the left edge is a single border; the other half is a double edge border. And if the viewed portion of the object is the middle portion, the single line portion of the left edge is centered on the double line portion.

This same notion applies for the bottom edge, but in this case column position is indicated.

B.4 The window stack

Windows in the work area are arranged as a stack, higher windows cover lower windows. Holding $\boxed{\text{shift}}$-$\boxed{■ \; □ \; ❏}$ anywhere on the screen causes the stack-list menu to appear. Releasing the button causes the menu to disappear.

The stack-list menu shows, in order, all windows in the work area. The items of the menu will look something as follows:

◇	[5C9E]	FixedValueBinding	Δ ≪x≫	1
	[5C9E]	FixedValueBinding	Δ ≪x≫	1
	[5C7E]	BindingList	≪y≫	2
▫	[5C4E]	ArbitraryList		3
◇▫	[5C2E]	Empty		4

A ◇ distinguishes a text window from a node window. (The selection point is not shown in the stack-list menu so it cannot be used to distinguish the two in this case.) A ▫ distinguishes a scrap window from an ordinary window. (More on this latter.) An identity number and a node class, e.g., [5C9E] FixedValueBinding, indicate the owner node.

Additional information is shown if the root of the owner is a meaningful top-level node, i.e., a type-binding, fixed-value-binding, fixed-value-declaration, or binding-list. In this case, the defined-identifier of the root, or the defined-identifier of the *primary* binding of the root, is shown, e.g., $\ll$x$\gg$. A Δ precedes the name if the root is not up-to-date with respect to the version stored in the file system. More on all this later.

A final number is assigned to each window to indicate which windows provide different views of the same root node. Numbers are assigned to nodes in a top-down fashion and a window is assigned the same number as a window above it if their owners have the same root. The largest number in the stack-list menu, therefore, indicates the number of disjoint nodes.

The stack-list menu can be used to reorder windows. Holding $\boxed{\text{shift}}$–$\boxed{■ ⧉ ⧉}$ and dragging the cursor to a particular line of the stack-list menu highlights that line. Releasing the button makes the highlighted window become the top window.

B.5 Basic window commands

This section describes commands that apply to either type of window.

B.5.1 The selected window

One of the windows in the window stack is the selected window. It has a banner with yellow text. A window is made the selected window by clicking $\boxed{■ □ □}$ on its banner. (Actually, the selected window is only relevant for one command right now, the *definition-copy* command, which we will see later.)

B.5.2 Stack positioning

Window position within the stack is changed by the stack-list menu, or by clicking $\boxed{■ □ □}$ on a window corner.

Clicking $\boxed{■ □ □}$ on one of the top two corners either makes the window become the top window, or if it is already the top window, makes the window below the top window become the top window. This is an easy way to exchange two overlapping windows.

Clicking $\boxed{■ □ □}$ on one of the bottom two corners either makes the window become the top window, or if its already the top window, make it become the bottom window. This is an easy way to cycle through all windows.

B.5.3 Resizing a window

Windows are resized by repositioning a corner. This is done by holding $\boxed{\square\;\square\;\blacksquare}$ on any corner, dragging the corner to its new position, and releasing the button. As soon as $\boxed{\square\;\square\;\blacksquare}$ is pressed on a corner, an elastic window frame is displayed (in magenta on black). This frame stretches and shrinks to reflect the position of the cursor.

The smallest window has a body of 2 characters by 2 lines. Including the frame, then, the smallest window is 4 characters by 4 lines. When a window is resized and the selection is not in view in the result, the view is corrected to bring it into view.

B.5.4 Moving a window

A window is moved by holding $\boxed{\square\;\blacksquare\;\square}$ on any corner, dragging the window to a new position, and releasing the button. As soon as $\boxed{\square\;\blacksquare\;\square}$ is pressed on a corner, a window frame. is displayed (in magenta on black). This frame moves to reflect the position of the cursor.

B.5.5 Repositioning the view

The portion of the object in view can be changed by clicking mouse buttons on the left or bottom edge.

The view is moved down in the object (i.e., the object is moved up) by clicking $\boxed{\square\;\square\;\blacksquare}$ on the left edge. The indicated line moves up to becomes the first line, unless it is already the first line, in which case the lines move up by one.

The view is moved up (i.e., the object is moved down) by clicking $\boxed{\blacksquare\;\square\;\square}$ on the left edge. The first line moves down to the position of the indicated line, unless the first line is indicated, in which case the lines move down by one.

The view can be completely repositioned by clicking $\boxed{\square\;\blacksquare\;\square}$ on the left edge. The view is then positioned so that the fraction of the object above and below the view is proportional to the fraction of the left edge that is above and below the click point. Hence, clicking $\boxed{\square\;\blacksquare\;\square}$ at the top of the left edge repositions the view to the beginning of the object. Clicking $\boxed{\square\;\blacksquare\;\square}$ at the bottom of the left edge repositions the view to the end of the object.

Clicking $\boxed{\blacksquare\;\square\;\square}$, $\boxed{\square\;\square\;\blacksquare}$, or $\boxed{\square\;\blacksquare\;\square}$ on the bottom edge has the corresponding effect as on the left edge, except column position is affected.

B.6 Window conversion

A new node window is created by clicking $\boxed{\blacksquare\ \square\ \square}$ on the *new* command of the main menu. The new window becomes both the top window and the selected window. It owns a new empty node, which is also its selection point. (In node windows, an empty node is printed as $\emptyset$ rather than as **nothing**.)

A node window is converted to a text window by pressing any printable key while the cursor is on the window. (If a range of nodes is selected, the selection is reset to the parent first.) The resulting text window contains a textual representation of the selected node, which is also the owner. The selection starts out as the entire textual object so it can be easily deleted and replaced.

Clicking $\boxed{\square\ \square\ \blacksquare}$ on the right edge of a text window converts it back to a node window. This is done by parsing the text to yield a node and replacing the owner with that node. The resulting node window has the root node as its owner and the new replacement node as its selection.

As will be explained next, the replaced node is inserted in the scrap node window and the parsed text is inserted in the scrap text window.

B.7 Scrap windows

The PCAcer environment will, at all times, contain at least two windows, namely the scrap text window and the scrap node window. A scrap window is distinguished from other windows by the fact that its banner begins with a □ .

The scrap windows are special in several ways. Firstly, they cannot be converted between node and text, as can ordinary windows. And most importantly, all deletions are inserted into the appropriate scrap window. For this reason, editing a scrap window can have rather unexpected effects. For example, deleting from a scrap window has the effect of inserting elsewhere in the same window. How this works shall become clear as the editing commands for ordinary windows are described.

The contents of the scrap windows can be discarded simply by clicking $\boxed{\blacksquare\ \square\ \square}$ on the *clear* command of the main menu. This may become necessary as space limitations become a problem.

B.8 Editing text

An empty text window is created as follows: click the *new* command of the main menu; move the cursor onto the new window; press $\boxed{\text{space}}$ to convert it to a text window; and click $\boxed{\text{alt}}\text{-}\boxed{\blacksquare\ \square\ \square}$, while the cursor is still in the window, to delete all the text. The result is in an empty text window with an irrelevant unattached node as its owner.

The purpose of a text window is to specify textually a replacement for its owner. In the above case, the owner is irrelevant because it is unattached and empty, but in general, the owner may be attached. In that case, a successful conversion of the text window to a node window, by clicking $\boxed{\square\ \square\ \blacksquare}$ or the right edge, results in the owner being replaced, in context, by the newly parsed node.

If conversion to a node window is unsuccessful, a flashing yellow error message is displayed on the bottom edge of the window. Conversion can fail because either the resulting node cannot stand in place of the owner, or some token cannot be accepted by the parser. In the later case, the offending text becomes the selection point and an error diagnostic is displayed.

When a window displays a flashing error message, no further operations can be performed on that window until, while the cursor is on the window, a button is clicked or a key is pressed. That click or press has the effect of removing the error message, and no other effect.

B.8.1 Selection

The primary operation on a text window is setting the selection.

Clicking $\boxed{\blacksquare\ \square\ \square}$, while the cursor is in the body of a text window, selects the character at that position. Holding $\boxed{\blacksquare\ \square\ \square}$, dragging it, and releasing it selects a range of characters. The selection is continuously modified on the screen to reflect the position of the cursor.

Another way of selecting a range of characters, particularly useful when the range is not entirely in view, is selecting one end point by clicking $\boxed{\blacksquare\ \square\ \square}$ and selecting the other end point by clicking $\boxed{\square\ \square\ \blacksquare}$. A view modification can occur between the two selections. Holding $\boxed{\square\ \square\ \blacksquare}$ and dragging it extends the selection, just as it does when dragging $\boxed{\blacksquare\ \square\ \square}$.

Textual selection can take on three different modes, *character*, *token*, and *line*. Ordinarily, the selection mode is *character*. However, clicking $\boxed{\blacksquare\ \square\ \square}$ on a character that is already in the selection causes the selection mode to become *token*. As a result, the token at the click point becomes selected. Tokens are determined by the same lexical analyzer used by PCAcer's parser.

Furthermore, clicking $\boxed{\blacksquare\ \square\ \square}$ within the selection while in *token* mode causes the selection mode to become *line*. As a result, the line at the click point becomes selected. Thus, double clicking selects a token and triple clicking selects a line.

Holding $\boxed{\blacksquare\ \square\ \square}$ and dragging it while in *token* mode results in the selection being extended by whole tokens. The same goes for *line* mode. The affect of $\boxed{\square\ \square\ \blacksquare}$, when extending the selection point, is also affected by the selection mode. For example, triple clicking $\boxed{\blacksquare\ \square\ \square}$ at one point and clicking $\boxed{\square\ \square\ \blacksquare}$ at another selects a range of complete lines.

B.8.2 Textual entry

Pressing a printing character, while the cursor is on a text window, inserts the specified character before the first character of the selection. Pressing $\boxed{\text{enter}}$ breaks a line. (PCAcer displays the line-ending character as a space so that it can be selected and even deleted.) Pressing $\boxed{\text{backspace}}$ erases the character before the insertion point; at the beginning of a line, it erases the line-ending character on the previous line, thereby joining the two.

Textual entry does not modify the selection, but the selection does revert to character mode.

B.8.3 Delete

Selected text is deleted by clicking $\boxed{\text{alt}}\!\!-\!\!\boxed{\blacksquare\ \square\ \square}$ in the window body. Afterwards, the selection, which remains in the same mode, is set as if $\boxed{\blacksquare\ \square\ \square}$ where clicked at the position of the first character of the original selection. For example, triple clicking $\boxed{\blacksquare\ \square\ \square}$ and then double clicking $\boxed{\text{alt}}\!\!-\!\!\boxed{\blacksquare\ \square\ \square}$ deletes two lines and leaves a line selected.

The entire range of lines affected by deletion is copied to the scrap text window. The selection of the scrap text window then specifies precisely the range of affected characters within those lines.

The scrap text window is not only a destination for deletions but also a source for insertions, as we shall see next.

B.8.4 Insert

The selected text of the scrap text window is inserted before the selection point of some other text window by clicking $\boxed{\text{alt}}\!\!-\!\!\boxed{\square\ \square\ \blacksquare}$ on the body of that window. The selection point of the window is set to the inserted text.

For any text window (except the scrap window) clicking $\boxed{\text{alt}}\!\dashv\!\boxed{\blacksquare\ \square\ \square}$ and then clicking $\boxed{\text{alt}}\!\dashv\!\boxed{\square\ \square\ \blacksquare}$ has no effect other than copying the selection to the scrap window and changing the selection mode to character.

B.8.5 Yank

Clicking $\boxed{\square\ \blacksquare\ \square}$ in a text window has the same effect as effect as clicking $\boxed{\text{alt}}\!\dashv\!\boxed{\blacksquare\ \square\ \square}$ and then clicking $\boxed{\text{alt}}\!\dashv\!\boxed{\square\ \square\ \blacksquare}$, except that the selection mode is unchanged. In other words, a copy of the range of lines containing the selection is inserted in the scrap text window and the appropriate characters within that range are selected.

A $\boxed{\square\ \blacksquare\ \square}$ is usually followed by a command to change the selection followed by a $\boxed{\text{alt}}\!\dashv\!\boxed{\square\ \square\ \blacksquare}$ to insert the yanked text.

B.8.6 Destructive delete

Clicking $\boxed{\text{ctrl}}\!\dashv\!\boxed{\blacksquare\ \square\ \square}$ on a text window body deletes the selection without copying it to the scrap text window. The deleted text is lost!

This command works as expected on the scrap text window.

B.8.7 Canceling a text window

Pressing $\boxed{\text{esc}}$ while the cursor is on a text window cancels the window. That is, the text is inserted in the scrap text window and the window is converted back to a node window.

B.9 Node editing

An empty node window is created by clicking the *new* command on the main menu. The result is a node window that owns a new unattached empty node, which is also its selection point.

Nodes are much richer objects than mere text. Hence, node windows provide more operations than text windows.

B.9.1 Selection

The selection mechanisms for nodes closely model those for text.

Clicking $\boxed{\blacksquare\ \square\ \square}$, while the cursor is in the body of a node window, selects the node that owns the token at or before the position of the cursor. (Since empty nodes are printed as $\emptyset$, all nodes are visible for selection via their tokens.) Holding $\boxed{\blacksquare\ \square\ \square}$, dragging it, and releasing it selects a range of nodes. Such a range of nodes must have a common parent. The selection is continuously modified on the screen to reflect the position of the cursor. Dragging is not very useful for nodes because often tokens of the parent intervene between the children and so if that token is selected, the whole parent is selected.

A better way of selecting a range of nodes, particularly useful when the range is not entirely in view, is selecting one end point by clicking $\boxed{\blacksquare\ \square\ \square}$ and selecting the other end point by clicking $\boxed{\square\ \square\ \blacksquare}$. A view modification can occur between the two selections. Holding $\boxed{\square\ \square\ \blacksquare}$ and dragging it extends the selection, just as it does when dragging $\boxed{\blacksquare\ \square\ \square}$.

Node selection does not take on different modes, as does textual selection.

B.9.2 Textual entry

Pressing a printing character while the cursor is on a node window, converts the node window to a text window with the selected node as the owner. Modification of the owner's textual form may then commence. Upon completion, a $\boxed{\square\ \square\ \blacksquare}$ on the right edge converts the text window back to a node window, replacing the owner in context. The replacement node, as specified by the text, becomes the selection.

B.9.3 Delete

Selected nodes are deleted by clicking $\boxed{\text{alt}}$-$\boxed{\blacksquare\ \square\ \square}$ in the window body. Delete behaves differently depending on the selection.

On the one hand, if a list child is selected, or a range of list children are selected, the children are simply deleted. Afterwards, the selection becomes the next child, if there is one, the previous child, if there is no next child, or the parent, if no children remain. In addition, the deleted children are inserted into a new arbitrary-list, which in turn is inserted at the end of the arbitrary-list owned by the scrap node window. The selection of the scrap node window is set to the range of nodes that were deleted.

On the other hand, if a construction child is selected, or a range of construction children are selected, one of several possibilities occur. If a child is optional, it can be replaced by an empty node. Otherwise, it must be replaced by a placeholder: for a component that is a value, the identifier ? is used; for a component that is a type, the identifier _ is used;

and for component that is a list, the appropriate empty list is used. Because the deletion of a construction child has the effect of replacing it with a different node, after deletion, the selection becomes the repalcement children. The deleted children are inserted in the scrap node window just as for deleted list children.

It is meaningful to delete (or replace) an unattached node because this has the effect of replacing it with an empty node in every window in which it occurs.

B.9.4 Insert

Clicking $\boxed{\text{alt}}\!\!-\!\!\boxed{\square\ \blacksquare\ \square}$ on the body of a node window inserts a placeholder list element between the tokens on either side of the click point. This new placeholder becomes the selection.

An element can only be inserted into a list since a construction has a fixed number of children. If a list element cannot be inserted at the click point, a warning beep is issued.

B.9.5 Replace

The selection of the scrap node window is inserted in place of the selection of an other node window by clicking $\boxed{\text{alt}}\!\!-\!\!\boxed{\square\ \square\ \blacksquare}$ on the body of that window. Again, replace behaves differently depending on the selection.

On the one hand, if a list child is selected, or a range of list children are selected, the children are deleted and the selected nodes of the scrap node window are inserted in their place. If the inserted nodes do not conform as elements, no operation is performed and a warning beep is issued. In addition, an error message diagnosing the problem appears at the bottom edge of the window. This message is canceled by any button click or key press while the cursor is on the window.

On the other hand, if a construction child is selected, or a range of construction children are selected, the children are replaced by the selected nodes of the scrap node window. The range of nodes selected in the scrap node window must match in number the range of nodes selected in the target window, and each node must conform to the requirements of the construction. Again, a beep and an error message are issued if this is not the case.

The deleted (replaced) nodes are copied to the scrap node window but the selection of the scrap node window remains unchanged.

B.9.6 Yank

Clicking $\boxed{\square\ \blacksquare\ \square}$ on a node window body copies the range of selected nodes to the scrap node window and sets the selection of the scrap node window to those nodes. A subsequent insert uses these nodes.

Note that yank does not copy the definitions of denoters. In fact, yank substitutes an any-type in place of each type-denoter and a void-literal in place of each value-denoter. (High-level copying is provided by definition-copy.)

B.9.7 Destructive delete

Clicking $\boxed{\text{ctrl}}$-$\boxed{\blacksquare\ \square\ \square}$ on a node window body deletes the selection as before, but without copying it to the scrap node window. The deleted nodes are lost!

This command works as expected on the scrap node window.

B.10 Multiple views

The stack-list can contain multiple views of the same node. We have already seen how to alternate between a node view and a text view but such views can even coexist.

B.10.1 Simultaneous text views

Pressing $\boxed{\text{enter}}$ on a node window creates a new text window on top of the existing node window. Like the text window created by pressing $\boxed{\text{space}}$, this text window has the selected node (or the parent of the selected nodes) as its owner, and begins with everything selected. But it occupies only the lower half of the area occupied by the node window and the node window remains on the stack-list. (The text window is positioned so the whole frame of the node window remains visible.)

B.10.2 Multiple node views

When $\boxed{\square\ \square\ \blacksquare}$ is held on the top edge of a node window, an elastic window frame appears. Dragging the mouse stretches and shrinks this frame to reflect the position of the cursor. Releasing the button finishes the command. The result is a new node window, on top of the original. It has the specified frame and the same owner and selection as the original.

B.10.3 Multiple text views

Holding $\boxed{\square\ \square\ \blacksquare}$ on the top edge of a text window has the corresponding effect as on a node window. A new text window, on top the original, is created and it has the same owner and selection as the original.

B.10.4 Subviews and node traversing

Holding $\boxed{\square\ \blacksquare\ \square}$ on the top edge of a node window has the same effect as holding $\boxed{\square\ \square\ \blacksquare}$, except that the owner of the created node window is not the owner of the original window but the selected node (or the parent of the selected nodes) of the original window.

The owner of a node window can be changed in one of three ways. Clicking $\boxed{\blacksquare\ \square\ \square}$ on the right edge sets the owner to be the root of the original owner. Clicking $\boxed{\square\ \blacksquare\ \square}$ on the right edge sets the owner to be the parent of the original owner. And clicking $\boxed{\square\ \square\ \blacksquare}$ on the right edge sets the owner to be the selected node (or the parent of the selected nodes). Hence, one can zoom to the root, the parent, or the selection.

B.10.5 Closing windows

Clicking $\boxed{\text{alt}}$-$\boxed{\blacksquare\ \square\ \square}$ on the top edge of a window closes the window. For a text window, if it is the last view of the text, the text is inserted in the scrap text window. For both types of window, if the owner is not owned, or contained by the owner, of any other window, the node is inserted in the scrap node window, where it becomes the selection. Note that attempting to close the last view of a scrap window simply destroys is contents (much like *clear* does).

B.10.6 Implications of multiple views on editing

Clearly, multiple views must be kept consistent because any changes in one window can affect the views of others.

For multiple text windows, for example, any text entered in one will have to be reflected in the others. Furthermore, if the selection of one window is deleted by an edit command in another, the selection must even be updated. The same problems arise for node windows. PCAcer keeps views consistent using the techniques that follow.

Because a text window is considered a node modification in progress, PCAcer restricts operations that modify nodes that are being viewed as text. Clearly, such operations would be lost when the text is converted to a node and the owner is replaced by that node. Therefore, one cannot delete, insert, or replace any node that is the owner, or is enclosed by the owner, of a text window. If such an operation is attempted, the offending text window is brought to the top of the stack-list with an error message suggesting that it be updated, i.e., either discarded or converted a node. Similarly, one cannot create a text window for a node that is the owner, or is enclosed by the owner, of some other text window.

To keep node windows consistent, any command that results in the deletion or replacement of a node has the following effects. If the node is simply deleted and not replaced by anything, every node window with that nodes as its owner is set to own instead the parent of that node. Otherwise, the deleted node is replaced by another node and so every node window with the deleted nodes as its owner is set to own instead the replacement node. Since deleted nodes are inserted into the scrap node window, any node window that retained a view on a deleted node would provide merely a subview of the scrap node window.

Clearly, the selection of each node window too, like the owner itself, must be kept consistent with respect to the deletion, insertion, and replacement of nodes. This is done much like it is for the owner.

Note that multiple views are possible even for the scrap windows. This may be confusing however, because only the selection of the *original* window is used as the source for insertions and there is nothing to distinguish the copy from the original, except that the copy can be closed whereas attempting to close the original only deletes its contents.

B.11 Top-level nodes

There are four classes of node that are meaningful at the top level, i.e., as unattached nodes without context:

- type-binding
- fixed-value-binding
- fixed-value-declaration
- binding-list

Each will be considered in turn.

B.11.1 Top-level type-bindings

A type-binding is meaningful at the top level because it introduces a global type-identifier. To make the type-identifier visible, however, the type-binding must first be *stored* in a file. This is done by holding $\boxed{\text{shift}}$$\boxed{□ ■ ◫}$ on the node window that owns the type-binding, which causes the node command menu to pop up. Dragging the mouse to select a command and releasing the button invokes the command.

When *store* is invoked for a type-binding, a file containing the type-binding in compacted form is created. The file is stored in the current directory, unless a previous version is present in some search directory, in which case the file is stored in that directory. Because of the limitation of MS-DOS file names, only the first eight characters of the binding's defined-identifier are used. For type-binding files the suffix is '.tb.'

Previous versions of the type-binding are renamed as backups, e.g., '*.tb' becomes '*.tb1,' '.tb1' becomes '*.tb2,' and '*.tb2' is erased. Up to 9 backup versions could be maintained in this way, although currently only two backups are maintained. If the current directory contains a directory named 'bak,' the backup versions are moved to that directory. This helps to keep working directories clear of various auxiliary files.

It is the presence of a type-binding file in the file system that makes the defined-identifier globally visible. PCAcer automatically loads the files as necessary, maintaining a unique node to represent the contents of each file. Therefore, when a type-binding is stored, it becomes an *attribute* node. A node window indicates that the root of its owner is an attribute by the symbol ⊙ at the beginning of its banner.

Attribute nodes are different from ordinary nodes because they should not and cannot be modified. Consider what would happen if a stored type-binding could be modified. Some other node may well map the defining-occurrence of a type-identifier to the defined-identifier of that type-binding; after all, the defined-identifier is globally visible. A subsequent edit could then remove the defined-identifier from the type-binding, leaving the other node with a type-identifier that maps to a now invalid defining-occurrence. Furthermore, since every node potentially refers to the type-binding, the attributes of all nodes could be invalidated by a single change to the type-binding. For these reasons, editing of attributes is prevented.

To modify a type-binding that has become an attribute, all that is necessary is to make a copy, create a new window, and replace the node in the new window with the copy, e.g., click $\boxed{■ □ □}$ on the type-binding, click $\boxed{□ ■ □}$ on the window, click $\boxed{■ □ □}$ on the *new* command, and click $\boxed{\text{alt}}$$\boxed{□ □ ■}$ on the new window. The result is a copy of the type-binding, which can be edited and then stored to replace the previous version. Storing the new version converts it to an attribute node; the old version reverts to being an ordinary node.

B.11.2 Top-level fixed-value-bindings

A fixed-value-binding is meaningful at the top level because it can be *compiled* to produce a fixed-value declaration, which introduces a global value-identifier. It can also be executed to yield its definition. Furthermore, a fixed-value-binding can be *stored* just like a type-binding to create a file containing the fixed-value-binding in compacted form. For a fixed-value-binding, the file has suffix '.vb' and previous versions are backed up, just as for a type-binding.

Unlike a type-binding file, the presence of a fixed-value-binding file in the file system does not make the defined-identifier globally visible. Compilation is necessary first. For this reason, a fixed-value-binding does not become an attribute node when stored. PCAcer nevertheless keeps track of whether a fixed-value-binding is up-to-date with respect to its stored version—if it is not, the stack-list menu displays a Δ in front of the binding's identifier.

A fixed-value-binding is compiled by invoking *compile* in the node command menu. The binding is stored, if it is not up-to-date; it is *validated*, i.e., checked for correctness; it is translated to produce an assembler ('.a') file; and the assembler file is assembled to produce an object ('.o') file. In addition, a fixed-value-declaration is created, stored in a '.vd' file, and made the owner of a new node window; it is is an attribute node and its defined-identifier is globally visible by virtue of the presence of the '.vd' file.

Assembler files, object files, and fixed-value-declaration files are created with the same base name as the fixed-value-binding file. They are stored in the current directory, unless the current directory contains an 'a' directory, an 'o' directory, or a 'vd' directory, in which case the file is stored in the directory that corresponds to its suffix.

Validation can be performed separately from compilation by invoking *validate* in the node commands menu. If an error is found, an offending node is selected and an associated diagnostic error message is produced on the bottom edge. Compilation proceeds only if validation is successful.

PCAcer keeps track of dependencies between files so that files can be correctly maintained. This information is stored in a dependency file associated with each binding file ('.tdp' for a type-binding, and '.vdp' for a fixed-value-binding). These files are stored in the current directory, unless the current directory contains a 'dp' directory, in which case they are stored there instead. Dependency files are created as they are needed and contain essentially just the closure of the binding.

Because of dependencies, compiling a fixed-value-binding may require the compilation of bindings it depends on. This is handled automatically and is of little concern to the user. Any errors that arise are reported by a node window. PCAcer stores all top-level fixed-value-bindings and binding-lists before compilation begins. The scrap windows are cleared to free up space.

A fixed-value-binding is executed by invoking *run* in the node commands menu. The binding is first compiled, if necessary; the required object ('.o') files are then linked to produce an executable ('.axe') file, which is stored in either the current directory or the 'axe' directory of the current directory; and finally, the executable file is loaded to begin execution.

Upon termination, the definition of the binding is yielded. This value, represented as a node, is displayed in a new node window. PCAcer does not yet provide Acer programs with the ability to perform terminal IO so only side-effects to the file system are possible. Hence, every program should produce a result value.

Note that linking can be performed separately from execution by invoking *link* in the node commands menu. The executable file is then created but it is not loaded and executed.

B.11.3 Top-level binding-lists

A binding-list is meaningful at the top level because it is used to compile mutually dependent fixed-value-bindings. A top-level binding-list may only contain type- and fixed-value-bindings and must contain at least one fixed-value-binding; the first fixed-value-binding, then, is called the *primary* binding. A binding-list is executed to yield the definition of the primary binding.

A binding-list is *stored*, just like a type- or fixed-value-binding. The result is the creation of a '.vb' file for each fixed-value-binding in the binding-list. These files are stored just as for individual fixed-value-bindings, including the maintenance of backups. The '.vb' file for the primary binding contains the binding-list; each other file contains a reference to the primary file. Any type-bindings are not stored separately. (As we shall see later, fetching any of the files fetches the entire binding-list.)

A binding-list, like a fixed-value-binding, does not become an attribute node when stored but PCAcer does keeps track of whether a binding-list is up-to-date with respect to its stored version—if it is not, the stack-list menu displays a Δ in front the primary binding's identifier.

A binding-list is compiled by invoking *compile* in the node command menu. It is stored, if it is not up-to-date; it is *validated*, i.e., checked for correctness; it is translated to produce an assembler ('.a') file; and the assembler file is assembled to produce an object ('.o') file.

The assembler file and the object file are created only for the primary binding, although they contain the code to evaluate all the bindings, but a fixed-value-declaration file is created for each fixed-value-binding. The result of compilation is a window for each declaration.

Note that a dependency ('.vdp') file is created only for the primary binding, but it includes the dependencies for all the bindings.

A binding-list is executed by invoking *run* in the node commands menu. The binding-list is first compiled, if necessary; the required object ('.o') files are then linked to produce an executable ('.axe') file, which is stored in either the current directory or the 'axe' directory of the current directory; and finally, the executable file is loaded to begin execution. Upon termination, the definition of the primary binding is yielded. This value, represented as a node, is displayed in a new node window.

B.12 Fetching files

Nodes and text can be fetched from the file system. Automatic fetching of type-bindings and fixed-value-declarations, to resolve defining-occurrences, is performed when necessary. To guide automatic fetching, the directories specified by the directory list of the MS-DOS environment variable `ACERINPUT` are searched. This variable is set something as follows:

```
    set ACERINPUT=.;C:\acer\lib;.\vd;.\dp;.\o;.\axe;.\a
```

PCAcer also supports manual fetching and can fetch four different types of object: text (i.e., a text file with a '.ace' suffix containing an ASCII representation of a node), types (i.e., a '.tb' file), values (i.e., a '.vb' file), and declarations (i.e., a '.vd' file).

Fetch is invoked by holding $\boxed{\blacksquare\ \square\ \square}$ on the *fetch* command of the main menu, dragging the mouse to select the desired option of the pop-up menu, and releasing the button. The screen then fills with a fetch menu. There are four different fetch menus, but they are all very similar. The similarities are described first.

The title at the top indicates whether the menu is for fetching text, types, values, or declarations. The next line displays the current drive and directory, as well as the other drives that can be selected. This is followed by an alphabetized list of files and directories in the current directory. Only files appropriate to the fetch menu are shown in the listing, i.e., files with the appropriate suffix. Furthermore, for '.tb,' '.vb,' and '.vd' files, the file is examined to determine the full name of the contained node and this full name is displayed in the listing. (Recall that file names use only the first eight characters of the defined-identifier.) At the bottom of the screen is a prompt for directly entering the name of the desired file.

The file system can be navigated by clicking $\boxed{\blacksquare\ \square\ \square}$ on a drive name to change the current drive. Clicking $\boxed{\blacksquare\ \square\ \square}$ on a directory in the listing changes the current directory to the selected directory. (Directories are distinguished from files by color.) The listing changes to reflect each change to the current drive and directory.

Selecting a file by clicking $\boxed{\blacksquare\ \square\ \square}$ on its name in the listing loads that file. A new window is created to hold its contents: either a text window, in the case of a '.ace' file, or a node window, otherwise. When a text window is created, it is immediately converted to a node window, if possible.

Pressing $\boxed{\text{esc}}$ cancels the fetch menu. The current drive and directory are restored upon exiting a fetch menu.

B.12.1 Fetching text

The *fetch text* menu lists only directories and '.ace' files in the current directory. However, the name prompt at the bottom of the menu can be used to load any file, including ones without a '.ace' suffix, by typing in the full file name (and path, if desired). Pressing $\boxed{\text{enter}}$ then loads the file into a new text window.

B.12.2 Fetching types

The *fetch type* menu lists only directories and '.tb' files in the current directory. A full name can also be typed directly, in which case a corresponding file is searched according to the search path of `ACERINPUT`. This is like asking for the defining-occurrence of the entered name.

B.12.3 Fetching values

The *fetch value* menu lists only directories and '.vb' files in the current directory. A full name can also be typed directly, in which case a corresponding file is searched according to the search path of `ACERINPUT`.

If the specified fixed-value-binding is part of a binding-list, the entire binding-list is loaded.

B.12.4 Fetching declarations

The *fetch declaration* menu lists only directories and '.vd' files in the current directory. A full name can also be typed directly, in which case a corresponding file is searched according to the search path of ACERINPUT. This is like asking for the defining-occurrence of the entered name.

Note that when the *fetch declaration* menu is invoked, if there is a 'vd' directory in the current directory, that directory automatically becomes the current directory. This, after all, is where declarations will be stored.

B.13 Restoring files

Recall that type-bindings, fixed-value-bindings, and binding-lists are backed up when they are *stored*. A backup version can be *restored*. Restore is invoked by holding $\boxed{■ □ □}$ on the *restore* command of the main menu, dragging the mouse to select the desired option of the pop-up menu (either type or value), and releasing the button. The screen then fills with the selected restore menu, which looks very much like a fetch menu.

B.13.1 Restoring types

The *restore type* menu lists only directories and '.tb' files in the current directory for which a backup exists (i.e., a '.tb1' file). Usually the backup will exist in the 'bak' directory of the current directory. A full name can also be typed directly, in which case a corresponding file is searched according to the search path of ACERINPUT.

Restoring a type-binding has the following effect. First the current version of the type-binding is fetched and loaded into a new window. Then the current version is deleted from the file system. Next the backup versions are renamed, e.g., '*.tb1' becomes '*.tb,' and '*.tb2' becomes '*.tb1.' The result is that the backup version becomes the current version, which is then also loaded into a new window. If the backup version exists in the 'bak' directory, it is moved to the current directory.

B.13.2 Restoring values

The *restore value* menu lists only directories and '.vb' files in the current directory for which a backup exists (i.e., a '.vb1' file). A full name can also be typed directly, in which case a corresponding file is searched according to the search path of ACERINPUT.

Restoring a fixed-value-binding has the following effect. First the current version of the fixed-value-binding is fetched and loaded into a new window. Then the current version is deleted from the file system. Next the backup versions are renamed, e.g., '*.vb1' becomes

'*.vb,' and '*.vb2' becomes '*.vb1.' The result is that the backup version becomes the current version, which is then also loaded into a new window. If the backup version exists in the 'bak' directory, it is moved to the current directory.

When the restored fixed-value-binding is part of a binding-list, the entire binding-list is loaded and all the other fixed-value-bindings in the list are also restored, i.e., their backups are renamed as for the primary binding.

B.14 Querying semantic attributes of nodes

The following semantic attributes can be queried by invoking the corresponding command of the node command menu, which pops up when $\boxed{\text{shift}}$-$\boxed{\square\ \blacksquare\ \square}$ is held on a node window: defining-occurrence, definition, denotation, type, and kind. The query is applied to the selection of the node window. As a result, a new node window is created to contain the node yielded by the query—the root of the yielded node is the owner and the yielded node is the selection. If a query cannot be processed, an error message is displayed instead. Defining-occurrence applies only for an identifier or reused-identifier and the others apply only for an expression.

PCAcer may have to create new nodes in response to a query, just as it has to create a declaration during compilation. Such a created node is an attribute node and so cannot be modified. Furthermore, modifying the source node of an attribute node created in response to a query results in the destruction of the attribute node and its window. This way, displayed attributes are consistent with their source.

Note that the *validate* command of the node command menu can be applied to detect errors in any node.

B.15 Re-expressing a node

A node can be re-expressed in a different context as follows. First, select the source node and select its window's banner to make it the selected window. Then select the destination node, i.e., the node at which the source node is to be expressed. Finally, apply the *definition-copy* command of the node command menu to the destination window. This replaces the destination node with a node that expresses the same object as the selection of the selected window.

Definition-copy is a high-level way of yanking a node and copying it to another context.

It ensures that the copy refers to the same defining-occurrences in the new context as the original does in its context. (Reused-identifiers may have to be introduced.) In addition, its ensures that the copy is expressed without denoters. (To eliminate a denoter, its definition is also copied, and blocks must be introduced when recursion is involved.)

Note that the source window cannot be the same as the destination window because a window can have at most one selection. It is simple, however, to create a copy of the window so that two selections are specified.

B.16 And so on

Clearly, many other operations can be provided, both for node windows and text windows. However, the basics have been covered.

Appendix C

The implementation manual

This appendix describes the implementation of PCAcer and its supporting metaprogramming system. It uses Turbo Pascal running under MS-DOS and consists of the following units, each of the indicated length:

Core	19	Access	1563
Utility	66	MPS	1408
Mouse	286	FileServer	1349
Manager	528	AST	4378
Index	1557	AST2	1478
Sequence	406	Compile	2930
Line	188	Run	2142
Linebuffer	250	Link	1226
Grammar	1109	ObjectView	245
MPS1	2267	LineView	1021
Keywords	125	TokenLine	95
LexicalAnalyzer	529	TokenBuffer	426
Parser	1745	TokenView	1129
Unparser	2675	WindowStack	1196
Makers	853	NodeView	3577

As such, the implementation comprises more than 36,000 lines of code.

The remainder of the appendix is organized as follows. Each section documents the purpose of a particular unit, discussing any interesting implementation techniques; the Turbo Pascal interface appears first, followed by a discussion. The implementation section of each unit is not listed due to the shear volume of code. In addition, repetitive parts of some units are elided (i.e., as ...). Typically, each unit is described before the units that use it.

A familiarity with the basics of (Turbo) Pascal and MS-DOS is assumed.

C.1 Core

```
UNIT Core; INTERFACE {$F+,L-}

  CONST MD = FALSE;

IMPLEMENTATION ...
END.
```

Core serves three purpose: it sets up the use of overlays in its implementation part[1] by initializing an overlay buffer, its uses CRT and then Assigns Input and Output to what they were before CRT changed them, and it provides the Boolean constant MD, which is set to true when the system is to be debugged for memory use, e.g., to check for leakage.

Core is of no particular interest.

C.2 Utility

```
UNIT Utility; INTERFACE {$F+,O+,L-}

  TYPE
    STRING6 = STRING[6];
    STRING11 = STRING[11];

  PROCEDURE ErrorNoise;
  FUNCTION Max(i, j : LONGINT) : LONGINT;
  FUNCTION Min(i, j : LONGINT) : LONGINT;
  FUNCTION IntegerString(x : WORD) : STRING6;
  FUNCTION LongIntegerString(x : LONGINT) : STRING11;

IMPLEMENTATION ...
END.
```

Utility provides convenient functions used in a number of different modules. Again, it is of no particular interest.

[1]Overlays are Turbo Pascal's way of freeing up memory by loading only part of a program's code at a time.

C.3 Mouse

```
UNIT Mouse; INTERFACE {$F+,L-}

  TYPE
    ActionType =
      (NoAction, MovedLeft, MovedUp, MovedDown, MovedRight,
       LeftButtonPressed, LeftButtonReleased, RightButtonPressed,
       RightButtonReleased, MiddleButtonPressed, MiddleButtonReleased);
    ShiftKeys = (Shift, Ctrl, Alt);
    ShiftStateType = SET OF ShiftKeys;
    EventType =
      RECORD
        theAction : ActionType;
        theShiftState : ShiftStateType;
        x, y : INTEGER
      END;

  PROCEDURE Initialize;
  PROCEDURE HideCursor;
  PROCEDURE ShowCursor;
  PROCEDURE MouseGotoXY(x, y : Byte);
  PROCEDURE MouseGetXY(VAR x, y : Byte);
  FUNCTION MousePressed : BOOLEAN;
  PROCEDURE ResetMouse;
  PROCEDURE ReadMouse(VAR theEvent : EventType);

IMPLEMENTATION ...
END.
```

`Mouse` provides an interface for a three button mouse. The mouse driver is started by a call to `Initialize`. Associated with the driver is a screen cursor, which is made visible or invisible using `ShowCursor` and `HideCursor`. The position of the cursor is read by `MouseGetXY` and is set by `MouseGotoXY`. Naturally, movement of the mouse itself also sets the position of the cursor. Whether a mouse button is being held down is determined by `MousePressed`, which is analagous to the `KeyPressed` function provided by `CRT`.

The mouse driver records a buffer of mouse events, e.g., button presses and releases and the position of the cursor and the shift-state of the keyboard at the time of the press or release. The buffer is erased by `ResetMouse`. An event is read by `ReadMouse`.

In short, `Mouse` provides the minimal operations required to support the functionality of a mouse as used in the PCAcer environment.

C.4 Manager

```
UNIT Manager; INTERFACE {$F+,O+,L-}

  TYPE ShortPointer = WORD;

  FUNCTION AllocatePointer(SizeInBytes : WORD) : POINTER;
  PROCEDURE DeallocatePointer(thePointer : POINTER; SizeInBytes : WORD);
  FUNCTION AllocateShortPointer(SizeInBytes : WORD) : ShortPointer;
  PROCEDURE DeallocateShortPointer
              (theShortPointer : ShortPointer; SizeInBytes : WORD);

  VAR FreeUnusedSpace : FUNCTION : BOOLEAN;

IMPLEMENTATION ...
END.
```

`Manager` is the centralized memory manager and must be used for all memory allocation. Regular memory allocation is accomplished by `AllocatePointer` and `DeallocatePointer`.

What is special about this unit is that it also provides a type called `ShortPointer`, which can be used like a regular `Pointer` but occupies only two bytes, rather than the four bytes occupied by a regular `Pointer`. To dereference a `ShortPointer x`, the notation `PTR(x,0)^` is used, a built-in notation of Turbo Pascal.

The reason this is possible is because an MS-DOS 4 byte memory address stored in a `Pointer` variable `p` consists of a 2 byte segment address `SEG(p^)` and a 2 byte offset address `OFS(p^)`. The physical hardware address is derived from the segment and offset addresses by the formula `16 * SEG(p^) + OFS(p^)`; segments are 16 byte chunks of memory. As a result, the offset address of a `Pointer` can be *normalized* to within a range of 0 to 15, with the rest of the offset included as the segment address. As such, only 4 bits of the 16 bits of a normalized `Pointer`'s offset address are actually required.

`AllocateShortPointer` exploits this situation by allocating the requested amount of memory on an even segment boundary. Hence, `AllocateShortPointer` need yield only the segment address, since the offset address is 0. `DeallocateShortPointer` is used to free such a `ShortPointer`.

To maintain the integrity of the allocation scheme, `Manager` must be used for all allocations. Note also that very little memory is wasted, e.g., if a `ShortPointer` to 4 bytes of memory is allocated, the remaining 12 bytes of the segment will be used to satisfy a request for a regular `Pointer` to 12 bytes of memory. (Actually, some memory may be wasted because memory requests are rounded up to the nearest multiple of four.)

The function variable FreeUnusedSpace can be set so that if Manager runs out of space, FreeUnusedSpace is called to free some memory. The function assigned to FreeUnusedSpace should therefore attempt to free memory and should yield true only if successful. Also, the new function should not simply replace the old function already stored in FreeUnusedSpace, it should store the old function in a local variable and should call that function as part of its effort to free up memory. In this way, a chain of functions is formed, each attempting to free memory. (This is analogous to the idea of exit-procedures in Turbo Pascal.)

To summarize then, Manager provides ShortPointer so that memory consumption is drastically reduced. A ShortPointer, after all, occupies half the space of a regular Pointer.

C.5 Index

```
UNIT Index;  INTERFACE {$F+,O+,L-}

  TYPE
    T = ^ IndexRecord;
    Link = ^ LinkRecord;
    Iterator =
      RECORD
        theLink : Link;
        KeySize, InformationSize : BYTE
      END;
    OrderingPredicate = FUNCTION (VAR x, y) : BOOLEAN;
    IndexRecord =
      RECORD
        Less : OrderingPredicate;
        KeySize, InformationSize : BYTE;
        Elements : Link
      END;
    LinkRecord =
      RECORD
        Key : POINTER;
        SiblingSegment : WORD;
        SiblingOffset : BYTE;
        CASE Leaf : BOOLEAN OF
          TRUE: (Child : Link);
          FALSE: (AssociatedInformation : POINTER)
      END;
    Processor = PROCEDURE (VAR x);
    ProcessKeyAndInformation =
        PROCEDURE (VAR TheKey, AssociatedInformation);

    FUNCTION Make(Less : OrderingPredicate;
                  KeySize, InformationSize : BYTE) : T;
    FUNCTION Copy(TheIndex : T) : T;
    PROCEDURE Clear(TheIndex : T);
    PROCEDURE Free(TheIndex : T);
    PROCEDURE Associate(VAR TheKey, AssociatedInformation; TheIndex : T);
    FUNCTION ConditionalAssociate
               (VAR TheKey, AssociatedInformation; TheIndex : T) : BOOLEAN;
    PROCEDURE Include(VAR TheKey; TheIndex : T);
    FUNCTION ConditionalInclude(VAR TheKey; TheIndex : T) : BOOLEAN;
    FUNCTION Lookup(VAR TheKey; TheIndex : T;
                    VAR AssociatedInformation) : BOOLEAN;
    FUNCTION Member(VAR TheKey; TheIndex : T) : BOOLEAN;
    PROCEDURE Exclude(VAR TheKey; TheIndex : T);
    FUNCTION ConditionalExclude(VAR TheKey; TheIndex : T) : BOOLEAN;
    FUNCTION Union(Index1, Index2 : T) : T;
    PROCEDURE UnionAndSet(Index1, Index2 : T);
    FUNCTION Intersection(Index1, Index2 : T) : T;
    PROCEDURE IntersectionAndSet(Index1, Index2 : T);
    FUNCTION Difference(Index1, Index2 : T) : T;
    PROCEDURE DifferenceAndSet(Index1, Index2 : T);
    FUNCTION Empty(Index1 : T) : BOOLEAN;
    FUNCTION Equal(Index1, Index2 : T) : BOOLEAN;
    FUNCTION Less(Index1, Index2 : T) : BOOLEAN;
    FUNCTION Disjoint(Index1, Index2 : T) : BOOLEAN;
    FUNCTION Subset(Index1, Index2 : T) : BOOLEAN;
    FUNCTION ProperSubset(Index1, Index2 : T) : BOOLEAN;
    PROCEDURE ScanKeys(TheIndex : T; Process : Processor);
    PROCEDURE ScanKeysAndInformation
        (TheIndex : T; Process : ProcessKeyAndInformation);
    PROCEDURE ScanInformation(TheIndex : T; Process : Processor);
    PROCEDURE GetIterator(VAR theIterator : Iterator; TheIndex : T);
    FUNCTION NextKey(VAR theIterator : Iterator; VAR TheKey) : BOOLEAN;
    FUNCTION NextKeyAndInformation
              (VAR theIterator : Iterator;
               VAR TheKey, TheInformation) : BOOLEAN;
    FUNCTION NextInformation(VAR theIterator : Iterator;
                             VAR TheInformation) :  BOOLEAN;
    PROCEDURE LeastKey(TheIndex : T; VAR Key);
    PROCEDURE LeastKeyAndInformation(TheIndex : T; VAR Key, Information);
    PROCEDURE LeastInformation(TheIndex : T; VAR Information);

IMPLEMENTATION ...
END.
```

`Index` provides a versatile data structure for creating sets and mappings for any base type for which a total order exists. In particular, it is used to create a mapping or association between values of one type, keys, and values of some other (though not necessarily different) type, associated information. If the associated information is absent, the mapping is just a set. The data structuring mechanism of `Index` facilitates $O(\log n)$ insertion, deletion, and searching, where n is the number of elements in the index.

An index is created by `Make`, which takes the following: an `OrderingPredicate` that in turn takes values of the key type and must act like less-than; a `KeySize` that indicates the size of keys and must be 4 or less; and an `InformationSize` that indicates the size of information and must also be 4 or less—if `InformationSize` is 0, the created index is just a set.

It is possible to produce a `Copy` of an index and to `Clear` an index of its elements. An index is deallocated by `Free`. A key and its associated information are inserted into an index by either `Associate` or `ConditionalAssociate`—`ConditionalAssociate` inserts the information only if the key is not already present and yields `true` only if the information is actually inserted. `Include` and `ConditionalInclude` work similarly, but no associated information is provided with the key.

To determine the information associated with a key in an index, `Lookup` is used. It returns `true` if the key is present, in which case the associated information is also set. Similarly, `Member` determines if a key is in an index.

`Exclude` removes a key from an index and `ConditionalExclude` does the same, but it returns `true` if the key was actually a member.

Various other set-based operations are provided for combining and comparing indexes. Their effect is evident from their names.

`Scan` functions are also provided. They apply a `Processor` function to all the keys and/or information of an index. Also, the notion of `Iterators` is supported by `GetIterator`, which initializes a record of type `Iterator`. This can then be used by one of the `Next` functions to sequence through the elements. A `Next` function yields `true` only if there is a next key and/or information.

Finally, the `Least` functions are provided for easy access to the lowest key and/or information in an index. (Remember, an index is sorted according to its predicate.)

A word of caution is in order with the use of this unit. Its facilities are extremely versatile but care must be taken in their use since there is no type checking to ensure that keys and information are associated with the right 'type' of index. This highlights a glaring weakness of Pascal. Certainly an Acer implementation would provide improved type safety.

C.6 Sequence

```
UNIT Sequence; INTERFACE {$F+,O+,L-}

USES Manager;

TYPE SequenceType = POINTER;

CONST MaxSequenceLength = 65536;

FUNCTION Construct(theSequenceLength : WORD) : SequenceType;
PROCEDURE Destruct(VAR theSequence : SequenceType);
PROCEDURE Expand(VAR theSequence : SequenceType; Position, Amount : WORD);
PROCEDURE Contract(VAR theSequence : SequenceType;
                   Position, Amount : WORD);
PROCEDURE Insert(Source : SequenceType;
                 VAR Destination : SequenceType;
                 DestinationPosition : WORD);
PROCEDURE Replace(Source : SequenceType;
                  VAR Destination : SequenceType;
                  DestinationPosition, DestinationAmount : WORD);
FUNCTION Copy(theSequence : SequenceType) : SequenceType;
PROCEDURE SubInsert(Source : SequenceType;
                    SourcePosition, SourceAmount : WORD;
                    VAR Destination : SequenceType;
                    DestinationPosition : WORD);
PROCEDURE SubReplace(Source : SequenceType;
                     SourcePosition, SourceAmount : WORD;
                     VAR Destination : SequenceType;
                     DestinationPosition, DestinationAmount : WORD);
FUNCTION SubCopy(theSequence : SequenceType;
                 StartPosition, Amount : WORD) : SequenceType;

IMPLEMENTATION ...
END.
```

Sequence provides a type for representing sequences of bytes. Such a sequence is allocated on even segment boundary and is represented as a Pointer to the byte after the last byte in the sequence. Therefore, the length of a sequence s is OFS(s^) and a Pointer to the nth byte of s is PTR(SEG(s^), n-1).

A sequence can be Constructed, Destructed, Expanded at a given by position by a given amount, and Contracted at a given position by a given amount. Various Insert, Replace, and Copy operations are provided as well.

A trick used in the implementation of Sequence is that the actual number of bytes allocated to a sequence is always an even power of 2. Thus if a sequence has length n, it is actually allocated as $2^{\lceil \log_2 n \rceil}$ bytes. Thus sequences can often be Expanded or Contracted without reallocation.

Sequence is used to implement a number of other units, including Line, LineBuffer, TokenLine, and TokenBuffer.

C.7 Line

```
UNIT Line; INTERFACE {$F+,O+,L-}

USES Manager, Sequence;

TYPE
  LineType = SequenceType;
  LinePointer = ^ LineType;
  CharPointer = ^ CHAR;
  StringPointer = ^ STRING;

CONST MaxLineLength = MaxSequenceLength;

FUNCTION Construct(theLineLength : WORD) : LineType;
PROCEDURE Destruct(VAR theLine : LineType);
PROCEDURE Expand(VAR theLine : LineType; Position, Amount : WORD);
PROCEDURE AppendElement(theCharacter : CHAR; VAR theLine : LineType);
PROCEDURE Contract(VAR theLine : LineType; Position, Amount : WORD);
FUNCTION Length(theLine : LineType) : WORD;
FUNCTION ToString(theLine : LineType) : StringPointer;
FUNCTION NthElement(theLine : LineType; n : WORD) : CharPointer;
PROCEDURE Insert(Source : LineType;
                 VAR Destination : LineType;
                 DestinationPosition : WORD);
PROCEDURE Replace(Source : LineType;
                  VAR Destination : LineType;
                  DestinationPosition, DestinationAmount : WORD);
FUNCTION Copy(theLine : LineType) : LineType;
PROCEDURE SubInsert(Source : LineType;
                    SourcePosition, SourceAmount : WORD;
                    VAR Destination : LineType;
                    DestinationPosition : WORD);
```

```
PROCEDURE SubReplace(Source : LineType;
                     SourcePosition, SourceAmount : WORD;
                     VAR Destination : LineType;
                     DestinationPosition, DestinationAmount : WORD);
FUNCTION SubCopy(theLine : LineType;
                 StartPosition, Amount : WORD) : LineType;

IMPLEMENTATION ...
END.
```

Line provides a type for representing space-efficient strings. Such lines are implemented using Sequence. A String of length n is represented as a sequence of bytes of length $n+1$, where the first byte is the length of the string. In this way, a line can be treated as a regular String using the conversion function ToString.

The same operations as those that apply to sequences are provided for lines, although Line also provides the functions Length and NthElement to access, respectively, the length and the nth character of a line. (Note that NthElement yields a pointer to the nth character.)

C.8 LineBuffer

```
UNIT LineBuffer; INTERFACE {$F+,O+,L-}

USES Line, Manager, Sequence;

TYPE
  LineBufferType = SequenceType;
  LineBufferPointer = ^ LineBufferType;

CONST
  MaxLineBufferLength = MaxSequenceLength DIV 4;
  NullLineBuffer : LineBufferType = NIL;

FUNCTION Construct(theLineBufferLength : WORD) : LineBufferType;
PROCEDURE Destruct(VAR theLineBuffer : LineBufferType);
PROCEDURE Expand(VAR theLineBuffer : LineBufferType;
                 Position, Amount : WORD);
PROCEDURE Contract(VAR theLineBuffer : LineBufferType;
                   Position, Amount : WORD);
FUNCTION Length(theLineBuffer : LineBufferType) : WORD;
FUNCTION NthElement(theLineBuffer : LineBufferType;
                    n : WORD) : LinePointer;
PROCEDURE Insert(Source : LineBufferType;
                  VAR Destination : LineBufferType;
                  DestinationPosition : WORD);
PROCEDURE DestructiveInsert(VAR Source : LineBufferType;
                            VAR Destination : LineBufferType;
                            DestinationPosition : WORD);
PROCEDURE Replace(Source : LineBufferType;
                  VAR Destination : LineBufferType;
                  DestinationPosition, DestinationAmount : WORD);
FUNCTION Copy(theLineBuffer : LineBufferType) : LineBufferType;
PROCEDURE SubInsert(Source : LineBufferType;
                    SourcePosition, SourceAmount : WORD;
                    VAR Destination : LineBufferType;
                    DestinationPosition : WORD);
PROCEDURE SubReplace(Source : LineBufferType;
                     SourcePosition, SourceAmount : WORD;
                     VAR Destination : LineBufferType;
                     DestinationPosition, DestinationAmount : WORD);
FUNCTION SubCopy(theLineBuffer : LineBufferType;
                 StartPosition, Amount : WORD) : LineBufferType;

IMPLEMENTATION ...
END.
```

LineBuffer provides a type for representing sequences of lines. It is implemented using Sequence and provides the same kinds of operations.

LineBuffer is used to represent the comments associated with nodes and the textual objects that appear in PCAcer text windows.

C.9 Grammar

```
UNIT Grammar; INTERFACE {$F+,O+,L-}

  TYPE
    NodeClass =
      (Empty,
       Accumulation, ..., VoidLiteral,
       AccumulationList, ..., WhenCondition,
       CharacterLiteral, ..., ValueIdentifier);
    NodeClassSet = SET OF NodeClass;

  CONST
    AbstractType = [OperatorCall, TypeIdentifier, TypeSelection];
    ...
    WhenBranch = [TypeWhenBranch, ValueWhenBranch];
    MinConstructionType = Accumulation;
    MaxConstructionType = VoidLiteral;
    MinListedType = AccumulationList;
    MaxListedType = WhenCondition;
    MinLexicalType = CharacterLiteral;
    MaxLexicalType = ValueIdentifier;
    ConstructionTypes = [MinConstructionType .. MaxConstructionType];
    ListedTypes = [MinListedType .. MaxListedType];
    LexicalTypes = [MinLexicalType .. MaxLexicalType];
    NoChildConstructions =
      [AnyType, Empty, VoidLiteral, TypeDenoter, ValueDenoter];
    ...
    FourChildConstructions = [Iteration];

  FUNCTION NodeClassString(k : NodeClass) : STRING;
  FUNCTION ComponentCount(k : NodeClass) : INTEGER;
  FUNCTION OptionalComponentQ(k : NodeClass; n : INTEGER) : BOOLEAN;
  FUNCTION NthComponentName(k : NodeClass; n : INTEGER) : STRING;
  PROCEDURE NthComponentDomain
              (k : NodeClass; n : INTEGER; VAR Result : NodeClassSet);
  PROCEDURE BaseDomainOfListedType(k : NodeClass;
                                   VAR Result : NodeClassSet);

IMPLEMENTATION ...
END.
```

Grammar defines the type NodeClass to classify nodes; constructions are first, followed by lists, and finally lexemes. The set of node classes specified by each Acer alternation rule is given as a set constant named according to the alternation rule.

Additional functions are provided to give generic access to Acer's abstract grammar. NodeClassString yields the spelling of a node class. ComponentCount yields the number of components defined by a construction class. OptionalComponentQ determines whether a particular component of a construction is optional, NthComponentName determines the name, and NthComponentDomain determines the set of node classes that may occur.

Similarly, BaseDomainOfListedType yields the set of node classes that may occur as a list element.

C.10 MPS1

```
UNIT MPS1; INTERFACE {$F+,O+,L-}
```

```
USES Grammar, LineBuffer, Line, Manager, Sequence;

TYPE
  Node_ = ShortPointer;
  NodePointer = ^ NodeCell;
  ElementPointer = ^ Node_;
  EmptyRecord =
    RECORD
    END;
  AccumulationRecord =
    RECORD
      AccumulatorOf, ElementsOf : Node_;
      Definition : Node_
    END;
  ...
  VoidLiteralRecord =
    RECORD
    END;
  AccumulationListRecord =
    RECORD
      Elements : SequenceType
    END;
  ...
  WhenConditionRecord =
    RECORD
      Elements : SequenceType
    END;
  CharacterLiteralRecord =
    RECORD
      LexicalValue : StringPointer
    END;
  ...
  ValueIdentifierRecord =
    RECORD
      LexicalValue : StringPointer;
      DefiningOccurrence : Node_
    END;
  LargestVariant = IterationRecord;
  FlagNumberType = 0..15;
  NodeCell =
    RECORD
      Flags : SET OF FlagNumberType;
      Position : WORD;
      Parent : Node_;
      Comment : LineBufferType;
      CASE NodeType : NodeClass OF
```

```
        Grammar.Empty: (Empty : EmptyRecord);
        ...
        ValueIdentifier: (ValueIdentifier : ValueIdentifierRecord)
    END;
  STRING4 = STRING[4];

VAR Print : PROCEDURE (x : Node_);
    theUnattachedEmptyNode : Node_;

FUNCTION TestFlag(FlagNumber : FlagNumberType; x : Node_) : BOOLEAN;
PROCEDURE SetFlag(FlagNumber : FlagNumberType; x : Node_);
PROCEDURE ClearFlag(FlagNumber : FlagNumberType; x : Node_);
FUNCTION AddressString(x : Node_) : STRING4;
FUNCTION ConstructCopy(x : Node_) : Node_;
FUNCTION DenoterlessCopy(x : Node_) : Node_;
FUNCTION TestEquality(x, y : Node_) : BOOLEAN;
FUNCTION TestLess(x, y : Node_) : BOOLEAN;
PROCEDURE Attach(aParent : Node_; aPosition : INTEGER; aChild : Node_);
PROCEDURE Reattach(aParent : Node_; aPosition : INTEGER; aChild : Node_);
PROCEDURE Unattach(x : Node_);
FUNCTION GetNthChild(aParent : Node_; aPosition : INTEGER) : Node_;
FUNCTION GetComponent1(aConstruction : Node_) : Node_;
FUNCTION GetComponent2(aConstruction : Node_) : Node_;
FUNCTION GetComponent3(aConstruction : Node_) : Node_;
FUNCTION GetComponent4(aConstruction : Node_) : Node_;
FUNCTION Allocate(aNodeClass : NodeClass) : Node_;
PROCEDURE Destruct(VAR x : Node_);
PROCEDURE Deattribute(x : Node_);
FUNCTION Construct0
           (aNoChildConstruction : NodeClass;
            aComment : LineBufferType) : Node_;
FUNCTION Construct1
           (aOneChildConstruction : NodeClass;
            x1 : Node_;
            aComment : LineBufferType) : Node_;
FUNCTION Construct2
           (aTwoChildConstruction : NodeClass;
            x1, x2 : Node_;
            aComment : LineBufferType) : Node_;
FUNCTION Construct3
           (aThreeChildConstruction : NodeClass;
            x1, x2, x3 : Node_;
            aComment : LineBufferType) : Node_;
FUNCTION Construct4
           (aFourChildConstruction : NodeClass;
            x1, x2, x3, x4 : Node_;
            aComment : LineBufferType) : Node_;
```

```
FUNCTION ConstructList(aList : NodeClass;
                       aComment : LineBufferType) : Node_;
PROCEDURE ExpandList(aList : Node_; aPosition, anAmount : INTEGER);
PROCEDURE AppendElement(aList, anElement : Node_);
FUNCTION SelectElement(aSequence : SequenceType;
                       n : WORD) : ElementPointer;
FUNCTION ConstructTypeIdentifier(aLexicalValue : STRING) : Node_;
FUNCTION ConstructValueIdentifier(aLexicalValue : STRING) : Node_;
FUNCTION ConstructIntegerLiteral(aLexicalValue : STRING) : Node_;
FUNCTION ConstructRealLiteral(aLexicalValue : STRING) : Node_;
FUNCTION ConstructStringLiteral(aLexicalValue : STRING) : Node_;
FUNCTION ConstructCharacterLiteral(aLexicalValue : STRING) : Node_;
FUNCTION GetLexicalValue(aLexeme : Node_) : StringPointer;
FUNCTION GetInteger(anIntegerLiteral : Node_) : LONGINT;
FUNCTION GetReal(aRealLiteral : Node_) : REAL;
FUNCTION GetCharacter(aCharacterLiteral : Node_) : CHAR;
PROCEDURE SetDefiningOccurrence(id, theDefiningOccurrence : Node_);
FUNCTION GetDefiningOccurrence(id : Node_) : Node_;
PROCEDURE SetDefinition(x, theDefinition : Node_);
FUNCTION GetDefinition(x : Node_) : Node_;
PROCEDURE SetSort(aValue, aType : Node_);
FUNCTION GetSort(aValue : Node_) : Node_;
PROCEDURE ShowNode(x : Node_);

CONST BaseSize = SizeOf(NodeCell) - SizeOf(LargestVariant);

VAR TouchMonitor : PROCEDURE (theRoot : Node_);

IMPLEMENTATION ...
END.
```

MPS1 provides primitive facilities for representing and manipulating nodes. However, it is not intended for general use because its facilities are low-level. As such, it declares the type Node_ with an underscore to represent low-level nodes.

A node, then, is simply a ShortPointer, which can be converted to a NodePointer using NodePointer(PTR(theNode,0)). A NodePointer points at a NodeCell, which is a variant record that includes the following: a set of 16 Flags, a Position, a Parent, a Comment, a tag called NodeType, and a variant part for each possible node class. The head of the NodeCell record is common to all node classes and the tail is specific to each node class.

For each node-class x, a variant field also named x is defined when NodeType is x; this field is of a record type named xRecord, the components of which are class dependent. In general, for a construction, a component of type Node_ is declared for each construction component, with a name derived from the component name; for a list, a component of type SequenceType named Elements is declared; and for a lexeme, a component of type StringPointer named LexicalValue is declared.

Additional components are declared in some xRecords to store the semantic attributes defining-occurrence, type, and definition. They are named DefiningOccurrence, Sort, or Definition, respectively. Identifiers have a DefiningOccurrence component and expressions can, but do not necessarily, have a Sort component and a Definition component.

The following facilities are provided to support the manipulation of nodes. First, a low-level print routine is provided by the procedure variable Print. It is originally set to the procedure ShowNode, which simply displays the contents of a NodeCell record. However, if Unparser is used, the original Print is replaced by a more sophisticated printer, which prints a node and its children according to Acer's concrete syntax.

The variable theUnattachedEmptyNode contains a special unattached empty node that has itself as its parent. This node is the parent of every unattached node, including itself.

To reduce dependence on the particular data structure used in MPS1, various access routines are provided. For example, TestFlag, SetFlag, and ClearFlag are used to access the Flags field of a node.

AddressString yields a four-character hexadecimal representation of a node viewed as a ShortPointer. The PCAcer environment uses these as node identity numbers.

ConstructCopy produces a copy of the entire tree rooted at the given node; when a denoter is copied, a new denoter is created with the same node for its definition as the original. DenoterlessCopy is similar but each copied type-denoter is replaced by an any-type and each copied value-denoter is replaced by a void-literal.

Testing for structural equality and lexicographic order is supported by TestEquality and TestLess.

The primitive functions Attach, Reattach, and Unattach are provided for building and editing nodes. Attach takes a parent node, a position, and an unattached child and attaches the child at the specified position of the parent. Reattach is the same but the child may be attached, in which case it is removed from its context. And Unattach simply detaches a child, i.e., it sets the position to 0 and the parent to theUnattachedEmptyNode.

GetNthChild selects a child at a given position and the GetComponentn functions serve the same purpose but only for construction nodes.

The ability to create and free nodes is provided by Allocate and Destruct. Note that destructed nodes are not freed to the heap, they are stored for later reuse by Allocate. Deattribute removes all semantic attributes associated with a node and any of its children.

There are other ways than `Allocate` to construct a node. In particular, a construction node can be created, given appropriate unattached children, using one of the `Constructn` functions. Similarly, an empty list can be created by `ConstructList`; the `Elements` fields of such a list can then be modified by `ExpandList` to permit children to be `Attached`. `ExpandList` specifies the position at which to expand and by how much. As well, an element can be appended to a list using `AppendElement`.

Elements of a list are accessed as follows. `SelectElement`, given the `Elements` field of a list node and a position n, yields an `ElementPointer`, i.e., a pointer at a node. With this function, the nth element of a list can be either assigned to or simply examined.

For constructing a lexeme from its spelling, various `ConstructLexeme` functions are provided. Also, for accessing the spelling of a lexeme, `GetLexicalValue` is provided. The special functions `GetInteger`, `GetReal`, and `GetCharacter` are provided for accessing integer-, real-, and character-literal values respectively.

Finally, functions for `Getting` and `Setting` the semantic attributes defining-occurrence, definition, and type are provided.

`MPS1` uses a number of implementation techniques that are of interest and must be understood to explain the purpose of `TouchMonitor`.

To begin with, one of the goals of `MPS1` is to support the maintenance of valid semantic attributes in the presence of editing. In general, whenever a node modification takes place, all nodes within the tree will have invalid semantic attributes since they are all interdependent. Therefore, one approach for maintaining attributes would be to simply traverse the entire tree and remove all attributes. However, this would make a simple edit an operation that takes $O(n)$ time, where n is the number of nodes in the tree. This is unacceptable when such an edit can be done in constant time in the absence of attributes.

The approach used in `MPS1` has a more acceptable overhead. To outline the approach, assume that every node has a flag, let's call it `ModifiedQ`, which starts out as `false`. Then, whenever an edit (e.g., `Attach`, `Reattach`, or `Unattach`) takes place at a given node, the `ModifiedQ` of that node and every node from it to the root is set to `true`. The overhead on editing is thus proportional to the depth of the tree. But, since syntax trees tend to get wider rather than deeper, editing is essentially still a constant-time operation.

Whenever an attribute lookup is applied for a given node (e.g., `GetSort`), all parents of the node are examined to see if any have a `true` `ModifiedQ`. If any are `true`, the stored attribute is invalid and is discarded. The overhead on attribute lookup is therefore also proportional to the depth of the node.

ModifiedQ becomes false only when an attribute is stored at a node. In this case, the tree is again traversed from the node to the root but on the way back down, for any node with a true ModifiedQ, the ModifiedQ of each of its children is set to true and the node's own ModifiedQ is set to false. This downward propagation of flags continues until the start node is reached. Thus, after an attribute is set for some node, every node between it and the root will have a false ModifiedQ. Hence, a subsequent lookup will simply return the stored attribute. Of course, other nodes in the tree may still have a true ModifiedQs but these flags will invalidate only the attributes within those subtrees and will also become false when attributes are stored within those subtrees. The overhead on attribute storage is potentially proportional to the number of nodes in the tree, but then the determination of the attribute itself is typically the dominating time factor anyway.

With the above approach, the various Get functions yield only valid attributes, and can discard any invalid attributes.

Now, another problem with attributes is that they are typically nodes that are automatically created in response to a query. Hence, when an edit is performed and an attribute is no longer valid, what is to become of it? Turbo Pascal does not support garbage collection,[2] so invalid attributes cannot simply continue to exist as they will eventually exhaust memory. Thus invalid attributes should somehow be automatically reclaimed.

To this end, the NodeCell record is used in the following tricky way. If the Parent field of a NodeCell is set to theUnattachedEmptyNode, we know that the Position field must be 0. Therefore, the Position field can be used for another purpose in this case. For this reason, when an unattached node is Set to be the attribute of some node, the Position field of the attribute is used to store the identity of the 'attribute parent.' Because of this, when the attribute of a node x is known to be invalid using the previous flagging technique and the attribute node indicates x to be its attribute owner, the attribute node can be Destructed.

It is important to realize that although attributes can be shared by many nodes and hence a Destructed attribute may still appear as an attribute of some other node this does not cause a problem. After all, there is at most one attribute owner so an attribute will only be Destructed once. In addition, a Destructed node is not actually returned to the heap, so it is still valid to access its Position field to check for ownership, even if the node is subsequently Allocated to serve a new purpose.

One final trick used in MPS1 is that the position of an unattached node can also be set

[2]Note that Acer does have garbage collection so attribute nodes may continue to exist until they become garbage.

to 1 (rather than 0). In this case, whenever part of that node is modified, the function stored in TouchMonitor is applied to the root node. This is how the PCAcer environment (and FileServer in particular) keeps track of whether a top-level fixed-value binding or binding-list is modified with respect to the version stored in the file system.

C.11 Keywords

```
UNIT Keywords; INTERFACE {$F+,O+,L-}

  PROCEDURE InstallKeyword(spelling : STRING; tokenCode : INTEGER);
  FUNCTION FindKeyword(VAR spelling : STRING) : INTEGER;
  PROCEDURE ShowKeywords;

IMPLEMENTATION ...
END.
```

Keywords provides a hash table in which keywords can be installed for efficient lookup. InstallKeyword, given a spelling and the *token-code* of the keyword represented by the spelling (i.e., the ORD of the TokenType as defined by LexicalAnalyzer.), installs the information in the hash table. A subsequent call to FindKeyword, with a particular spelling, yields the token-code associated with that spelling, or 0 if the spelling is not in the hash table.

ShowKeywords prints the contents of the hash table.

C.12 LexicalAnalyzer

```
UNIT LexicalAnalyzer; INTERFACE {$F+,O+,L-}

USES Grammar, Keywords, Line, LineBuffer, MPS1;

TYPE
  TokenType =
    (EOStoken,
     DotToken, ..., CommaToken,
     LetToken, ..., TryingToken,
     IntegerLiteralToken, ...,  StringLiteralToken,
     BadToken);
  CharacterGenerator = PROCEDURE (VAR Ch : CHAR);
  ErrorHandler = PROCEDURE (x, y : WORD; Reason : STRING);

CONST
  MinToken = EOStoken;
  MinPunctuationToken = DotToken;
  MaxPunctuationToken = CommaToken;
  MinKeywordToken = LetToken;
  MaxKeywordToken = TryingToken;
  MinLexicalToken = IntegerLiteralToken;
  MaxLexicalToken = StringLiteralToken;
  MaxToken = BadToken;
  MaxTokenString = 26;

VAR TokenString : ARRAY [TokenType] OF STRING[MaxTokenString + 1];
    CurrentLexeme : Node_;
    CurrentToken : TokenType;
    TokenPosition, TokenLine, CommentPosition : WORD;

PROCEDURE AcceptToken(token : TokenType; VAR Comment : LineBufferType);
PROCEDURE InitializeLexicalAnalyzer
          (NextChar : CharacterGenerator; Failure : ErrorHandler);
PROCEDURE FileInitializeLexicalAnalyzer(VAR InputFile : TEXT);
PROCEDURE ParseFailure(str : STRING);

IMPLEMENTATION ...
END.
```

LexicalAnalyzer supports the conversion of a stream of characters to a stream of tokens. TokenType is provided to classify tokens, which are grouped to reflect the categories punctuation, keyword and lexeme. The spelling of each token is stored in the array TokenString.

LexicalAnalyzer is initialized to begin analysis either by calling the procedure InitializeLexicalAnalyzer with a function that generates characters and a function that handles errors, or by calling FileInitializeLexicalAnalyzer with a TEXT file. If a parser detects an error during parsing, it can call ParseFailure to terminate analysis and produce an error message associated with the CurrentToken.

Once analysis begins, the interface variables contain information about the most recently analyzed token: CurrentToken contains the TokenType, CurrentLexeme contains a lexeme if CurrentToken is a lexical token, TokenPosition contains the column position, TokenLine contains the row position, and CommentPosition contains the column position at which the first associated comment starts.

A parser accepts a token by calling AcceptToken with the type of token expected; the associated comment is set and the interface variables are updated to reflect the new information about the next token. If CurrentToken is not of the specified type, LexicalAnalyzer calls the error handler function—for FileInitializeLexicalAnalyzer the error handler writes an error message to standard output but a programmer supplied handler may do what it pleases.

Recall from Chapter 5 that tokens have row and column positions by virtue of the fact that the stream of characters from which they are derived contains ASCII formatting characters that imply discrete lines of text. This position information is essential for producing informative error messages.

C.13 Parser

```
UNIT Parser;  INTERFACE {$F+,O+,L-}

USES LineBuffer, LexicalAnalyzer, MPS1;

TYPE Node = Node_;

FUNCTION Parse(VAR inputFile : TEXT) : Node;
PROCEDURE ParseLineBuffer(theLineBuffer : LineBufferType;
                          VAR ResultNode : Node;
                          VAR ErrorPositionX, ErrorPositionY : WORD;
                          VAR ErrorReason : STRING);
FUNCTION ParseStream(NextChar : CharacterGenerator;
                     Failure : ErrorHandler) : Node;

IMPLEMENTATION ...
END.
```

Parser provides three different parsing routines. Parse parses a node from a TEXT file; ParseStream parses a node given a function that generates characters and a function that handles errors; and ParseLineBuffer parses a node from a LineBuffer, it sets the resulting node to ResultNode and if an error occurs, the error information (i.e., the position and error message) is assigned to the remaining parameters. The three routines share a common implementation.

There is nothing very interesting about the implementation of Parser.

C.14 Unparser

```
UNIT Unparser; INTERFACE {$F+,O+,L-}

USES Index, Grammar, LexicalAnalyzer, Line, LineBuffer, MPS, MPS1;

TYPE
  UnparseState =
    RECORD
      vPosition, hPosition : INTEGER;
      Hold, Reserve : INTEGER;
      CurrentNode : Node;
      PreviousToken : TokenType;
      BreakPending : BOOLEAN;
      PendingHold : INTEGER;
      CASE Fitting : BOOLEAN OF FALSE: (Narrow : BOOLEAN)
    END;
  UnparseItem = ^ UnparseItemCell;
  UnparseItemCell =
    RECORD
      theToken : TokenType;
      theOwner : Node;
      hPosition : INTEGER;
      NthCommentLine : WORD
    END;
  BreakPrinter = PROCEDURE (VAR state : UnparseState);
  CommentLinePrinter = PROCEDURE (Nth : INTEGER; VAR state : UnparseState);
  TokenPrinter = PROCEDURE (token : TokenType; VAR state : UnparseState);
  NodeEmitter = PROCEDURE (x : Node; VAR state : UnparseState);

VAR LineWidth, DenoterCutOffWidth : INTEGER;
    PrintEmptyNodes : BOOLEAN;
    TheBreakPrinter : BreakPrinter;
    TheCommentLinePrinter : CommentLinePrinter;
    TheTokenPrinter : TokenPrinter;
    TheNodeEmitter : NodeEmitter;
    DelimitQ : ARRAY [TokenType] OF SET OF TokenType;
    ReserveForA : ARRAY [MinPunctuationToken..MaxKeywordToken] OF
                    1..MaxTokenString + 1;

CONST
  IndentStep = 2;
  HalfIndentStep = 1;

PROCEDURE OutputSyntagm(x : Node);
PROCEDURE PrintSyntagm(VAR f : TEXT; x : Node);
```

```
FUNCTION FitNode(x : Node; VAR state : UnparseState) : BOOLEAN;
PROCEDURE EmitNode(x : Node; VAR state : UnparseState);
PROCEDURE Emit(VAR state : UnparseState);
PROCEDURE EmitComment(Comment : LineBufferType; VAR state : UnparseState);
PROCEDURE Break(VAR state : UnparseState);

IMPLEMENTATION ...
END.
```

Unparser provides facilities for pretty-printing nodes. Function parameters are used for the fitting and printing.

A number of interface variables are declared. LineWidth is set to indicate the number of columns in which the formatted output should fit. (The actual output may be wider when there is insufficient space to print the node.) DenoterCutOffWidth is set to indicate the maximum length of a denoter's printed definition; if this length is exceeded, the definition is not printed. PrintEmptyNodes is set to indicate whether empty nodes should be printed as the keyword **nothing** or should be invisible. The array DelimitQ, for each type of token t, indicates the set of tokens that must be delimited by white space when they follow t. The array ReserveForA, for each type of token t, indicates how must space should be reserved at the end of a line in order that t will still fit.

The remaining interface variables are procedure variables for customizing the pretty printer. Each takes a state parameter of type UnparseState, which includes the following information. The current vertical and horizontal position vPosition and hPosition; the Hold, i.e., the amount of indentation to appear after the next line-break; the Reserve, i.e., the amount of space to be left unused at the end of the line; the CurrentNode that is being printed or fitted; the PreviousToken that was printed or fitted; the PendingBreak, i.e., whether a break is pending before the next token is printed; the PendingHold, i.e., the hold that is in effect for the pending break; the Fitting flag, which indicates whether the printer is testing if a node fits or is actually printing; and finally, if the printer is not fitting, Narrow indicates whether the entire CurrentNode fits on the current line. (BreakPending and PendingHold are used in conjunction with the printing of comment lines so that comments appear as a block, i.e., each % character is lined up in a column).

TheBreakPrinter is used by Unparser to print a line-break and to set up the indentation for the next line. Therefore, it should normally increment vPosition and set hPosition to Hold. TheBreakPrinter is not called during fitting.

TheCommentLinePrinter is used by Unparser to print the nth comment-line of the CurrentNode as indicated by state. Therefore, it should normally increment hPosition by the amount of space taken up by the % character and the comment-line itself. It is not called during fitting. (A node that is commented is always printed as narrow.)

`TheTokenPrinter` is used by `Unparser` to fit and print tokens. Normally it increments `hPosition` to indicate the amount of space used by the token and `DelimitQ` is consulted to determine if space should precede the token. Only when `Fitting` is `false` in `state`, should tokens actually be printed.

`TheNodeEmitter` over-rides the normal formatting of nodes. Initially `TheNodeEmitter` is set to be the same as `EmitNode`. However, it can be replaced with a procedure that does special formatting. This procedure can call `EmitNode` for cases to be handled as usual.

`FitNode`, `Emit`, `EmitNode`, `EmitComment`, and `Break` are used to implement node formatting. The details of how this is done will not be described. Instead, a typical node formatting routine is shown below:

```
PROCEDURE EmitDyadicMethodCall(VAR state : UnparseState);

 BEGIN
  WITH state DO
    BEGIN
     EmitToken(LCurlyBracToken, state);
     Hold := hPosition;
     INC(Reserve,
         System.Length
           (GetLexicalValue(GetComponent2(CurrentNode))^) + 1);
     TheNodeEmitter(GetComponent1(CurrentNode), state);
     DEC(Reserve,
         System.Length
           (GetLexicalValue(GetComponent2(CurrentNode))^) + 1);
     TheNodeEmitter(GetComponent2(CurrentNode), state);
     Break(state);
     IF NOT Fitting OR (hPosition + Reserve < LineWidth) THEN
       BEGIN
        INC(Reserve, ReserveForA[RCurlyBracToken]);
        TheNodeEmitter(GetComponent3(CurrentNode), state);
        DEC(Reserve, ReserveForA[RCurlyBracToken]);
        EmitToken(RCurlyBracToken, state);
        EmitComment(CommentOf(CurrentNode)^, state)
       END
    END
 END;
```

A programmer wishing to customize the formatting of a particular class of node will typically make a copy of such an existing routine and modify it.

The pretty printer could easily be modified to fit and print proportional characters just by using hPosition to indicate point sizes rather than simply column position. In other words, the units represented by hPosition, Hold, LineWidth, and so on, do not matter.

Note that the type UnparseItemCell is use by the unit TokenLine to represent lines of tokens. Each such token has a TokenType, an owner node, and a starting hPosition. In addition, if the token is a comment-line, NthCommentLine indicates which line of theOwner the token stands for.

C.15 Makers

```
UNIT Makers;  INTERFACE {$F+,O+,L-}

USES MPS;

FUNCTION MakeAccumulation(x1, x2 : Node) : Node;
...
FUNCTION MakeVoidLiteral : Node;

IMPLEMENTATION ...
END.
```

Makers provides functions for constructing each class of construction node from the appropriate number and type of children. (If an attached child is provided, a copy is automatically produced.)

C.16 Access

```
UNIT Access; INTERFACE {$F+,O+,L-}

USES MPS;

FUNCTION AccumulationQ(x : Node) : BOOLEAN;
...
FUNCTION WhenConditionQ(x : Node) : BOOLEAN;
FUNCTION AccumulatorOf(x : Node) : Node;
...
FUNCTION VariantsOf(x : Node) : Node;

IMPLEMENTATION ...
END.
```

Access provides two sets of functions: a recognizer for each class and category of node; and a selector for each construction component name. The `xQ` functions yield `true` if their argument node is of the corresponding class or category; the `xOf` functions yield the `x` component of their construction argument.

C.17 MPS

```
UNIT MPS;  INTERFACE {$F+,O+,L-}

USES Grammar, Keywords, MPS1, LineBuffer;

TYPE Node = Node_;

VAR theUnattachedEmptyNode : Node;

FUNCTION NodeType(x : Node) : NodeClass;
PROCEDURE ContextDomain(x : Node; VAR Result : NodeClassSet);
FUNCTION DeletableContextQ(x : Node) : BOOLEAN;
FUNCTION EmptyQ(x : Node) : BOOLEAN;
FUNCTION LexemeQ(x : Node) : BOOLEAN;
FUNCTION ListQ(x : Node) : BOOLEAN;
FUNCTION ConstructionQ(x : Node) : BOOLEAN;
FUNCTION AttributeQ(x : Node) : BOOLEAN;
FUNCTION Parent(x : Node) : Node;
FUNCTION AttributeParent(x : Node) : Node;
FUNCTION Root(x : Node) : Node;
FUNCTION AttributeRoot(x : Node) : Node;
FUNCTION Depth(x : Node) : WORD;
FUNCTION FirstLexeme(x : Node) : Node;
FUNCTION StructurallyEqualQ(x, y : Node) : BOOLEAN;
FUNCTION StructurallyLessQ(x, y : Node) : BOOLEAN;
FUNCTION EnclosesQ(x, y : Node) : BOOLEAN;
FUNCTION ImproperlyEnclosesQ(x, y : Node) : BOOLEAN;
FUNCTION AttributeEnclosesQ(x, y : Node) : BOOLEAN;
FUNCTION ContainsQ(theLeftNode, theRightNode,
                   theTargetNode : Node) : BOOLEAN;
FUNCTION Enclosing(AncestorDomain : NodeClassSet; x : Node) : Node;
FUNCTION CommonAncestor(x, y : Node) : Node;
FUNCTION CommentOf(x : Node) : LineBufferPointer;
PROCEDURE AppendComment(x : Node; Comment : LineBufferType);
PROCEDURE ReplaceComment(x : Node; NewComment : LineBufferType);
FUNCTION FirstElement(list : Node) : Node;
FUNCTION NthElement(list : Node; n : INTEGER) : Node;
FUNCTION FirstComponent(aConstruction : Node) : Node;
FUNCTION NthComponent(aConstruction : Node; n : INTEGER) : Node;
FUNCTION FirstChild(aParent : Node) : Node;
FUNCTION NthChild(aParent : Node; n : INTEGER) : Node;
FUNCTION NumberOfChildrenOf(x : Node) : INTEGER;
FUNCTION Position(x : Node) : INTEGER;
FUNCTION Previous(x : Node) : Node;
FUNCTION Next(x : Node) : Node;
FUNCTION Copy(x : Node) : Node;
PROCEDURE Replace(x1, x2 : Node);
PROCEDURE ReplaceAndSet(VAR x1 : Node; x2 : Node);
PROCEDURE Exchange(x1, x2 : Node);
PROCEDURE Delete(x : Node);
PROCEDURE Insert(list : Node; n : INTEGER; elem : Node);
PROCEDURE TailInsert(list : Node; elem : Node);
PROCEDURE Splice(list : Node; n : INTEGER; SubList : Node);
FUNCTION Concat(elem, list : Node) : Node;
FUNCTION Append1(list, elem : Node) : Node;
FUNCTION Append(list1, list2 : Node) : Node;
FUNCTION SubList(list : Node; n1, n2 : INTEGER) : Node;
FUNCTION NullList(kind : NodeClass) : Node;
FUNCTION List1(kind : NodeClass; e1 : Node) : Node;
FUNCTION List2(kind : NodeClass; e1, e2 : Node) : Node;
FUNCTION List3(kind : NodeClass; e1, e2, e3 : Node) : Node;
FUNCTION List4(kind : NodeClass; e1, e2, e3, e4 : Node) : Node;
FUNCTION List5(kind : NodeClass; e1, e2, e3, e4, e5 : Node) : Node;
FUNCTION MakeLexeme(k : NodeClass; aLexicalValue : STRING) : Node;
FUNCTION MakeTypeIdentifier(aLexicalValue : STRING) : Node;
FUNCTION MakeValueIdentifier(aLexicalValue : STRING) : Node;
FUNCTION MakeIntegerLiteral(aLexicalValue : STRING) : Node;
FUNCTION MakeRealLiteral(aLexicalValue : STRING) : Node;
FUNCTION MakeStringLiteral(aLexicalValue : STRING) : Node;
FUNCTION MakeCharacterLiteral(aLexicalValue : STRING) : Node;
FUNCTION StringOf(aLexeme : Node) : STRING;
FUNCTION IntegerOf(anIntegerLiteral : Node) : LONGINT;
FUNCTION RealOf(aRealLiteral : Node) : REAL;
FUNCTION CharacterOf(aCharacterLiteral : Node) : CHAR;
FUNCTION BuildIntegerLiteral(x : LONGINT) : Node;
FUNCTION BuildRealLiteral(x : REAL) : Node;
FUNCTION BuildCharacterLiteral(x : CHAR) : Node;

IMPLEMENTATION ...
END.
```

MPS provides the primary metaprogramming system abstraction. It declares the type Node and the special node call theUnattachedEmptyNode.

NodeType determines the NodeClass of a node. ContextDomain determines the set of

NodeClasses that may appear in the context at which the argument node is attached. DeletableContextQ determines if its argument can be Deleted from its context. EmptyQ, LexemeQ, ListQ, and ConstructionQ are recognizers that behave as implied by their names.

AttributeQ determines if a node is an attribute node. Recall that an unattached node used as a semantic attribute has its position field specially marked to indicate its attribute parent. Therefore, AttributeQ determines if the position field of the root of its argument is so marked.

Parent determines the parent of a node; theUnattachedEmptyNode is yielded if the argument node is unattached. Similarly, AttributeParent determines the attribute parent of the root of the argument node and it too yields theUnattachedEmptyNode if AttributeQ is false for its argument.

Root determines the root node of a node by repeated application of Parent until the parent is empty. Similarly, AttributeRoot determines the attribute root of a node by repeated application of AttributeParent until the AttributeParent is empty.

Depth determines the depth of a node with respect to its root. FirstLexeme determines the first lexeme reached by an pre-order traversal. StructuallyEqualQ and StructurallyLessQ test for structural equivalence and lexicographic order.

EnclosesQ determines whether the node x contains the node y as a descendant. ImproperlyEnclosesQ does the same but yields true when x and y are the same node. AttributeEnclosesQ determines if x can be reached from y by repeated applications of AttributeParent.

ContainsQ determines if theLeftNode, theRightNode, or some node in between ImproperlyEnclosesQ theTargetNode Therefore, theLeftNode and theRightNode must be children of the same construction or list node and theLeftNode must appear before theRightNode.

Enclosing applies Parent until the parent's class is a member of AncestorDomain. CommonAncestor determines the node with largest depth that has both x and y as a descendant.

CommentOf yields a pointer to a line-buffer. This can be used both to access and to update the comment field of a node. A line-buffer can be appended to the existing comment using AppendComment. ReplaceComment replaces the comment and destroys the old comment.

FirstElement yields the first child of a list node, unless the list is empty, in which case it yields theUnattachedEmptyNode. NthElement yields the nth child of a list node. The argument n may be negative, in which case children are accessed right to left (i.e., the

-1 child is the last child). FirstComponent and NthComponent behave similarly but for constructions. And FirstChild and NthChild also behave similarly but apply for both lists and constructions.

NumberOfChildren determines the number of children of a node; lexemes have 0 children. Position determines the position of a node; an unattached node has position 0. Previous and Next determine the previous or next node with respect to the context; they yield theUnattachedEmptyNode when there is no previous or next node.

Copy creates a copy of a node, i.e., a node that is StructurallyEqualQ but is unattached.

Replace replaces x1 in context with x2. The node x1 must be attached and if x2 is attached it is copied. The class of x2 must be a member of the ContextDomain of x1. ReplaceAndSet does the same thing but sets x1 to be x2.

Exchange exchanges two nodes in context. Each node's class must be a member of the other's ContextDomain. Both nodes must be attached.

Delete deletes a node. The node must be attached and in a deletable context. Deleting a node that is an optional construction component has the effect of replacing the node with a new empty node.

Insert inserts an element at the specified position of a list. Just as for NthChild, the position may be negative to specify right to left order. After the insertion the element will be at the indicated position. TailInsert is like Insert with n set to -1.

Splice inserts copies of the elements of the sublist at the indicated position of the list. Concat creates a new list from an element and a list by including the element as the first element of the list. Append1 is similar but the element is put at the end. Append creates a new combined list from two existing lists. Sublist creates a new list from a subrange of elements of an existing list.

NullList creates an empty list of the specified class and the Listn functions create lists of the indicated length, given a class and appropriate elements.

MakeLexeme creates a lexeme of the given class and with the given spelling. Individual MakeX functions for each class of lexeme are provided as well. StringOf yields the spelling of a lexeme. IntegerOf, RealOf, and CharacterOf yield the Pascal value of an integer-, real-, or character-literal, respectively. Similarly, BuildIntegerLiteral, BuildRealLiteral, and BuildCharacterLiteral build the appropriate class of lexeme given its Pascal value.

This completes the interface for manipulating objects of Acer's context-free syntax. All these routines are implemented in terms of the data structures provided by MPS1.

C.18 FileServer

```
UNIT FileServer;  INTERFACE {$F+,O+,L-}

USES Manager, Index, DOS, LineBuffer, MPS;

CONST BackupLevel = 2;

PROCEDURE Store(x : Node);
FUNCTION FetchDeclaration(Name : STRING) : Node;
FUNCTION FetchBinding(Name : STRING) : Node;
FUNCTION RestoreBinding(Name : STRING; VAR OldBinding : Node) : Node;
PROCEDURE StoreClosure(Name : STRING; theClosure : Node);
FUNCTION FetchClosure(Name : STRING) : Node;
PROCEDURE StoreInstructions
            (Name : STRING; theInstructions, theClosure : Node);
FUNCTION FetchInstructions(Name : STRING; theClosure : Node) : Node;
PROCEDURE StoreCode(theCluster : Node; theSize : WORD; theCode : POINTER);
FUNCTION FetchCode(Name : STRING; VAR theCluster : Node) : ShortPointer;
PROCEDURE StoreExecutable(theCluster : Node;
                          theSize : WORD; theCode : POINTER);
FUNCTION FetchExecutable(Name : STRING;
                        VAR theCluster : Node) : ShortPointer;
FUNCTION FindBindingFile(IdentifierName : STRING;
                         VAR f : FILE) : BOOLEAN;
FUNCTION BindingFileTime(IdentifierName : STRING) : LONGINT;
FUNCTION FindDeclarationFile(IdentifierName : STRING;
                             VAR f : FILE) : BOOLEAN;
FUNCTION DeclarationFileTime(IdentifierName : STRING) : LONGINT;
FUNCTION FindCodeFile(IdentifierName : STRING; VAR f : FILE) : BOOLEAN;
FUNCTION CodeFileTime(IdentifierName : STRING) : LONGINT;
FUNCTION FindInstructionFile(IdentifierName : STRING;
                             VAR f : FILE) : BOOLEAN;
FUNCTION InstructionFileTime(IdentifierName : STRING) : LONGINT;
FUNCTION FindExecutableFile(IdentifierName : STRING;
                            VAR f : FILE) : BOOLEAN;
FUNCTION ExecutableFileTime(IdentifierName : STRING) : LONGINT;
FUNCTION FindClosureFile(IdentifierName : STRING; VAR f : FILE) : BOOLEAN;
FUNCTION ClosureFileTime(IdentifierName : STRING) : LONGINT;
FUNCTION FetchFullName(VAR f : FILE) : STRING;
PROCEDURE CheckPointNode(VAR f : FILE; theNode : Node);
FUNCTION RestoreNode(VAR f : FILE) : Node;
PROCEDURE CheckPointLineBuffer(VAR f : FILE;
                               theLineBuffer : LineBufferType);
FUNCTION RestoreLineBuffer(VAR f : FILE) : LineBufferType;
FUNCTION FileTime(VAR f) : LONGINT;
FUNCTION HasBackup(Name : STRING) : BOOLEAN;
PROCEDURE WritePath(VAR theFile : FILE; theNode : Node);
FUNCTION ReadPath(VAR theFile : FILE; theNode : Node) : Node;
PROCEDURE ShowExternals;

VAR ConformingDeclarationQ : FUNCTION (d1, d2 : Node) : BOOLEAN;
    Externals : Index.T;
    oDir, aDir, dpDir, axeDir, vdDir, bakdir : PathStr;

IMPLEMENTATION ...
END.
```

FileServer provides high-level access to the file system for storing, fetching, backing up, and restoring nodes and various associated objects. The following file-naming conventions are followed. Fixed-value-bindings and top-level binding-lists are stored in '.vb' files and type-bindings are stored in '.tb' files—the file name is derived from the first 8 character of the binding's name. Backups are stored in '.vb1,' '.tb1,' '.vb2,' and '.tb2' files, where the value of the constant BackupLevel indicates the number of backups to be maintained. Closure information about fixed-value-bindings and binding-lists is stored in '.vdp' files and closure information about type-bindings is stored in '.tdp' files. Fixed-value-declarations are stored in '.vd' files. Assembler code is stored in '.a' files. Linkables are stored in '.o' files. And executables are stored in '.axe' files.

Searching for objects is guided by the MS-DOS environment variable ACERINPUT, which should be set to be something as follows:

```
    set ACERINPUT=.;C:\acer\lib;.\vd;.\dp;.\o;.\axe;.\a
```

Accordingly, the storing of declarations, closures, backups, executables, linkables, and assembler code is such that if the current directory contains a 'vd,' 'dp,' 'bak,' 'axe,' 'o,' or 'a,' respectively, the object is stored in that directory rather than the current directory. Furthermore, if a previous version of an object can be found in a directory other than the current directory using the search path, the new version is stored in the same directory as the previous version.

FileServer's main procedure is Store, which takes a node x and stores it as follows. Of course, the node x must be a valid top-level construct, that is, a fixed-value-binding, a binding-list containing a fixed-value-binding (the first one of which is called the *primary unit*), a type-binding, or a fixed-value-declaration. How Store works depends on the class of node.

If x is a fixed-value-binding, the binding is stored in *checkpointed* form in a '.vb' file. The checkpointed form is a compacted syntactic representation, that is, the node's class is encoded as a byte which is followed by either the spelling, if it is a lexeme, or the number of children and the children themselves, if it is a list or construction. Old versions of the '.vb' file are backed up.

Similarly, if x is a binding-list, each fixed-value-binding is stored in checkpointed form in a '.vb' file. Actually, the file for the primary unit contains the checkpointed binding-list and the files for the remaining fixed-value-bindings simply refer to the primary file.

If x is a type-binding, it is stored just like a fixed-value-binding but in a '.tb' file. A type-binding node becomes an attribute after it is stored. Old versions of the '.tb' file are backed up.

If x is a fixed-value-declaration, it is stored in a '.vd' file. Just as for a type-binding, it thereafter becomes an attribute node.

FileServer keeps an index, Externals, of mappings from StringPointers to type-bindings and fixed-value-declarations currently in memory. Store therefore has the effect of including additional associations in this index. FileServer also keeps track of whether a fixed-value-binding or binding-list is modified with respect to its stored version; it uses MPS1's TouchMonitor facility for this.

FetchDeclaration, given the spelling of a value-identifier, searches for the fixed-value-declaration associated with that spelling, which it loads and returns as its result. If a declaration cannot be found, theUnattachedEmptyNode is returned. The declaration is either loaded from the file system or found in Externals.

Similarly, FetchBinding, given the spelling of a value-identifier, searches for the fixed-value-binding (or the binding-list containing the fixed-value-binding) associated with that spelling, which it then loads and returns as its result.

RestoreBinding is similar to FetchBinding but the '.vb' or '.tb' file(s) are removed after loading and the backup versions are renamed so that the latest backup again becomes the current version.

StoreClosure is given the spelling of an identifier and a closure, which is represented as an arbitrary-list of identifiers, and it stores the closure in checkpointed form in a '.vdp' or '.tdp' file, depending on whether the spelling is that of a value-identifier or a type-identifier. FetchClosure is the inverse in that, given a spelling, it loads the associated closure; theUnattachedEmptyNode is returned if the closure is not stored.

StoreInstructions, given the spelling of a value-identifier, instructions represented as a code-patch, and a closure of the associated declaration file, stores the information in a '.a' file. FetchInstructions, given a spelling and the closure, retrieves the instructions—the closure is modified to reflect the closure stored in the '.a' file.

StoreCode is given a cluster represented as either an value-identifier or an arbitrary-list of value-identifiers, a size, and a Pointer at encoded instructions. Then, if the cluster is a value-identifier, StoreCode stores the encoded instructions in a '.o' file derived from the value-identifier. Otherwise, if the cluster is an arbitrary-list of identifiers, a '.o' file is created for each identifier, although only the first file contains the actual code with the remaining files simply referring back to the first file. (The notion of a cluster derives from the fact that a binding-list usually contains several fixed-value-bindings for which '.o' files must be created, and all these '.o' files are interdependent.) FetchCode, given the spelling of a value-identifier, returns a ShortPointer to the encoded instructions associated with the identifier; it also sets theCluster.

StoreExecutable and FetchExecutable are directly analogous to StoreCode and FetchCode but they refer to '.axe' files.

A series of functions are provided for Finding files and for determining the creation FileTime for files. A pair of such functions is provided for bindings, declarations, code, instructions, executables, and closures.

FetchFullName, given a File associated with a binding or declaration file, determines the complete spelling of the object stored in that file. (PCAcer uses this feature to make file names more readable.)

CheckPointNode checkpoints a node to a file and RestoreNode is its inverse. Similarly, CheckPointLineBuffer checkpoints a line-buffer to a file and RestoreLineBuffer is its inverse.

FileTime, given a file, returns its creation time.

HasBackup, given the spelling of a value- or type-identifier, determines if there is a '.vbl' or '.tbl' associated with that spelling.

WritePath, given a file and a node, stores in the file the information about how to reach the node from its root. This information is given in terms of a list of child positions that must be selected starting at the root. ReadPath is the inverse of WritePath. (PCAcer uses this feature to store the position of a node window's selection).

Finally, ConformingDeclarationQ is used to determine if a stored declaration is compatible with the previous version so that the creation time of the previous version can be used. The unit AST sets this variable with a function that tests whether the two declarations have equivalent types.

FileServer has a complex implementation but little would be gained by describing it in detail.

C.19 AST

```
UNIT AST;  INTERFACE {$F+,O+,L-}

USES Index, Access, Grammar, Makers, Line, LineBuffer, MPS, MPS1,
     Unparser, Parser;

FUNCTION ConstructorQ(x : Node) : BOOLEAN;
FUNCTION CompilationUnitQ(x : Node) : BOOLEAN;
FUNCTION PrimaryUnitOf(x : Node) : Node;
FUNCTION Global(id : STRING) : Node;
FUNCTION DefiningOccurrenceOf(id : Node) : Node;
FUNCTION DefinitionOf(x : Node) : Node;
FUNCTION DenotationOf(x : Node) : Node;
FUNCTION TypeOf(x : Node) : Node;
FUNCTION KindOf(x : Node) : Node;
FUNCTION SubTypeQ(t1, t2 : Node) : BOOLEAN;
FUNCTION EquivalentTypeQ(t1, t2 : Node) : BOOLEAN;
FUNCTION MaxType(t1, t2 : Node) : Node;
FUNCTION SameValuesQ(v1, v2 : Node) : BOOLEAN;
FUNCTION SubSignatureQ(x1, x2 : Node) : BOOLEAN;
FUNCTION ValidArgumentsQ(theArguments, theSignature : Node) : BOOLEAN;
FUNCTION InvalidArgumentOf(theArguments, theSignature : Node) : Node;
FUNCTION TagMatch(x, id : Node) : Node;
FUNCTION AbstractBaseOf(theType : Node) : Node;
FUNCTION QuantifierOf(theType : Node) : Node;
FUNCTION VariableIdentifierQ(id : Node) : BOOLEAN;
FUNCTION RecursiveTypeOperatorQ(x : Node) : BOOLEAN;
FUNCTION DefinitionCopy(y, At : Node; Substitutions : Index.T) : Node;
FUNCTION DefinitionCopyAt(theNode, At : Node) : Node;
FUNCTION ClosureOf(theNode : Node) : Index.T;
FUNCTION MakeDenoter(x : Node) : Node;
FUNCTION ErrorTypeQ(theType : Node) : BOOLEAN;
FUNCTION VoidTypeQ(theType : Node) : BOOLEAN;
FUNCTION BooleanTypeQ(theType : Node) : BOOLEAN;
FUNCTION StringTypeQ(theType : Node) : BOOLEAN;
FUNCTION CharacterTypeQ(theType : Node) : BOOLEAN;
FUNCTION RealTypeQ(theType : Node) : BOOLEAN;
FUNCTION IntegerTypeQ(theType : Node) : BOOLEAN;
FUNCTION RaiseTypeQ(theType : Node) : BOOLEAN;
FUNCTION ReferenceTypeQ(theType : Node) : BOOLEAN;
FUNCTION ReferenceBaseTypeOf(theType : Node) : Node;
FUNCTION PointerTypeQ(theType : Node) : BOOLEAN;
FUNCTION PointerBaseTypeOf(theType : Node) : Node;
FUNCTION ExceptionTypeQ(theType : Node) : BOOLEAN;
FUNCTION ExceptionBaseTypeOf(theType : Node) : Node;
FUNCTION ArrayTypeQ(theType : Node) : BOOLEAN;
FUNCTION ArrayBaseTypeOf(theType : Node) : Node;
FUNCTION AccumulatorTypeQ(theType : Node) : BOOLEAN;
FUNCTION AccumulatorBaseTypeOf(theType : Node) : Node;
FUNCTION AccumulatorResultTypeOf(theType : Node) : Node;
FUNCTION IteratorTypeQ(theType : Node) : BOOLEAN;
FUNCTION IteratorBaseTypeOf(theType : Node) : Node;

IMPLEMENTATION ...
END.
```

AST provides routines for manipulating Acer's context-dependent syntax. Its simple interface is an indication of the successful design of a simple context-dependent syntax for Acer

ConstructorQ determines if x is a constructor. CompilationUnitQ determines if x is a valid top-level construct. PrimaryUnitOf selects the first fixed-value-binding of a binding-list.

Global, given the spelling of an identifier, uses FileServer to determine the fixed-value-declaration or type-binding associated with that spelling.

DefiningOccurrenceOf, DefinitionOf, DenotationOf, TypeOf, and KindOf select the indicated semantic attribute.

SubtypeQ determines whether t1 is a subtype of t2. EquivalentTypeQ determines whether t1 is a subtype of t2 and t2 is a subtype of t1. MaxType yields the largest type (in terms of the partial order defined by SubtypeQ) of its two arguments—if either of its arguments is the error-type or the arguments are unrelated as subtypes it yields the error-type; only if both arguments are the raise-type does it yield the raise-type. SameValuesQ determines if the value v1 and the value v2 must always denote the same value at run-time. SubSignatureQ determines if the signature x1 is a subsignature of the signature x2.

ValidArgumentsQ determines if the argument-list theArguments is valid with respect to theSignature. InvalidArgumentOf yields the first argument of theArguments that is invalid with respect to theSignature.

TagMatch, given a value-identifier and a variant-type or variant-inspection, yields the branch associated with the identifier.

AbstractBaseOf and QuantifierOf select the abstract-base and quantifier, respectively, of an abstract-type.

VariableIdentierQ determines if id is introduced by a variable-value-declaration or a variable-value-binding. RecursiveTypeOperatorQ determines if x is a recursive type-operator. Recursive type-operators are invalid.

DefinitionCopy, given a node y to copy, a location At from which copying takes place (usually the parent of y), and an index of substitutions to be made during copying, returns a node that denotes the same object as the original; it is typically expressed using denoters. DefinitionCopyAt, given a node y and a location At at which y is to be expressed, creates a copy that denotes the same object as y but is expressed in terms of the scope at At without the use of denoters. (Actually, DefinitionCopyAt produces a copy of the denoted object if it cannot simply refer to the original.)

ClosureOf, given a node, yields an index representing the set of global identifiers used in the node. MakeDenoter makes a denoter; it determines whether to make a type-denoter or a value-denoter based on whether x is a type or a value. The remaining functions are simply convenient recognizers and selectors for dealing with standard types that are known to AST.

Several tricky techniques are used to implement the features of AST. For example, to determine that the denotation of *x* in

{**let** *x* **be** *y*; **let** *y* **be** *x*;

is *error*, DenotationOf must mark nodes so that recursive applications of DefinitionOf do not lead to an infinite loop. Thus, before asking for the DefinitionOf a node, the node is marked so that if it is reached again, DenotationOf detects the loop.

Determining the TypeOf *x* in the above case would also lead to an infinite loop so TypeOf must mark nodes as well. However, TypeOf must also handle recursive types such as the following:

let *x* **be tuple** *x* **end**

To handle such a case, when TypeOf is applied to a tuple-literal, it associates with the tuple-literal the empty tuple-type **Tuple end** before proceeding to determine the type of each argument in the literal. By doing so, when the TypeOf x is recursively determined, it turns out to be the very tuple-type associated with the tuple-literal. After recursively determining this, the tuple-type is modified by inserting an anonymous declaration with a type that refers to the tuple-type using a type-denoter, e.g.,

Tuple :{[]} **end**

All recursive types for constructors are handled in this way, that is, by associating with the literal a partially determined type before proceeding to determined the types of the literal's components.

Note also that if `TypeOf` starts at the defined-identifier x, it cannot simply mark x so that if x is reached again, a loop is detected because this is precisely what would happen: the first x would be marked, then the type of the tuple-literal would be determined, which would recursively request the type of x, which would now be marked. Thus for `TypeOf`, a node must be marked and reached *twice* before a loop is detected.

In Acer's definition manual, subtyping is defined in terms of `DefinitionCopy` but this is not an efficient implementation strategy since it involves the creation of massive numbers of nodes. Instead, `AST` keeps an index of substitutions that are in effect during subtyping so that copying can be avoided.

Furthermore, `SubType` must be prevented from going into an infinite loop. For example, during recursive subtyping of components of the two types *T1* and *T2* in

{**let** *T1* **be Tuple** :*T1* **end**;
let *T2* **be Tuple** :*T2* **end**;

`SubType` goes into a loop. To circumvent this problem, `Subtype` assumes that the two tuple-types are valid subtypes while recursively subtyping their components. As a result, instead of going into a loop, `SubType` detects that the components are subtypes by assumption. Of course, if subtyping fails for some other reason, the assumption will be proven invalid and is nullified. Using these techniques, `SubType` can efficiently determine whether two types are related as subtypes even for recursive types.

C.20 AST2

```
UNIT AST2;  INTERFACE {$F+,O+,L-}

USES Index, MPS;

FUNCTION ClosedTypeQ(theType : Node) : BOOLEAN;
FUNCTION Validate(x : Node) : Index.T;
PROCEDURE FindFirstError
           (x : Node; VAR ErrorNode : Node; VAR ErrorMessage : STRING);
```

```
IMPLEMENTATION ...
END.
```

AST2 is provided for testing whether a node conforms with all context-dependent constraints. One of its main responsibilities is to detect invalid delayed-occurrences.

ClosedTypeQ determines whether a type can be expressed with global scope. It is used to restrict the type used as a tag of a dynamic.

Validate, given a node, yields an index containing mappings from nodes to error messages. Validate detects all errors. Similarly, FindFirstError given a node, checks it for errors but it yields the first erroneous node, along with its error message, that it finds.

No special implementation tricks are used in this unit.

C.21 Compile

```
UNIT Compile; INTERFACE {$F+,O+,L-}

USES Index, Access, Makers, Grammar, AST, LineBuffer, Line, Unparser,
     Parser, MPS, MPS1;

TYPE
  Registers =
    (A0, A1, A2, A3, A4, A5, A6, A7,
     D0, D1, D2, D3, D4, D5, D6, D7, NULL);
  EffectiveAddressModes =
    (Immediate, Direct, Indirect, MemoryIndirect, PreDecrement,
     PostIncrement, Label_, Special);
  InstructionKinds = (Lea, Jsr, Bz, Bnz, Bra, Rtd, Trap, Move, Lbl);

CONST
  RegisterName : ARRAY [A0..NULL] OF STRING[5] =
    ('a0', 'a1', 'a2', 'a3', 'a4', 'a5', 'a6', 'a7',
     'd0', 'd1', 'd2', 'd3', 'd4', 'd5', 'd6', 'd7', 'NULL');
  HeaderSize = 5;

FUNCTION Translate(x : Node; theClosure : Node) : Node;
FUNCTION Linker(UnitName : STRING; VAR theCluster : Node) : Node;
FUNCTION AddressModeType(theEffectiveAddress : Node) :
           EffectiveAddressModes;
FUNCTION InstructionType(theInstruction : Node) : InstructionKinds;
FUNCTION RegisterType(theRegister : Node) : Registers;

IMPLEMENTATION ...
```

```
END.
```

`Compile` provides routines for translating a fixed-value-binding or binding-list to a code-patch. For PCAcer, such a code-patch expresses a simple form of assembly language that is executable on a simulated machine. It is a greatly simplified form of MC680x0 code. Hence, the abstract machine provides registers A0 through A7 and D0 through D7, and supports the address modes: `Immediate`, `Direct`, `Indirect`, `MemoryIndirect`, `PreDecrement`, `PostIncrement`, `Label`, and `Special`. The machine provides only the instructions with the names: `Lea` (load effective address), `Jsr` (jump to subroutine), `Bz` (branch on zero), `Bnz` (branch on not zero), `Rtd` (return from subroutine), `Trap` (call a numbered routine coded in Turbo Pascal), and `Move` (move a four byte word from one place to another). All additional operations, such as addition, are provided as traps to Pascal routines. They are set up by assigning to the `TrapProcedures` array of the `Run` unit.

The details of the translation will not be described as they are fairly straightforward and would require a complete description of the abstract machine and its instruction set. Only the interface is described.

`Translate`, given a fixed-value-binding or bindinging-list and its closure, yields a code-patch representing the translation. This translation can be stored in a '.a' file.

`Linker`, given the spelling of a value-identifier, yields a code-patch that modifies the associated translation to include the instructions for setting up the run-time closure required to execute the instructions; `theCluster` of identifiers associated with the translation is set as well. The resulting code-patch can then be encoded and stored as a '.o' file.

`AddressModeType`, `InstructionType`, and `RegisterType` are used by the unit `Link` to analyze a code-patch so as to encode it in compacted form. The small number of instructions allows for a very compact encoding of '.o' and '.axe' files.

C.22 Run

```
UNIT Run; INTERFACE {$F+,O+,L-}

USES Dos, Manager, Compile;

CONST
  HeaderSize = 5;
  StackSize = 16 * 1024;
  NumberOfTraps = 50;
```

```
TYPE
  PointerPointer = ^ POINTER;
  BytePointer = ^ BYTE;
  ShortIntPointer = ^ SHORTINT;
  IntegerPointer = ^ INTEGER;
  WordPointer = ^ WORD;
  LongIntPointer = ^ LONGINT;
  RealPointer = ^ REAL;
  AcerFlag = (Traced, Marked, Coded);
  AcerValuePointer = ^ AcerValueType;
  AcerValueType =
    RECORD
      Size : WORD;
      Next : ShortPointer;
      Flags : SET OF AcerFlag;
      CASE WORD OF
        0: (theBytes : ARRAY [0..65520] OF BYTE);
        1: (theWords : ARRAY [0..32760] OF WORD);
        2: (thePointers : ARRAY [0..16380] OF POINTER)
    END;

VAR theRegisters : ARRAY [A0..D7] OF AcerValuePointer;
    theProgramCounter : AcerValuePointer;
    TrapProcedures : ARRAY [0..NumberOfTraps] OF PROCEDURE;
    theStack : ShortPointer;

FUNCTION AllocateTraced(SizeInBytes : WORD) : ShortPointer;
FUNCTION AllocateUntraced(SizeInBytes : WORD) : ShortPointer;
FUNCTION AllocateCoded(SizeInBytes : WORD) : ShortPointer;
PROCEDURE Execute(theProgram : AcerValuePointer);
PROCEDURE Decode(theCode : ShortPointer);
PROCEDURE ReclaimMemory;

IMPLEMENTATION ...
END.
```

Run defines the run-time representation of Acer programs and values; in fact, an Acer program is an Acer value so their is just one representation. Thus an Acer value is represented as an `AcerValuePointer`, which points at an `AcerValueType`, a record consisting of the following: a `Size` indicating the number of bytes allocated to the record; a `Next` `ShortPointer`, which points at another `AcerValueType` record (All values are linked as a chain for the purpose of garbage collection.); a `Flags` field, which is explained below; and a variant part that allows the rest of the record to be viewed as an array of bytes, words, or pointers.

The Flags field of an AcerValueType is used as follows: if the rest of the record is an array of AcerValuePointers, the Traced flag is set so that garbage collection will know that it should trace the rest of the record. The Traced flag is not set when the rest of the record encodes the bit representation of some value such as a string or a real. The Marked flag, as the name suggests, is used during garbage collection. Finally, the Coded flag is used to indicate that the rest of the record encodes machine instructions; this flag is necessary so that dynamic-values that involve code are properly handled. (A record containing code contains a reference to a node that in turn contains all the type-tags used by its dynamic-values.)

The AllocateX functions are used to allocate AcerValuePointers but because the record to which they point is segment aligned, only a ShortPointer is yielded.

Execute takes an AcerValuePointer, typically one that is yielded by the Loader function of the Link unit, and runs it to completion.

Decode prints a readable representation of an AcerValueType record that represents code. It is used for debugging.

ReclaimMemory reclaims all the memory used during the execution of an Acer program. (In this implementation, no garbage collection is performed while a program is actually executing. A practical implementation must include this.)

C.23 Link

```
UNIT Link;  INTERFACE {$F+,O+,L-}

USES FileServer, Manager, Index, Access, Makers, Grammar, AST, LineBuffer,
     Line, Unparser, Parser, MPS1, Run, Compile, MPS;

PROCEDURE StoreEncodedInstructions(theCluster, theInstructions : Node);
FUNCTION Loader(Name : STRING) : AcerValuePointer;
PROCEDURE WriteAcerValue(theValue : AcerValuePointer; theType : Node);
FUNCTION AcerValueToNode(theValue : AcerValuePointer;
                         theType : Node) : Node;

IMPLEMENTATION ...
END.
```

Link serves several roles (and is perhaps badly named).

StoreEncodedInstructions, given a cluster and a code-patch representing instructions, encodes the instructions in compact form and stores them in a '.o' file using the FileServer unit.

Loader, given a value-identifier spelling, finds its '.o' file and all the '.o' files it depends on. The information from these files are then loaded as an AcerValuePointer. The result can then be Executed.

WriteAcerValue, given an Acer value and its type, prints to standard output the Acer syntactic representation of that value. Similarly, AcerValueToNode, given an Acer value and its type, converts the value to its representation as a node. This is the routine PCAcer uses to produce the node that results from running a program.

C.24 ObjectView

```
UNIT ObjectView;  INTERFACE {$F+,O+,L-}

USES Line, MPS;

TYPE
  RectanglePointer = ^ Rectangle;
  Rectangle =
    OBJECT
      x1, y1, dx, dy : WORD;
      FUNCTION x2 : WORD;
      FUNCTION y2 : WORD;
      FUNCTION Contains(x, y : WORD) : BOOLEAN;
      PROCEDURE Intersect(VAR Operand2, Result : Rectangle);
      PROCEDURE Displace(DeltaX, DeltaY : INTEGER);
    END;
  T = ^ ObjectCell;
  Iterator =
    OBJECT
      FUNCTION Next(VAR theViewedObject : T) : BOOLEAN; VIRTUAL;
      PROCEDURE Reset; VIRTUAL;
    END;
  ObjectCell =
    OBJECT
      theOwner : Node;
      ViewBox : RectanglePointer;
      theBanner : STRING;
      theStatusLine : STRING;
      x1, y1, x2, y2 : WORD;
      CONSTRUCTOR Construct(theNode : Node; theViewBox : RectanglePointer);
      DESTRUCTOR Destruct; VIRTUAL;
      FUNCTION Copy(theViewBox : RectanglePointer) : T; VIRTUAL;
      FUNCTION dx : WORD; VIRTUAL;
      FUNCTION dy : WORD; VIRTUAL;
      FUNCTION TextAt(x, y : WORD) : WORD; VIRTUAL;
      FUNCTION CharacterAt(x, y : WORD) : CHAR;
      FUNCTION AttributeAt(x, y : WORD) : BYTE;
      FUNCTION Banner : StringPointer; VIRTUAL;
      FUNCTION Owner : Node;
      PROCEDURE CorrectView; VIRTUAL;
      PROCEDURE Select(x, y : WORD); VIRTUAL;
      PROCEDURE ReSelect(x, y : WORD); VIRTUAL;
      PROCEDURE Extend(x, y : WORD); VIRTUAL;
      PROCEDURE Delete(VAR AffectedViews : Iterator); VIRTUAL;
      PROCEDURE Destroy(VAR AffectedViews : Iterator); VIRTUAL;
      PROCEDURE Insert(VAR AffectedViews : Iterator); VIRTUAL;
      PROCEDURE Yank; VIRTUAL;
    END;

IMPLEMENTATION ...
END.
```

ObjectView provides RectanglePointer and Rectangle to manipulate rectangular objects. A rectangle object has an upper-left corner (x1, y1), a lower-right corner (x2, y2), a width dx, and a height dy. One can test if a rectangle Contains a coordinate (x, y). One can determine the Intersection of two rectangles, which is itself a rectangle. And one can Displace a rectangle.

The main purpose of ObjectView, however, is to represent the behavior that is common to both text windows and node windows in the PCAcer environment. Thus a window is represented as a ObjectCell, which has theOwner as its owner node; a ViewBox, which is the area bounded by the window's frame; theBanner and theStatusLine, which are represented as a String; an upper-left corner at (x1, y1); and a lower-right corner at (x2, y2). A window can be Constructed, Destructed, and Copied. The width and height of a window are given by dx and dy. CharacterAt, given a coordinate in the ViewBox, determines the character at the coordinate and similarly, AttributeAt determines the color attribute at a given coordinate; the two are simultaneously determined by TextAt. The variables theBanner and theOwner should be access using Banner and Owner.

A window can be asked to CorrectView its ViewBox so that its selection point is in view. A window can be Selected at a given coordinate and since selecting the selection of a text window alters the selection mode, a window can be Reselected as well—this works just like Select but resets the selection mode. The selection of a window can also be Extended.

A window must also support the editing commands Delete, Destroy, Insert, and Yank.

The first three take an Iterator argument that iterates through the sequence of other windows that are also effected by the edit. Iterator is defined as an object with a Next function for mapping through the sequence of windows and a Reset function for restarting the iteration.

ObjectView does not implement useful bodies for most ObjectCell routines; the units LineView and TokenView do this.

C.25 LineView

```
UNIT LineView;  INTERFACE {$F+,O+,L-}

USES Line, LineBuffer, MPS, ObjectView;

TYPE
  ModeType = (CharacterMode, TokenMode, LineMode);
  T = ^ ObjectCell;
  ObjectCell =
    OBJECT (ObjectView.ObjectCell)
      theLineBuffer : LineBufferType;
      Mode : ModeType;
      CONSTRUCTOR Construct(theNode : Node; theViewBox : RectanglePointer);
      CONSTRUCTOR ArbitraryConstruct
                   (theNode : Node;
                    InitialLineBuffer : LineBufferType;
                    theViewBox : RectanglePointer);
      DESTRUCTOR Destruct; VIRTUAL;
      FUNCTION Copy(theViewBox : RectanglePointer) : ObjectView.T; VIRTUAL;
      FUNCTION dx : WORD; VIRTUAL;
      FUNCTION dy : WORD; VIRTUAL;
      FUNCTION TextAt(x, y : WORD) : WORD; VIRTUAL;
      FUNCTION Banner : StringPointer; VIRTUAL;
      PROCEDURE CorrectView; VIRTUAL;
      PROCEDURE Select(x, y : WORD); VIRTUAL;
      PROCEDURE Extend(x, y : WORD); VIRTUAL;
      PROCEDURE Delete(VAR AffectedViews : Iterator); VIRTUAL;
      PROCEDURE Destroy(VAR AffectedViews : Iterator); VIRTUAL;
      PROCEDURE Insert(VAR AffectedViews : Iterator); VIRTUAL;
      PROCEDURE Yank; VIRTUAL;
      PROCEDURE EnterText(ch : CHAR; VAR AffectedViews : Iterator);
      PROCEDURE BuildLineBuffer;
    END;

VAR theScrapView : T;
```

```
FUNCTION Linearize(theNode : Node; Width : WORD) : LineBufferType;

IMPLEMENTATION ...
END.
```

LineView extends the behavior of a window as defined by ObjectView. It implements PCAcer text windows.

The type ModeType is defined to denote the three different selection modes that a text window supports. Like ObjectView, LineView defines T and ObjectCell, which inherit their implementation from ObjectView. LineView extends an ObjectCell as follows.

A text window contains theLineBuffer to hold the buffer of text that it views. Mode indicates the current selection mode. A text window can be constructed using Construct, which then uses BuildLineBuffer to make the appropriate line-buffer from the owner node. ArbitraryConstruct constructs a text window with a given line-buffer.

A text window defines the same operations as an ObjectView window. In addition, EnterText is provided for inserting a character at the selection point; it takes an iterator that sequences through other windows that are affected by this insert. BuildLineBuffer is provided to construct a line-buffer from the owner; it uses Linearize to format the owner node.

The variable theScrapView is provided as a text window from which insertions are taken and to which deletions go.

C.26 TokenLine

```
UNIT TokenLine;   INTERFACE {$F+,O+,L-}


USES Line, LexicalAnalyzer, MPS, Sequence;

TYPE
  TokenLineType = SequenceType;
  TokenLinePointer = ^ TokenLineType;
  TokenRecord =
    RECORD
      theToken : TokenType;
      theOwner, theEnclosingDenoter : Node;
      thehPosition : WORD;
      theString : StringPointer
    END;
  TokenRecordPointer = ^ TokenRecord;

CONST
  MaxTokenLineLength = MaxSequenceLength DIV SizeOf(TokenRecord);
  NullTokenLine : TokenLineType = NIL;

FUNCTION Construct(theTokenLineLength : WORD) : TokenLineType;
PROCEDURE Destruct(VAR theTokenLine : TokenLineType);
FUNCTION Length(theTokenLine : TokenLineType) : WORD;
FUNCTION NthElement(theTokenLine : TokenLineType; n : WORD) :
           TokenRecordPointer;
PROCEDURE AppendElement
            (NextToken : TokenType;
             NextOwner, NextEnclosingDenoter : Node;
             NexthPosition : WORD;
             NextString : StringPointer;
             VAR theTokenLine : TokenLineType);

IMPLEMENTATION ...
END.
```

Just as the unit Line provides for lines of characters, the unit TokenLine provides for lines of tokens, which are represented using the Sequence unit.

A token-line is created by Construct and freed by Destruct. Its length is given by Length and its elements are accessed by NthElement.

The only other operation provided is AppendElement, which appends a TokenRecord to a token-line. Such a TokenRecord consists of a Token, the token's owner node, the token's closest-containing denoter, the token's column position, and the token's spelling.

C.27 TokenBuffer

```
UNIT TokenBuffer;  INTERFACE {$F+,O+,L-}

USES Grammar, TokenLine, Sequence, MPS;

TYPE
  TokenBufferType = SequenceType;
  TokenBufferPointer = ^ TokenBufferType;

CONST
  MaxTokenBufferLength = MaxSequenceLength DIV 4;
  NullTokenBuffer : TokenBufferType = NIL;

FUNCTION Tokenize(theNode : Node; Width : WORD) : TokenBufferType;
FUNCTION Construct(theTokenBufferLength : WORD) : TokenBufferType;
PROCEDURE Destruct(VAR theTokenBuffer : TokenBufferType);
PROCEDURE AppendElement
            (NextTokenLine : TokenLineType;
             VAR theTokenBuffer : TokenBufferType);
FUNCTION Length(theTokenBuffer : TokenBufferType) : WORD;
FUNCTION NthElement(theTokenBuffer : TokenBufferType; n : WORD) :
           TokenLinePointer;
FUNCTION NearestTokenRecord(theTokenBuffer : TokenBufferType;
                              VAR x, y : WORD)
           : TokenRecordPointer;
PROCEDURE NearestTokenRecords
            (theTokenBuffer : TokenBufferType;
             x, y : WORD;
             VAR theLeftTokenRecord,
                 theRightTokenRecord : TokenRecordPointer);

IMPLEMENTATION ...
END.
```

Just as the unit LineBuffer provides a type for representing sequences of Lines, the unit TokenBuffer provides a type for representing sequences of TokenLines. It is implemented using the unit Sequence and provides the same kinds of operation.

In addition, it provides the function NearestTokenRecord, which takes a TokenBuffer and a coordinate and yields a TokenRecordPointer to the TokenRecord nearest to the coordinate. Similarly, the procedure NearsetTokenRecords takes a TokenBuffer and a coordinate and sets to theLeftTokenRecord and theRightTokenRecord pointers to the two TokenRecords on either side of the coordinate.

C.28 TokenView

```
UNIT TokenView;  INTERFACE {$F+,O+,L-}

USES Access, Line, TokenBuffer, TokenLine, MPS, ObjectView, AST;

TYPE
  T = ^ ObjectCell;
  ObjectCell =
    OBJECT (ObjectView.ObjectCell)
```

```
      theTokenBuffer : TokenBufferType;
      theLeftNode, theRightNode, theEnclosingDenoter : Node;
      theUnparseWidth : WORD;
      CONSTRUCTOR Construct(theNode : Node; theViewBox : RectanglePointer);
      DESTRUCTOR Destruct; VIRTUAL;
      FUNCTION Copy(theViewBox : RectanglePointer) : ObjectView.T; VIRTUAL;
      FUNCTION dx : WORD; VIRTUAL;
      FUNCTION dy : WORD; VIRTUAL;
      FUNCTION TextAt(x, y : WORD) : WORD; VIRTUAL;
      FUNCTION Banner : StringPointer; VIRTUAL;
      PROCEDURE CorrectView; VIRTUAL;
      PROCEDURE Select(x, y : WORD); VIRTUAL;
      PROCEDURE Extend(x, y : WORD); VIRTUAL;
      PROCEDURE Delete(VAR AffectedViews : Iterator); VIRTUAL;
      PROCEDURE Destroy(VAR AffectedViews : Iterator); VIRTUAL;
      PROCEDURE Insert(VAR AffectedViews : Iterator); VIRTUAL;
      PROCEDURE DefinitionCopyAt
                  (VAR AffectedViews : Iterator; theNode : Node); VIRTUAL;
      PROCEDURE Yank; VIRTUAL;
      PROCEDURE CreateElement(x, y : WORD; VAR AffectedViews : Iterator);
      VIRTUAL;
      PROCEDURE BuildTokenBuffer;
      PROCEDURE SetNodeSelection(Left, Right, EnclosingDenoter : Node);
    END;

PROCEDURE MonitoredReplace
            (theNode, theReplacementNode : Node;
             VAR AffectedViews : Iterator);
PROCEDURE MonitoredDelete
            (theLeftNode, theRightNode : Node;
             VAR AffectedViews : Iterator);

VAR theScrapView : T;

IMPLEMENTATION ...
END.
```

TokenView extends the behavior of a window as defined by ObjectView. It implements PCAcer node windows. Like ObjectView, TokenView defines T and ObjectCell, which inherit their implementation from ObjectView. TokenView extends an ObjectCell as follows.

A node window uses theTokenBuffer to hold a formatted version of its owner. For specifying its selection point it uses theLeftNode, theRightNode, and theEnclosingDenoter. (Because the definition of a denoter may print a node that is also printed elsewhere, to completely specify the selected nodes of a node window, the enclosing denoter must be specified as well.) The variable theUnparseWidth is set to indicate the width used to when formatting theTokenBuffer.

In addition to the regular operations provided by ObjectView windows, a node window provides the following. The procedure CreateElement inserts an element into a list at the given coordinate. The procedure BuildTokenBuffer makes a TokenBuffer by formatting the owner node to a TokenBuffer. And the procedure SetNodeSelection sets the selection of a node window.

MonitoredReplace and MonitoredDelete are used in place of MPS's Replace and Delete so that their affects on window owners and selections can be maintained. For example, an unattached node can be replaced using MonitoredReplace with the effect that if it is the owner of any window then the owner is replaced.

The variable theScrapView is provided as the node window from which insertions are taken and to which deletions go.

C.29 WindowStack

```
UNIT WindowStack;  INTERFACE {$F+,O+,L-}

USES MPS, ObjectView, Line;

CONST
  Rows = 50;
  Columns = 80;

CONST
  TopMenuLine : STRING[80] =
    ' New ' + ' Clear ' + ' Fetch ' + ' Restore ' +
    '                                               ' + ' Quit ';

TYPE
  Corner = (UpperLeft, UpperRight, LowerLeft, LowerRight);
  Direction = (Left, Right, Up, Down);
  Window = ^ WindowObject;
  T = ^ ObjectCell;
  WindowObject =
    OBJECT
      ParentStack : T;
      ScreenBox, ViewBox : Rectangle;
      ViewedObject : ObjectView.T;
      theTopLine, theBottomLine, theStatusLine : STRING[Columns - 2];
      theLeftLine, theRightLine : STRING[Rows - 1];
    CONSTRUCTOR Construct(theParentStack : T; x1, y1, dx, dy : BYTE);
    DESTRUCTOR Destruct; VIRTUAL;
    PROCEDURE Reset;
    FUNCTION TopLine : StringPointer;
    FUNCTION BottomLine : StringPointer;
    FUNCTION LeftLine : StringPointer;
    FUNCTION RightLine : StringPointer;
    FUNCTION TextAt(x, y : BYTE) : WORD;
    FUNCTION CharacterAt(x, y : BYTE) : CHAR;
    FUNCTION AttributeAt(x, y : BYTE) : BYTE;
    PROCEDURE MoveViewDownward(dy : BYTE);
    PROCEDURE MoveViewUpward(dy : BYTE);
    PROCEDURE MoveViewLeftward(dx : BYTE);
    PROCEDURE MoveViewRightward(dx : BYTE);
    PROCEDURE RepositionViewY(Index : BYTE);
    PROCEDURE RepositionViewX(Index : BYTE);
    PROCEDURE CorrectView;
    PROCEDURE MoveScreen(theCorner : Corner);
    PROCEDURE MoveCorner(theCorner : Corner);
    PROCEDURE Copy(theCorner : Corner);
    FUNCTION IOCheck : BOOLEAN;
  END;
ObjectCell =
  OBJECT
    ErrorMessage : STRING[80];
    theWindows : ARRAY [0..63] OF Window;
    theBottom : BYTE;
    SelectedWindow, ScrapTokenWindow, ScrapLineWindow : Window;
    CONSTRUCTOR Construct;
    DESTRUCTOR Destruct; VIRTUAL;
    FUNCTION TopWindow : Window;
    FUNCTION Length : BYTE;
    FUNCTION NthElement(n : BYTE) : Window;
    PROCEDURE CreateWindow(x1, y1, dx, dy : BYTE); VIRTUAL;
    PROCEDURE DestroyWindow(w : Window); VIRTUAL;
    PROCEDURE RedrawBox(VAR LastBox, CurrentBox : Rectangle);
    PROCEDURE UndrawBox(VAR theRectangle : Rectangle);
    PROCEDURE PutOnTopOrPutAtBottom(w : Window);
    PROCEDURE PutOnTopOrPutBelowTop(w : Window);
    PROCEDURE PutAbove(w1, w2 : Window);
    PROCEDURE PutBelow(w1, w2 : Window);
    PROCEDURE SetSelectedWindow(w : Window);
    FUNCTION TextAt(x, y : BYTE) : WORD;
    FUNCTION WindowAt(x, y : BYTE) : Window;
    PROCEDURE Refresh(w : Window);
    PROCEDURE Redraw(VAR theRectangle : Rectangle);
    PROCEDURE DoubleRedraw(VAR Rectangle1, Rectangle2 : Rectangle);
    END;
  StackIterator =
    OBJECT (Iterator)
      theWindow : Window;
      theOwner : Node;
      CurrentWindow : BYTE;
      CONSTRUCTOR Construct(w : Window);
      FUNCTION Next(VAR theViewedObject : ObjectView.T) : BOOLEAN; VIRTUAL;
      PROCEDURE Reset; VIRTUAL;
    END;

PROCEDURE DrawBox(VAR theRectangle : Rectangle; TextAttribute : BYTE);
PROCEDURE MoveCornerHelper(w : Window; theCorner : Corner);

CONST CrtBox : Rectangle = (x1 : 1; y1 : 1; dx : Columns; dy : Rows);

IMPLEMENTATION ...
END.
```

WindowStack is used to specify the behavior of windows and stacks of windows independent from the types of windows that may exist. It defines Rows and Columns as constants representing the size of the display screen. TopMenuLine contains the top line of text displayed in the PCAcer environment. Corner is defined to enumerate the types of corners and Direction is defined to enumerate the types of directions. Window is defined as a pointer to a WindowObject and T is defined as a pointer to an ObjectCell, which is used to represent a stack of windows.

A WindowObject is defined as follows. It contains a ParentStack, that is, the stack in which it occurs. It contains a ScreenBox, which is the rectangle containing the window on the screen. It contains a ViewBox, which is the rectangle specifying which part of the ViewedObject is displayed on the screen. And it contains theTopLine, theBottomLine, theStatusLine, theLeftLine, and theRightLine to specify the appearance of the window's frame.

A window is Constructed given theParentStack, the upper-left corner, and its size. It is destructed using Destruct. Reset is used to reset the appearance of the window's frame, in particular, to clear the status line. TopLine, BottomLine, LeftLine, and RightLine should be used to access the appearance of the window's frame. TextAt, CharacterAt, and AttributeAt determine the appearance of a window's body.

MoveViewDownward, MoveViewUpward, MoveViewLeftward, and MoveViewRightward, modify theViewBox. RepositionViewY (RepositionViewX) modifies the position of theViewBox relative to the amount of the frame above and below (to the left and to the right) of Index. CorrectView modifies theViewBox so that the window selection is in view. MoveScreen modifies theScreenBox so as to move the window on the screen. MoveCorner modifies the size of theViewBox and theScreenBox.

Finally, IOCheck determines if an IO error has occurred and displays an error message on the status line in that case.

WindowStack defines ObjectCell to represent a stack of windows. A window stack contains an ErrorMessage, which if it is not empty, is displayed in place of the TopMenuLine; theWindows, which is an array of 64 Window objects; theBottom, which is the index of the last array element to contain a valid window; and the SelectedWindow, the ScrapTokenWindow, and the ScrapLineWindow to indicate these special windows.

A window stack can be Constructed and Destructed. The variable theWindows is not usually accessed directly because the functions TopWindow, Length, and NthElement should be used instead. A new window is created on the top of the stack by CreateWindow and a window is removed from the stack by DestroyWindow.

A frame can be drawn to the screen using RedrawBox, which first removes the LastBox and then draws the CurrentBox. UndrawBox removes a frame drawn to the screen.

PutOnTopOrPutAtBottom, PutOnTopOrPutBelowTop, PutAbove, and PutBelow reorder the windows on the stack. SetSelectedWindow sets the selected window of the stack. TextAt yields the character and color attribute at a given coordinate of the window stack. WindowAt yields the top-most window to appear at the given coordinate.

Refresh redraws to the screen the contents of the specified window. Redraw redraws the specified portion of the window stack. Similarly, DoubleRedraw redraws the specified portions of the window stack with any overlap being drawn only once.

A StackIterator object iterates through the windows of a window stack that are affected by a modification to a node. A StackIterator is Constructed given a window, which it uses to set theWindow, and theOwner. Next sequences through those windows of the ParentStack of theWindow that would be affected by an edit operation to theOwner node. This is the Iterator that must passed to the various ObjectView edit operations.

The procedure DrawBox and MoveCornerHelper are auxiliary procedures that operate independently of window stacks. The constant CrtBox defines the Rectangle that contains the entire display screen.

C.30 NodeView

```
UNIT NodeView;  INTERFACE {$F+,O+,L-}


USES Line, LineBuffer, MPS, LineView, TokenView, ObjectView, WindowStack;

TYPE
  MenuTitles = ARRAY [0..Rows - 4] OF STRING[Columns - 4];
  MenuAction = PROCEDURE (n : BYTE);
  Window = ^ WindowObject;
  WindowObject =
    OBJECT (WindowStack.WindowObject)
      CONSTRUCTOR TokenConstruct
                    (theParentStack : WindowStack.T;
                     theRootNode : Node;
                     x1, y1, dx, dy : BYTE);
      CONSTRUCTOR LineConstruct
                    (theParentStack : WindowStack.T;
                     theRootNode : Node;
                     theLineBuffer : LineBufferType;
                     x1, y1, dx, dy : BYTE);
      CONSTRUCTOR FileConstruct
                    (theParentStack : WindowStack.T;
                     FileName : STRING;
                     x1, y1, dx, dy : BYTE);
      DESTRUCTOR Destruct; VIRTUAL;
      PROCEDURE Toggle;
      PROCEDURE CancelToggle;
      PROCEDURE Select(x, y : BYTE);
      PROCEDURE ZoomCopy;
      PROCEDURE ZoomIn;
      PROCEDURE ZoomParent;
      PROCEDURE ZoomRoot;
      PROCEDURE Extend;
      PROCEDURE Delete;
      PROCEDURE Destroy;
      PROCEDURE Poke(x, y : BYTE);
      PROCEDURE Insert;
      PROCEDURE Yank;
      PROCEDURE EnterText(ch : CHAR);
      PROCEDURE FindDefiningOccurrence;
      PROCEDURE FindDefinition;
      PROCEDURE FindType;
      PROCEDURE FindDenotation;
      PROCEDURE FindKind;
      PROCEDURE Validate;
      PROCEDURE DefinitionCopyAt;
      PROCEDURE Store;
      PROCEDURE Compile;
      PROCEDURE Link;
      PROCEDURE Run;
      FUNCTION TimeStampOf(UnitName : STRING) : LONGINT;
      PROCEDURE PickWindowSpecifiedMenu;
    END;
  NodeWindowStack = ^ WindowStackObject;
  WindowStackObject =
    OBJECT (WindowStack.ObjectCell)
      CONSTRUCTOR Construct;
      DESTRUCTOR Destruct; VIRTUAL;
      PROCEDURE ResetScrapWindows;
      PROCEDURE CreateTokenWindow(theRootNode : Node;
                                  x1, y1, dx, dy : BYTE);
      PROCEDURE CreateLineWindow
                  (theRootNode : Node;
                   theLineBuffer : LineBufferType;
                   x1, y1, dx, dy : BYTE);
      PROCEDURE CreateFileWindow(FileName : STRING; x1, y1, dx, dy : BYTE);
      PROCEDURE CreateWindow(x1, y1, dx, dy : BYTE); VIRTUAL;
      PROCEDURE DestroyWindow(w : WindowStack.Window); VIRTUAL;
      PROCEDURE FileInput;
      PROCEDURE Fetch(thePattern : STRING);
      PROCEDURE Menu(VAR Titles : MenuTitles;
                     NumberOfEntries : BYTE;
                     Action : MenuAction);
      PROCEDURE PickStackList;
      PROCEDURE StoreValues;
    END;

VAR theWindowStack : WindowStackObject;

IMPLEMENTATION ...
END.
```

NodeView specializes the operations of WindowStack for windows based on LineView and TokenView. The type WindowObject is specialized as follows.

Three constructor functions TokenConstruct, LineConstruct, and FileConstruct are provided to create, respectively, a node window given a node, a text window given a node and a line-buffer, and a text window given the name of a file. A window can be Destructed as usual.

Toggle can be applied to a window to convert it from a node window to a text window or vice versa. CancelToggle converts a text window back to a node window, discarding the text. Select sets the selection point.

ZoomCopy creates a new node window with the selection as its owner. ZoomIn sets the owner to the selection. ZoomParent sets the owner to the parent of the owner. And ZooomRoot sets the owner to the root of the owner.

Extend extends the selection by following the position of the mouse.

Delete invokes the window specific delete operation. Similarly, for Destroy, Insert, and Yank. Poke has an effect only on a node window for which it is used to insert a placeholder list element. EnterText inserts a character into a text window; when applied to a node window, it has the same effect as Toggle.

The FindX routines apply only for node windows for which they determine the specified semantic attribute of the selection point. A new window is created to hold the result. Validate also applies only for node windows—it checks the correctness of the owner node and an error message is displayed on the status line of the window in case of error.

DefinitionCopyAt can be applied to a node window and has the effect of using AST's DefinitionCopyAt to express the selected node of the selected window at the selection of the window to which it is applied; it replaces the selection with the result of the copy. Store, Compile, Link, and Run can be applied to node windows to produce the desired effected of storing, compiling, linking, or running the owner node. TimeStampOf, given a global value-identifier, determines the creation time associated with that identifier. It has the side-effect of recompiling anything that is required to compile the owner of the window to which it is applied.

Finally, PickWindowSpecifiedMenu invokes a popup menu. The menu contains different operations depending on whether the window is a node window or a text window.

NodeView specializes window stacks as follows. ResetScrapWindows empties the contents of both scrap windows. Four window-constructing procedures CreateTokenWindow, CreateLineWindow, CreateFileWindow, and CreateWindow are provided. (CreateWindow creates a window with an empty node as its owner.)

FileInput invokes the PCAcer menu for loading '.acc' files, i.e., text files containing Acer syntax. Fetch invokes the PCAcer menu for loading either '.tb,' '.vb,' '.vd,' '.tbl,' or '.vbl' files.

Menu displays a popup menu, given the titles, the number of titles, and a procedure that performs the operation associated with each title. PickStackList displays PCAcer's stack-list menu.

`StoreValues` stores every top-level fixed-value-binding and binding-list that is not up-to-date with respect the the version stored in the file system. This operation is performed by the PCAcer environment before any programs are compiled.

And finally, `theWindowStack` contains the one `WindowStackObject` used to represent PCAcer's environment.

The body of `NodeView` begins an input loop that continues until the quit command of the PCAcer environment is invoked. As such, `NodeView` really acts as a program but because it is defined as a unit, a program must be written that imports `NodeView`. This program can then specify things such as how to overlay the various units, how big the stack should be, and so on.

This finally completes the description of PCAcer's implementation and that of its metaprogramming system. Much detail has been left out, but perhaps not enough.

Appendix D

Quick reference

A table describing Acer's context-dependent manipulation primitives is given below. The metaprogramming system interface for these primitives is described in section 5.5, which describes the Acer host-language version, and in sections C.19 and C.20, which describe the Pascal host-language version. Tables summarizing Acer's type-system follow.

D.1 Context-dependent manipulation primitives

relation	given	yields	reference
defining-occurrence	*id*	The corresponding identifier node that defines the identifier node *id* for its particular scope.	3.5.1, A.3.1
type	*expr*	The type node that represents the type of the value or type expression *expr*.	3.5.2, A.3.3
kind	*expr*	The type node that represents the kind (the type of the type) of the value or type expression *expr*.	4.7.1, A.3.3
definition	*expr*	The expression node that represents the meaning of the value or type expression *expr* as specified by Acer's rewrite rules.	3.5.3, A.3.2
denotation	*expr*	The expression node that results from rewriting the value or type expression *expr*, rewriting the rewritten expression, and so on, until the resulting expression cannot be rewritten.	4.7.1, A.3.2
definition-copy-at	*expr1* *expr2*	The expression that results when the vaulue or type expression *expr1* is re-expressed in terms of the scope at *expr2*.	A.5.2.2, A.5.2.1, 4.10, 4.11, 4.13

D.2 Identifier

type-identifier	*Type*	X	_	_*type*
value-identifier	*value*	x	?	!#$&*+-/<=>\^|~

D.3 Deriving declarations for named arguments

	type	fixed-value	variable-value
binding	**let** $T :: U$ **be** V	**let** $x : T$ **be** y	**let var** $x : T$ **be** y
declaration	$T :: U$	$x : T$	**var** $x : T$

D.4 Deriving declarations for anonymous arguments

	type	fixed-value	variable-value
argument	T	x	**let var be** x
declaration	$:: U$	$: T$	**var** $: T$

D.5 Component type equivalence

=	
$T :: U$	$:: U$
$x : T$	$: T$
var $x : T$	**var** $: T$

D.6 Component subtype

$\sqsubseteq$		condition
$T :: U1$	$T :: U2$	$U1 \sqsubseteq U2$
$x : T1$	$x : T2$	$T1 \sqsubseteq T2$
var $x : T1$	**var** $x : T2$	$T1 = T2$

D.7 Special nodes

empty	**nothing**
arbitrary-list	**arbitrary** [x] [T] [x **then** y] [**nothing**] **end**

D.8 Concrete-type

	literal	type
tuple	**tuple** **let** $T :: U$ **be** V, **let** $x : T$ **be** z, **let var** $y : T$ **be** z, T, x, **let var be** x **end**	**Tuple** $T :: U$; $x : T$; **var** $y : T$; $:: U$; $: T$; **var** $: T$ **end**
dynamic	**dynamic** **let** $T :: U$ **be** V, **let** $x : T$ **be** z, **let var** $y : T$ **be** z, T, x, **let var be** x **end**	**Dynamic** $T :: U$; $x : T$; **var** $y : T$; $:: U$; $: T$; **var** $: T$ **end**
record	**record** **let** $T :: U$ **be** V, **let** $x : T$ **be** z, **let var** $y : T$ **be** z **end**	**Record** $T :: U$; $x : T$; **var** $y : T$ **end**
variant	**variant** x **of** T **with** (**let** $T :: U$ **be** V, **let** $x : T$ **be** z, **let var** $y : T$ **be** z, T, x, **let var be** x) **end**	**Variant** T **of when** x **then** ($T :: U$; $x : T$; **var** $y : T$; $:: U$; $: T$; **var** $: T$) **end**
function	**function** ($T :: U$; $x : T$; $:: U$; $: T$) x **end**	**Function** ($T :: U$; $x : T$; $:: U$; $: T$) T **end**
enumeration	**Enumeration** x, y **end**.x	**Enumeration** x, y **end**
option	**Option** x, y **end**.x	**Option** x, y **end**
any		**Any**

D.9 Type-operator

abstract	concrete
Operator $(T :: U)$ T **end**	**Operator** $(T :: U)$ U **end**

D.10 Abstract-type

	literal	condition	visible supertype
type-identifier	T	$T :: U$	U
type-selection	$x.T$	x : **Tuple** $T :: U$ **end**	U
operator-call	$O(T)$	O :: **Operator** (:: U) V **end**	V

D.11 Standard types

literal	type	implemented by
array x, y,z **of** T **end**	*Array* (T)	*arrays*
'x'	*Character*	*character*
error	*Error*	*exceptions*
exception (T)	*Exception* (T)	*exceptions*
1	*Integer*	*integer*
pointer (x)	*Pointer* (T)	*pointers*
raise e **with** x **end**	*Raise*	*exceptions*
1.0	*Real*	*real*
reference (x)	*Reference* (T)	*references*
"xyz"	*String*	*string*
{}	*Void*	*exceptions*

D.12 Notation

	value	type
reused-identifier	x'[1]	T'[1]
block	{**let** x **be** y; $f(x)$}	{**let** T **be** U; $O(T)$}
denoter	{(x)}	{[T]}
type-designation	**TYPE** (x)	**TYPE** (T)
selection	x.x	$x.T$

D.13 Iterator and accumulator

```
let Iterator be
  Operator (BaseType :: Any)
    Tuple
      done : Exception ( Void)
      produce :
        Function () BaseType end
      terminate :
        Function () Void end
    end
  end
```

```
let Accumulator be
  Operator (BaseType :: Any; ResultType :: Any)
    Tuple
      done : Exception ( Void)
      consume :
        Function (: BaseType) Void end
      terminate :
        Function () ResultType end
    end
  end
```

D.14 Subtype

subtype $\sqsubseteq$ supertype		condition
Tuple *T* :: *U1*; : *T* **end**	**Tuple** *T* :: *U2* **end**	*U1* $\sqsubseteq$ *U2*
Dynamic *T* :: *U1*; : *T* **end**	**Dynamic** *T* :: *U2* **end**	*U1* $\sqsubseteq$ *U2*
Record *x* : *T*; *T* :: *U1* **end**	**Record** *T* :: *U2* **end**	*U1* $\sqsubseteq$ *U2*

Variant Enumeration *e1* **end of**
when *e1* **then** (*T* :: *U1*; : *T*)
end
	Variant Enumeration *e1*, *e2* **end of**
	when *e1* **then** (*T* :: *U2*);
	when *e2* **then** (: *V*)
	end
		U1 $\sqsubseteq$ *U2*

Function (:: *T1*) *U1* **end**	**Function** (:: *T2*) *U2* **end**	*U1* $\sqsubseteq$ *U2* *T2* $\sqsubseteq$ *T1*
Enumeration *e1* **end**	**Enumeration** *e1*, *e2* **end**	
Option *e1* **end**	**Option** *e2*, *e1* **end**	
T	*T*	*T* :: *U*
T1	*T2*	*T1* :: *U* *U* $\sqsubseteq$ *T2*
x1.T	*x2.T*	*x1* = *x2*
O1 (*T1*)	*O2* (*T2*)	*O1* = *O2* *T1* = *T2*
T	**Any**	
Error	*T*	
Raise	*T*	
T	*Error*	
T	*Raise*	

D.15 Computation

function-call	*f* (*T*, *x*)
unary-method-call	{*op x*}
dyadic-method-call	{*x op y*}
assignment	{*x* **becomes** *y*}
is-test	{*x* **is** *y*}
is-not-test	{*x* **isnot** *y*}
and-if-test	{*x* **andif** *y*}
or-if-test	{*x* **orif** *y*}
index	*x*[*y*]
dereference	*x*@
ord-call	**ord** (*x*)
val-call	**val** (*T*, *x*)
compound-value	**begin** *x*; *y*; *z* **end**
code-patch	**code** *T*; *x*; *y*; *z* **end**
accumulation	*accumulator* ([*x*, *y*, *z*])
iteration	**for** *element* **in** *iterator* **andif** *filter* (*element*) **do** *accumulator body* (*element*) **end**

conditional	**if** *condition1* **then** *consequent1*
	elsif *condition2* **then** *consequent2*
	else *default* **end**

dynamic-inspection	**inspect** *dynamicValue* **Then** **when** *T* **with** *definedIdentifier* **then** *consequent1*; **when** *U* **then** *consequent2* **else** *default* **end**
variant-inspection	**inspect** *variantValue* **then** **when** *x* **with** *definedIdentifier* **then** *consequent1*; **when** *y*, *z* **then** *consequent2* **else** *default* **end**
try	**try** *body* **then** **when** *exception1* **with** *definedIdentifier1* **do** *consequent1*; **when** *exception2*, *exception3* **do** *consequent2* **else** *default* **end**
keep-trying	**keep trying** *loopBody* **then** **when** *exception* **with** *definedIdentifier* **do** *consequent* **else** *default* **end**
try-finally	**try** *body* **finally** *finalAction* **end**

Bibliography

[ACPP91] Martin Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[Ada83] United States Department of Defense. *Reference Manual for the Ada Programming Language*, February 1983. ANSI/MIL-STD-1815A-1983.

[AU72] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation and Compiling*, volume 1: Parsing of *Series in Automatic Computation*. Prentice-Hall, Englewood Cliffs, New Jersey, 1972.

[AU73] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation and Compiling*, volume 2: Compiling of *Series in Automatic Computation*. Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

[AU77] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design*. Series in Computer Science and Information Processing. Addison-Wesley, Reading, Massachusetts, 1977.

[Ben87] Jon Bentley. Programming pearls—profilers. *Communications of the ACM*, 30(7):587–592, July 1987.

[BS86] Rolf Bahlke and Gregor Snelting. The PSG system: From formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547–576, October 1986.

[Cam88] R. D. Cameron. An abstract prettyprinter. *IEEE Software*, 5(6):61–67, November 1988.

[Cam89] R. D. Cameron. Efficient high-level iteration with accumulators. *ACM Transactions on Programming Languages and Systems*, 11(2):194–211, April 1989.

[Car89] Luca Cardelli. Typeful programming. Technical Report 45, Digital Systems Research Center, Palo Alto, California, May 1989.

[Cas84] Barbara A. Cassel, editor. *MC68020 32-Bit Microprocessor User's Manual.* Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[CDG+88] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. *Modula-3 Report.* Digital Systems Research Center, Palo Alto, California, August 1988.

[CF91] Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN 91 Conference on Programming Language Design and Implementation*, pages 278–292. SIGPLAN, June 1991.

[CI84] R. D. Cameron and M. R. Ito. Grammar-based definition of metaprogramming systems. *ACM Transactions on Programming Languages and Systems*, 6(1):20–54, January 1984.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):472–522, December 1985.

[CW90] G.V. Cormack and A.K. Wright. Type-dependent parameter inference. In *Proceedings of the ACM SIGPLAN 90 Conference on Programming Language Design and Implementation*, pages 127–136. SIGPLAN, June 1990.

[DEFH87] S.A. Dart, R.J. Ellison, P.H. Feiler, and A.N. Habermann. Software development environments. *IEEE Computer*, 20(11):18–28, November 1987.

[DGKLM84] V. Donzeau-Gouge, G. Kahn, B. Lang, and B. Mélèse. Document structure and modularity in Mentor. *SIGPLAN Notices*, 19(5):141–148, May 1984.

[DMS84] N. M. Delisle, D. E. Menicosy, and M. D. Schwartz. Viewing a programming environment as a single tool. *SIGPLAN Notices*, 19(5):49–56, May 1984.

[Dyc90] J. Michael Dyck. Syntactic manipulation systems for context-dependent languages. Master's thesis, School of Computing Science, Simon Fraser University, August 1990.

[FM91] Pascal Fradet and Daniel Le Métayer. Compilation of a functional language by program transformation. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[Gro89] Peter Grogono. Comments, assertions, and pragmas. *SIGPLAN Notices*, 24(3):79–84, March 1989.

[GWEB83] Gerhard Goos, William A. Wulf, Arthur Evans, Jr., and Kenneth J. Butler. *DIANA: An Intermediate Language for Ada*, volume 161 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1983.

[Har84] David M. Harland. *Polymorphic Programming Languages: Design and Implementation.* Series in Computers and Their Applications. Ellis Horwood, Chichester, 1984.

[Hil83] Paul N. Hilfinger. *Abstraction Mechanisms and Language Design.* ACM Distinguished Dissertations. The MIT Press, Cambridge, Massachusetts, 1983.

[Hoa87] C. A. R. Hoare. Hints on programming language design. In Ellis Horowitz, editor, *Programming Languages: A Grand Tour*, Computer Software Engineering Series, pages 31–40. Computer Science Press, Rockville, Maryland, third edition, 1987.

[HT85] Susan Horwitz and Tim Tietelbaum. Relations and attributes: A symbiotic basis for editing. In *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, pages 93–106. SIGPLAN, July 1985.

[HT86] S. Horwitz and T. Teitelbaum. Generating editing environments based on relations and attributes. *ACM Transactions on Programming Languages and Systems*, 8(4):577–608, October 1986.

[ISO83] Specification for computer programming language Pascal. ISO 7185, 1983.

[Kae88] Michael J. Kaelbling. Programming languages should not have comment statements. *SIGPLAN Notices*, 23(10):59–60, October 1988.

[KH89] Richard Kelsey and Paul Hudak. Realistic compilation by program transformation. *Proceedings of the Sixteenth Annual ACM Symposium on Princples of Programming Languages*, pages 281–292, January 1989.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[Knu84] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[Kur91] Andrew Kurn. *The G Programming Language.* PhD thesis, Simon Fraser University, December 1991.

[LAB+81] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual*, volume 114 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1981.

[LC87] Xing Liu and Patrick Conley. Program translation by manipulating abstract syntax trees. In *C++ Workshop*, pages 345–360. USENIX Association, November 1987.

[Mac87] Bruce J. MacLennan. *Principles of Programming Languages: Design, Evaluation, and Implementation.* Holt, Rinehart and Winston, New York, second edition, 1987.

[MDC92] E. A. T. Merks, J. M. Dyck, and R. D. Cameron. Language design for program manipulation. *IEEE Transactions on Software Engineering*, 18(1):19–33, January 1992.

[Mer87] Eduardus A.T. Merks. Compilation using multiple source-to-source stages. Master's thesis, School of Computing Science, Simon Fraser University, April 1987.

[Mey88] Bertand Meyer. *Object-oriented Software Construction.* Series in Computer Science. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[Mil78] Robin Milner. The standard ML core language. Technical Report Internal Report CSR-168-84, Edinburgh University, 1978.

[Min90] Sten Minör. *On Structure-Oriented Editing.* Doctoral dissertation, Department of Computer Science, Lund University, 1990.

[MN88] Ole Lehrmann Madsen and Claus Norgaard. An object-oriented metaprogramming system. In B. Shriver, editor, *Proceedings of the 21st Annual Hawaii International Conference: Software Track*, pages 406–415. IEEE Computer Society, January 1988.

[Pag81] F. G. Pagan. *Formal Specification of Programming Language: A Panoramic Primer.* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[Par90] Helmut A. Partsch, editor. *Specification and Transformation of Programs: A Formal Approach to Software Development.* Texts and Monographs in Computer Science. Springer-Verlag, Berlin, 1990.

[Pep84] Peter Pepper, editor. *Program Transformation and Programming Environments*, volume 8 of *NATO ASI Series F: Computer and Systems Sciences.* Springer-Verlag, Berlin, 1984.

[PK91] Dewayne E. Perry and Gail E. Kaiser. Models of software development environments. *IEEE Transactions on Software Engineering*, SE-17(3):283–295, March 1991.

[PS83] H. Partsch and R. Steinbruggen. Program transformation systems. *ACM Computing Surveys*, 15(3):199–236, September 1983.

[Rep84] Thomas W. Reps. *Generating Language-Based Environments.* ACM Distinguished Dissertations. The MIT Press, Cambridge, Massachusetts, 1984.

[Ros85] D. S. Rosenblum. A methodology for the design of Ada transformation tools in a DIANA environment. *IEEE Software*, 2(2):24–33, March 1985.

[RT84] T. Reps and T. Teitelbaum. The Synthesizer Generator. *SIGPLAN Notices*, 19(5):42–48, May 1984.

[RT89] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors.* Springer-Verlag, New York, 1989.

[Str86] Bjarne Stroustrup. *The C++ Programming Language.* Series in Computer Science. Addison-Wesley, Reading, Massachusetts, 1986.

[Ten81] R. D. Tennent. *Princples Of Programming Languages.* International Series in Computer Science. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[TR81] Tim Teitelbaum and Thomas W. Reps. The Cornell Program Synthesizer: A syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, September 1981.

[TRH81] Tim Teitelbaum, Thomas Reps, and Susan Horwitz. The why and wherefore of the Cornell Program Synthesizer. *SIGPLAN Notices*, 16(6):8–16, June 1981.

[Tur82] T. N. Turba. A facility for the downward extension of a high-level language. *SIGPLAN Notices*, 17(6):127–133, September 1982.

[Tur86] D. A. Turner. An overview of Miranda. *SIGPLAN Notices*, 21(12):158–166, December 1986.

[vWMP+76] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker, editors. *Revised Report on the Algorithmic Language Algol 68.* Springer-Verlag, Berlin, 1976.

[Wik87] Åke Wikström. *Functional Programming Using Standard ML.* Series in Computer Science. Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

[Wir85] Niklaus Wirth. *Programming in Modula-2.* Texts and Monographs in Computer Science. Springer-Verlag, Berlin, third corrected edition, 1985.

[Wir87] Niklaus Wirth. On the design of programming languages. In Ellis Horowitz, editor, *Programming Languages: A Grand Tour*, Computer Software Engineering Series, pages 23–30. Computer Science Press, Rockville, Maryland, third edition, 1987.