SYNTACTIC MANIPULATION SYSTEMS FOR CONTEXT-DEPENDENT LANGUAGES

by

Michael Dyck

B.Sc., University of Winnipeg, 1984

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the School

of

Computing Science

SIMON FRASER UNIVERSITY

August 1990

APPROVAL

Name: Michael Dyck

Degree: Master of Science

Title of thesis: Syntactic Manipulation Systems for Context-Dependent Languages

Examining Committee:

Chair: Dr. Binay Bhattacharya

Dr. Rob Cameron
Senior Supervisor

Dr. Lou Hafer
Committee Member

Dr. Nick Cercone
Committee Member

Dr. Warren Burton
Examiner

Date Approved: 1990 August 10

Title of Thesis/Project/Extended Essay

Syntactic Manipulation Systems for Context-Dependent Languages.

Author: [illegible]

(signature)

John Michael Dyck

(name)

1990 August 16

(date)

ABSTRACT

Software developers use many tools to increase their efficiency and improve their software. A *syntactic manipulation system* (SMS) is one such tool. An SMS is a set of programming-language routines that can be used to create, manipulate, and modify the syntactic structures defined by a grammar. An SMS is useful whenever a program manipulates data objects that are described well by a grammar; programs, structured documents, musical scores, software specifications, mathematical formulae, and recursive data structures are all objects that could be manipulated with an SMS.

Previous research has established a method for deriving an SMS from a context-free grammar. Thus, this method produces SMSs that deal abstractly only with those aspects of languages that are captured by context-free grammars. However, experience shows that programs using such SMSs routinely require more complex, *context-dependent* information. For example, if a program were using an SMS to manipulate Pascal code, it might need answers to the following questions:

- "What is the type of this variable?"
- "Has NEW been redefined for this scope?"
- "Will this identifier conflict with any existing identifiers?"
- "Where is the resolution for this forward declaration?"
- "Is this argument compatible with that parameter?"
- "What is the value of this constant-expression?"
- "Where are all the calls to this procedure?"

In augmenting context-free SMSs with routines to answer such questions, previous approaches have been ad hoc, with neither a formal strategy to determine what routines should be added, nor a method for deriving them from a description of the target-language. This thesis provides a systematic approach to replace the previous ad hoc approaches. Specifically, I have devised NURN, a notation for specifying context-dependent languages in terms of relations between the nodes of context-free abstract syntax-trees. Moreover, I have implemented Ginger, a system that takes a NURN grammar and generates the corresponding context-dependent SMS. To demonstrate the usefulness of NURN and Ginger, I have used NURN to fully define the syntax of Standard Pascal, and written a Pascal syntax-checker using the resulting SMS.

ACKNOWLEDGEMENTS

Thanks:

- To my Senior Supervisor Rob Cameron, for patience, availability, and insight, for dealing with the red tape, and for inventing GRAMPS, without which I might never have developed NURN.
- To the other members of my Examining Committee (Lou Hafer, Nick Cercone and Warren Burton), for their time and effort in reading my thesis.
- To Howard Hamilton in large quantities, the best editor a grad student could want.
- To Ed Merks for technical discussions, to Cathy Levinson for cheerful legwork, and to both for their comments on the thesis, but especially for their help in preparing my seminar.
- To Brian Terry, for developing GRAFS and entrusting it to my not-so-tender care.
- To Yvonne Coady, for relentless encouragement and the occasional free lunch.
- To Ranabir Gupta, Pat Pattabhiraman, and Fred Popowich, for various consultations.
- To the Natural Sciences and Engineering Research Council of Canada, the Province of British Columbia, Simon Fraser University, Rob Cameron (via the SFU Centre for Systems Science), and my dad Dave Dyck, for financial support during my career as a Masters student.

Thanks again everyone!

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

1.1 Problem Statement

Software developers use many tools to increase their efficiency and improve their software. A *syntactic manipulation system* (SMS) is one such tool. An SMS is a set of programming-language routines that can be used to create, manipulate, and modify the syntactic objects defined by a grammar. An SMS is useful whenever a program manipulates data objects that are described well by a grammar.

When referring to an SMS, it is important to distinguish two languages. The *host-language* of an SMS is a programming language; the routines of the SMS and the programs that use these routines are written in the host-language. On the other hand, the *target-language* of an SMS is the language defined by the grammar on which the SMS is based; the syntactic objects that the SMS manipulates represent phrases in its target-language. In most previous work with SMSs, the target-language has been a programming language[1], and may even be the SMS's host-language, but it might be a language of structured documents, musical scores, software specifications, data descriptions, mathematical formulae, or recursive data structures.

An SMS is particularly useful when its target-language is both human- and machine-readable. Typically, the SMS then provides *parsers* and *printers*, routines to convert between the textual and structured representations of target-language phrases. These routines allow the software developer to read and write target-language phrases in the familiar textual form and easily convert them into the structured form that the SMS manipulates.

Cameron has established GRAMPS[2] [CamIto84], a method for deriving an SMS from a context-free grammar. Consequently, this method produces SMSs that deal abstractly only with those aspects of languages that are captured by context-free grammars. However, experience shows that programs using such SMSs routinely require more complex information. Roughly speaking, this information involves concepts that arise when describing a target-language's context-dependent syntax. (This idea is clarified in Chapter 3.) The following questions illustrate the nature of this context-dependent information when the target-language is a procedural programming language such as Pascal.

---

[1] In [CamIto84], Cameron and Ito refer to such an SMS as a *metaprogramming system*, since the software developer uses it to write "metaprograms", i.e., programs written in the host-language about programs written in the target-language.

[2] a "GRAmmar-based MetaProgramming Scheme"

- "What is the type of this variable?"
- "Has NEW been redefined for this scope?"
- "Will this identifier conflict with any existing identifiers?"
- "Where is the resolution for this forward declaration?"
- "Is this argument compatible with that parameter?"
- "What is the value of this constant-expression?"
- "Where are all the calls to this procedure?"

For example, the first question involves the concept of *type*, which is common in describing the syntax of context-dependent (programming) languages, but unnecessary in defining context-free languages.

To answer such questions, the software developer needs an integrated set of context-dependent routines, specifically constructed for the target-language. Previous work [Cam87, Mer87, Mer88] indicates the usefulness of adding such routines to a context-free SMS. However, the approaches have been ad hoc, with neither a formal strategy to determine what routines should be added, nor a method for deriving them from a description of the target-language. This thesis provides a systematic approach to replace the previous ad hoc approaches.

1.2 Overview of this Thesis

In Chapter 2, I present some necessary background on context-free SMSs, reviewing GRAMPS, a method for deriving a context-free SMS from a context-free grammar. In Chapter 3, I introduce an extension of GRAMPS called NURN (a Notation Using Relations on Nodes), which defines context-dependent languages in terms of relations between the nodes of context-free syntax-trees. The use of NURN is illustrated with numerous examples drawn from a complete NURN specification for Standard Pascal. From a NURN grammar for a target-language, one can derive a context-dependent NURN SMS. In Chapter 4, I discuss the operation of NURN SMSs, and present Ginger, a system for automatically generating them from NURN grammars. Finally, in Chapter 5, I summarize the contributions of this thesis and consider directions for further research.

Appendix 1 gives a context-free syntax for NURN grammars. Appendix 2 presents a complete NURN grammar for the programming language Standard Pascal, as defined in [ANSI83]. Appendix 3 discusses the validation of the NURN SMS that Ginger generates from this grammar.

CHAPTER 2
CONTEXT-FREE SYNTACTIC MANIPULATION SYSTEMS

A *context-free SMS* is a system for manipulating syntactic objects defined by a context-free grammar. In the GRAMPS Report [CamIto84], Cameron and Ito introduced *GRAMPS*, a scheme for deriving a context-free SMS from a grammar for the target-language. The particular form of the grammar used and the SMS constructed has varied between implementations [CamIto84, Ter87, Mer90, MadNor88], but each conforms to the spirit of the GRAMPS Report, and we include them all (grammars and SMSs) under the designation "GRAMPS-style". A GRAMPS-style grammar has three important characteristics:

- In formal language terms, it is equivalent to a context-free grammar.
- It determines both the concrete and abstract syntax of the target-language.
- It provides terminology for referring to syntactic objects and their (context-free) relationships.

GRAMPS is an important prerequisite to NURN, the notation for defining context-dependent target-languages that will be introduced in Chapter 3: every NURN grammar includes a GRAMPS-style grammar, and every NURN SMS includes a GRAMPS-style SMS. This chapter will acquaint the reader with GRAMPS-style grammars in Section 2.1, and with GRAMPS-style SMSs in Section 2.2.

2.1 GRAMPS-Style Grammars

One of the major ideas of GRAMPS is that a single grammar should determine both the concrete and abstract syntax of the target-language. The concrete syntax defines a textual language, while the abstract syntax defines a language of syntactic structures or *syntagms*. We will see in Section 2.2 that an SMS based on such a grammar provides manipulation routines to construct, examine, and edit instances of these syntagms (*syntactic objects*), with parsing and printing routines to convert between syntactic objects and textual phrases of the target-language.

A GRAMPS-style grammar for a target-language comprises a set of *rules*. Concretely, each rule defines a symbol and its production in a *regular-right-part grammar* for the target-language. Abstractly, each rule defines a *syntactic domain*, i.e., a set of syntagms. The concrete symbol and the abstract syntactic domain defined by a rule are related: each phrase in the sublanguage derivable from the symbol corresponds to a single syntagm in the syntactic domain. Subsections 2.1.1 through 2.1.4 will examine GRAMPS-style rules in detail, but first, GRAMPS-style grammars will be introduced by means of a comparison with context-free grammars and regular-right-part grammars.

Regular-right-part grammars (RRPGs) [Lal77] are equivalent in power[1] to context-free grammars (CFGs), but allow the right part of a production to be a regular expression of symbols, including union and closure constructs as well as the simple concatenation allowed in CFGs. Using these additional constructs often results in an RRPG that is significantly more concise than the equivalent CFG. GRAMPS-style grammars have some of the conciseness of RRPGs, but curb their unrestricted complexity to produce a more disciplined treatment of syntactic domains.

In an RRPG, a production can use concatenation, union, and closure in arbitrarily complex combinations. In a GRAMPS-style grammar, these three constructs can all be used, but any particular rule can use only one of them. The resulting three kinds of rules are called *construction rules*, *alternation rules*, and *repetition rules*, respectively. Concretely, each of these rules defines one non-terminal symbol of an RRPG for the target-language.

Typically, the target-language also has lexical symbols, or *lexemes*. For example, a programming language has identifiers and numeric literals. To define these symbols, a GRAMPS-style grammar also needs *lexical rules*, but GRAMPS does not constrain the form of lexical rules. Since a GRAMPS-style SMS treats the content of a lexeme as just a sequence of characters, any notation for defining character-level syntax is fine, as long as the parsing routines can handle it.

Formally, GRAMPS-style grammars are equivalent in power to RRPGs and CFGs: there are straightforward conversions between the three. Therefore, outside of this section, the term "context-free" is used with reference to any of these formalisms.

To illustrate these kinds of grammars, the following example uses a CFG, an RRPG, and a GRAMPS-style grammar to define the syntax of a tiny language of statements, such as might appear in a programming language. For the sake of brevity, the language has only two kinds of statements (procedure-calls and for-loops), and only two kinds of expressions (identifiers and numerals). The productions for identifiers and numerals are omitted, since they are lexical symbols; how they are defined is not important to this discussion.

First, in a CFG, the syntax could be written:

```
Statement => ProcedureCall
Statement => ForLoop

ProcedureCall => Identifier "(" ExpressionList ")"
```

[1] By "equivalent in power", I mean that the two classes of grammars derive the same class of languages, namely the context-free languages [Lal77].

```
ExpressionList => Expression
ExpressionList => Expression "," ExpressionList

ForLoop =>
   "FOR" Identifier ":=" Expression "TO" Expression "DO"
      StatementList
   "END"

StatementList => Statement
StatementList => Statement ";" StatementList

Expression => Identifier
Expression => Numeral
```

Note how expression-lists and statement-lists, which could sensibly be considered repetitive constructs, must nevertheless be defined recursively.

In an RRPG, the same information could be compressed into two productions:

```
Statement => Identifier "(" Expression ("," Expression)* ")"

          |  "FOR" Identifier ":=" Expression "TO" Expression "DO"
                Statement (";" Statement)*
             "END"

Expression => Identifier | Numeral
```

Here, the recursive definitions of `ExpressionList` and `StatementList` are replaced by explicitly repetitive constructs using the closure symbol ('*'). However, performing this replacement also eliminates the symbols `ExpressionList` and `StatementList`. Similarly, `ProcedureCall` and `ForLoop` have disappeared. This elision of symbols is fine if you are only intent on defining a textual language, but if you want to manipulate syntactic objects, or even just talk about them, it is important that syntactic domains have names. Thus, while RRPGs can be concise, unrestricted conciseness can lead to difficulties.

Finally, in a GRAMPS-style grammar, this syntax could appear as:

```
ALTERNATE CLOSED Statement IS ProcedureCall | ForLoop

CONSTRUCT ProcedureCall IS
   <Callee:Identifier> "(" <Arguments:ExpressionList> ")"

LIST ExpressionList OF Expression SEPARATED_BY _ ","

CONSTRUCT ForLoop IS
   "FOR" <LoopVariable:Identifier> ":="
                <Initial:Expression> "TO" <Final:Expression> "DO"
      <Body:StatementList>
   "END"

LIST StatementList OF Statement SEPARATED_BY ";"
```

```
        ALTERNATE OPEN Expression IS Identifier | Numeral
```

The particular notation used here is that accepted by my implementation of GRAFS; it is very close to the syntax accepted by the original implementation of GRAFS [Ter87]. Each rule begins with a keyword indicating its kind: `CONSTRUCT` for construction rules, `LIST` for repetition rules, and `ALTERNATE` for alternation rules. Obviously, this grammar is not as concise as the equivalent RRPG, but it allows us to refer to `ProcedureCalls` and `ForLoops`, and also to the `Arguments` of a `ProcedureCall` or the `Body` of a `ForLoop`. This ability to explicitly name syntactic domains and the parts of syntactic structures is one of the major advantages of using a GRAMPS-style grammar, and is used extensively by NURN grammars, as we will see in Chapter 3.

In the next three subsections, we will examine construction rules, repetition rules, and alternation rules, showing how each contributes to the concrete and abstract syntax of the target-language.

2.1.1 Construction rules

Recall that in a GRAMPS-style grammar, a construction rule is a rule using concatenation. Abstractly, a construction rule identifies a fixed sequence of syntactic domains, called its *component* domains, and defines a syntactic domain that is the cross product of these component domains. That is, for a construction rule with *n* component domains, a member of the rule's domain (a *construction* syntagm) consists of an ordered *n*-tuple of syntagms: one syntagm from each component domain. Each of these syntagms is a *component* of the construction syntagm. For example, the construction rule for `ProcedureCall`:

```
   CONSTRUCT ProcedureCall IS
      <Callee:Identifier> "(" <Arguments:ExpressionList> ")"
```

has two component domains: the first is the `Identifier` domain and the second is the `ExpressionList` domain. Thus, every member of the `ProcedureCall` domain has two components: the first is a syntagm in the `Identifier` domain and the second is a syntagm in the `ExpressionList` domain. Informally, we say that every `ProcedureCall` consists of an `Identifier` and an `ExpressionList`.

Terminal symbols such as keywords and punctuation do not define syntactic domains, so they do not constitute component domains of a construction rule, and, in general, do not play a role in the abstract syntax of the target-language. However, they certainly contribute to the concrete syntax since they are vital when parsing and printing textual representations of syntagms.

Within a construction rule, the name of each component domain is prefixed by a unique *component* *name.*[2] These names are not significant with respect to the concrete syntax defined by the rule, but allow one to refer to each component of a construction syntagm in a way that is both unambiguous and suggestive of the component's semantic role in the construction. For example, in the rule for `ProcedureCall`, "`Callee`" is the first component name and "`Arguments`" is the second component name. Informally, we say that the `Callee` of a `ProcedureCall` is an `Identifier` and the `Arguments` of a `ProcedureCall` is an `ExpressionList`.

A component of a construction rule can be specified to be optional. For example, in the programming language Pascal, a `ProcedureCall` normally includes a parenthesized `ExpressionList`, as in the syntax given above. However, if the called procedure has no formal parameters, the `ExpressionList` and the surrounding parentheses are omitted. This optionality is indicated by embedding the optional component and any associated terminal symbols in an *optional phrase* delimited by square brackets. Thus, the syntax of the Pascal `ProcedureCall` can be written:

```
    CONSTRUCT ProcedureCall IS
      <Callee:Identifier> [ "(" <Arguments:ExpressionList> ")" ]
```

The syntactic domain denoted by an optional phrase is the union of the domain of the optional component and the `Empty` domain. (The `Empty` domain is not itself empty, but rather has a single member with no components.) For example, informally we say that the `Arguments` of a Pascal `ProcedureCall` is either an `ExpressionList` or `Empty`.

2.1.2 Repetition rules

Recall that a repetition rule is a rule using closure. Abstractly, a repetition rule identifies a single syntactic domain, called its *element* domain, and defines a syntactic domain that is the closure of its element domain. That is, a member of a repetition rule's domain (a *repetition* syntagm) consists of a sequence of syntagms, each from the element domain. Each of these syntagms is an *element* of the repetition. For example, the repetition rule for `StatementList`:

```
    LIST StatementList OF Statement SEPARATED_BY ";"
```

has `Statement` as its element domain: a member of the `StatementList` domain consists of a sequence of syntagms, each a member of the `Statement` domain. Informally, we say that a `StatementList` is a sequence of `Statements`.

Terry's notation for repetition rules, e.g., `Statement SEPARATED_BY ";"` is clearer than the RRPG notation previously presented, e.g., `Statement (";" Statement)*` in defining the concrete syntax of a repetition where the elements are separated by terminal symbols. The `SEPARATED_BY ...` part of the repetition rule is optional, since not all repetitions have separators.

[2] For convenience, a component name can be omitted, in which case it is taken to be the same as the name of the component domain.

2.1.3 Alternation rules

Finally, recall that an alternation rule is a rule using union. Abstractly, an alternation rule identifies a set of syntactic domains called its ***alternative*** domains, and defines a syntactic domain that is the union of its alternative domains. That is, any syntagm that is a member of one of the rule's alternative domains is a member of the rule's domain. For example, the alternation rule for `Statement`:

```
    ALTERNATE CLOSED Statement IS ProcedureCall | ForLoop
```

has the `ProcedureCall` domain and the `ForLoop` domain as its alternative domains: the `Statement` domain is the union of the `ProcedureCall` domain and the `ForLoop` domain. Informally, we say that a `Statement` is either a `ProcedureCall` or a `ForLoop`.

One might casually say that "every `ForLoop` is a `Statement`", but there is danger in such casualness. Consider the alternation rule for `Expression`:

```
    ALTERNATE OPEN Expression IS Identifier | Numeral
```

Since an `Expression` is either an `Identifier` or a `Numeral`, one might similarly say that "every `Identifier` is an `Expression`". However, observe that the `Callee` of a `ProcedureCall` is an `Identifier`, but is *not* an `Expression`. That is, it appears in a context that does not allow every member of the `Expression` domain to appear. On the other hand, in our tiny language of statements, syntagms in the `Statement` domain *are* always `Statements`. That is, a syntagm in the `Statement` domain can only appear in contexts that allow *any* such syntagm to appear.

This distinction, which is original to this thesis, gives rise to two flavours of alternation rule, *open* and *closed*, indicated by an additional keyword after "`ALTERNATE`". Their form is otherwise the same, and the way in which they define syntactic domains is the same, just as described above. The difference lies in the restrictions placed on the alternative domains.[3] An open alternation rule puts no restrictions on its alternative domains. A closed alternation rule requires that none of its alternative domains be used in the definition of any other rule, or be defined by an open alternation rule. These restrictions ensure that every syntagm in the domain defined by a closed alternation rule can only appear in contexts that allow *any* member of the domain to appear. For example, the `Statement` domain can be defined by a closed rule, but the `Expression` domain must be defined by an open rule.

---

[3] In addition, there is a difference in how they are treated when deriving a GRAMPS-style SMS, as we will see in Section 2.2.

2.1.4 Odds and Ends

In general, the elements of a repetition syntagm and the components of a construction syntagm are referred to as ***children*** of the syntagm.

A syntagm can be a member of several alternation-rule domains, but it must be a member of exactly one construction, repetition, or lexical domain. That domain is called the *class* of the syntagm, and the construction, repetition, or lexical rule that defines that domain is the syntagm's *defining rule.*

A syntagm whose defining rule does not define the start symbol of the grammar is designated a *fragment.* GRAMPS itself does not make this distinction, and GRAMPS-style grammars typically do not have a way to specify the start symbol, but the term is sometimes useful in discussions.

As was pointed out earlier, the notation used here for GRAMPS-style rules is that of [Ter87], with some slight differences. One such difference, already discussed, is the `OPEN/CLOSED` keyword in alternation rules. Another difference is the set of formatting directives used. These directives supplement the concrete syntax defined by the grammar in specifying how a pretty-printer should print a syntagm of the grammar. My implementation of GRAFS recognizes three formatting directives: `_`, `/`, and `%`. Each of these can occur in a construction rule or the separator of a repetition rule, and indicates an action that the pretty-printer should perform any time it reaches the corresponding point when printing out a instance of that rule.

- `_` (`NoSpacer`): Do not insert any white-space. (Normally, the pretty-printer inserts a blank between adjacent tokens.)
- `/` (`LineBreaker`): Insert a line-break.
- `%` (`ConditionalBreaker`): Insert a line-break if the syntagm being printed is too large to fit on a single line.

2.2 <u>GRAMPS-Style SMSs</u>

Recall that a GRAMPS-style SMS is an SMS that has been derived according to the GRAMPS scheme from a GRAMPS-style grammar. This scheme and the form of the resulting SMSs are described in this section. Although the examples will use Modula-2 as the host-language, the SMS would be much the same using any typed, procedural language such as Pascal, Ada, or Clu. The SMS might be significantly different if, for example, the host-language were object-oriented.[4]

[4] See [MadNor88] for a description of an object-oriented metaprogramming system.

2.2.1 Overall organization

As defined in Section 2.1, syntagms are ethereal mathematical objects. In a GRAMPS-style SMS, the corresponding host-language data-type is `Node`; a `Node` is an instance of a syntagm. A syntagm can have many distinct instances; i.e., two distinct nodes can be different instances of the same syntagm. For convenience, we use much of the same terminology for nodes and syntagms; thus, we refer to the components of a construction node, the elements of a repetition node, the defining rule of a node, and the class of a node.

Just as construction and repetition syntagms consist of other syntagms, so construction and repetition nodes contain references to their child nodes. Any given node is the root of a (sub)tree of nodes determined by such references. This tree can be thought of as an abstract syntax-tree for some phrase in the target-language. A parallel set of references link each node to its *parent*, i.e., the construction or repetition node of which it is a child. The operations of a GRAMPS-style SMS create and destroy nodes, and establish and change the references between them, always ensuring the context-free validity of the result, i.e., the SMS guarantees that every node is an instance of some syntagm in the abstract syntax of the target-language.

For a particular target-language, the corresponding SMS defines `NodeDomain`, an enumerated-type; the Identifiers in the enumeration are the names of the rules in a GRAMPS-style grammar for the target-language. For example, for our tiny language of statements defined in Section 2.1, the SMS would have the type-declaration

```
    TYPE NodeDomain =
      ( Empty,
        Statement,
        ProcedureCall,
        ExpressionList,
        ForLoop,
        StatementList,
        Expression,
        Identifier,
        Numeral ) ;
```

The function `GetClass` returns a `NodeDomain` value representing the class of a given `Node`.

```
    PROCEDURE GetClass ( n: Node ) : NodeDomain ;
```

The routines of a GRAMPS-style SMS can be partitioned on the basis of how they depend on the grammar for the target-language:

1. *Grammar-derived* routines are derived in name and possibly parameterization from the rules in a GRAMPS-style grammar for the target-language. The scheme for this derivation is central to GRAMPS and will be presented below.
2. *Generic* routines have the same name and parameterization in all SMSs. For example, the function

GetParent, which returns the parent of a given node (if it has one) or a null value (if it is an *orphan*), is applicable to any node, from any target-language. Other examples are GetNumberOfElements, which returns the number of elements of a repetition node, GetElementAt, which returns the element at a particular position, and Copy, which returns a copy of its argument. A generic routine is not necessarily grammar-independent: its operation may be dependent on some grammar-specific information that it accesses when it is invoked. For example, the generic procedure Delete cannot be correctly applied to just any node; only an element of a repetition node or an optional component of a construction node can be deleted. When Delete is invoked, it may have to consult the target-language grammar to determine if the requested deletion is valid. In practice, generic routines requiring grammar-specific information do not necessarily extract it from the grammar. Instead, all pertinent information is extracted from the grammar when the SMS is constructed, and saved in data structures that allow faster access. For example, this would almost certainly be the case for parsing routines.

Sometimes there is a choice as to whether to provide a facility of an SMS as a grammar-derived or generic routine. We will describe an example of this situation after discussing the GRAMPS derivation scheme in the next subsection.

This work has been done using a version of Terry's GRAFS system [Ter87], which uses the GRAMPS scheme to automatically generate a context-free SMS from a GRAMPS-style grammar. More precisely, given a grammar, GRAFS generates a Modula-2 module defining

1. a grammar-derived NodeDomain data-type,
2. all of the grammar-derived routines, and
3. some of the generic routines for that grammar (those that take or return a NodeDomain value).

It also generates two binary files which encode the grammar and pertinent grammar-specific information. The rest of the generic routines reside in two permanent modules, which are conceptually part of every SMS that GRAFS generates.

2.2.2 The GRAMPS derivation scheme

For each kind of rule in a GRAMPS-style grammar, the GRAMPS scheme specifies how to derive, for any rule of that kind, a set of routines for handling nodes in that rule's domain. The derivation scheme presented here is that used in my modified implementation of Terry's GRAFS system. The scheme differs from that of the Gramps Report [CamIto84], but mainly in minor details. The only major difference is in its recognition and treatment of two different flavours of alternation rule.

Every construction rule, repetition rule, and lexical rule yields a *recognizer*, a boolean function that ascertains whether its argument is in that rule's domain. Here, the name of the function is obtained by prepending `Is_a_` to the name of the rule. For example, recall the tiny grammar of statements given in Section 2.1; the two construction rules, two repetition rules, and two (omitted) lexical rules of this grammar yield the following recognizers:[5]

```
PROCEDURE Is_a_ProcedureCall (X:Node) : BOOLEAN ;
PROCEDURE Is_a_ForLoop (X:Node) : BOOLEAN ;
PROCEDURE Is_a_ExpressionList (X:Node) : BOOLEAN ;
PROCEDURE Is_a_StatementList (X:Node) : BOOLEAN ;
PROCEDURE Is_a_Identifier (X:Node) : BOOLEAN ;
PROCEDURE Is_a_Numeral (X:Node) : BOOLEAN ;
```

Every open alternation rule yields two recognizers. The *syntagmatic* recognizer, whose name is of the form `Is_in_the_X_domain`, simply checks whether a given node is a member of the alternation domain `X` defined by the rule. The *contextual* recognizer, whose name is of the form `Is_in_a_X_context`, checks whether a given node is being used in its capacity as a member of domain `X`, that is, whether it is in a context that allows any member of domain `X` to appear. For example, the open alternation rule for `Expression` yields the following recognizers:

```
PROCEDURE Is_in_the_Expression_domain (X:Node) : BOOLEAN ;
PROCEDURE Is_in_a_Expression_context (X:Node) : BOOLEAN ;
```

The first recognizer returns `TRUE` for any `Identifier` or `Numeral`; the second returns `TRUE` for an `Identifier` or `Numeral` only if it stands as an `Expression`, i.e., only if it is in a context that allows any member of the `Expression` domain. Specifically, the latter recognizer returns `FALSE` for the `Callee` of a `ProcedureCall`, for the `LoopVariable` of a `ForLoop`, and for an orphan `Identifier` or `Numeral`.

On the other hand, every closed alternation rule yields a single recognizer. Because of the restrictions placed on its alternative domains, its syntagmatic and contextual recognizers would be essentially equivalent[6], and so can be combined into a single routine. For example, the closed alternation rule for `Statement` yields the following recognizer:

```
PROCEDURE Is_a_Statement (X:Node) : BOOLEAN ;
```

Recognizers are the only routines derived for alternation and repetition rules, but construction and lexical rules yield additional routines, as described below.

Each construction rule yields a *constructor*, a function that creates an instance of that rule from an appropriate set of component nodes. The name of the function is obtained by prepending `Make_` to the name of the rule, and the names of the formal parameters are simply the names of the rule's components.

[5] For brevity, only the headings of these routines are shown here, as they would appear in a Modula-2 definition module.

[6] They would only differ for orphan nodes, a case that can easily be detected if it is deemed important.

For example, the `ProcedureCall` rule yields the constructor:

```
PROCEDURE Make_ProcedureCall (Callee,Arguments: Node) : Node ;
(* precondition: Is_a_Identifier(Callee)
   AND Is_a_ExpressionList(Arguments) *)
```

The comment (delimited by `(*` and `*)`) states a constraint that `Make_ProcedureCall` enforces on its arguments to ensure the context-free validity of the node it creates. To build a construction node with an optional component omitted, an empty `Node` is passed in as the actual parameter. An empty `Node` can be obtained by calling the generic constructor `Make_Empty()`.

Each construction rule also yields *selectors*, one for each component, with names of the form `The_<component>_of`. When applied to a node defined by that construction rule, a selector returns the corresponding component of the node. For example, the `ProcedureCall` rule, with its `Callee` and `Arguments` components, yields two selectors:

```
PROCEDURE The_Callee_of (X:Node) : Node ;
   (* precondition: Is_a_ProcedureCall(X) *)
PROCEDURE The_Arguments_of (X:Node) : Node ;
   (* precondition: Is_a_ProcedureCall(X) *)
```

If the same component name is used in more than one construction rule, an overloaded selector is derived. It looks the same as other selectors, but it applies to more than one class of node. For example, if function-calls, defined by the following rule:

```
CONSTRUCT FunctionCall IS
 <Callee:Identifier> "(" <Arguments:ExpressionList> ")"
```

were added to the `Expression` domain, then the selectors `The_Callee_of` and `The_Arguments_of` would be declared as before, but they would each have the precondition: `Is_a_ProcedureCall(X) OR Is_a_FunctionCall(X)`.

As with construction rules, each lexical rule also yields a constructor, a function to create an instance of that rule. Typically, a lexical constructor takes a single argument representing the spelling of the resulting lexeme. For example, a lexical rule defining the class `Identifier` might yield the constructor:

```
PROCEDURE Make_Identifier ( str : ARRAY OF CHAR ) : Node ;
```

The recognizers, constructors, and selectors derived by the scheme outlined above constitute the grammar-derived routines of a GRAMPS-style SMS.

The reader may have noticed that no means of creating a repetition node has been shown. This is an example of a facility that can be provided by either grammar-derived or generic routines, and both approaches have been used in the various implementations of GRAMPS. In the GRAMPS Report, each repetition rule yielded two constructors: one to create a node with no elements and one to create a single-element node. For example, the rule defining `StatementList` would yield the two grammar-derived constructors:

```
    PROCEDURE Null_StatementList () : Node ;
    PROCEDURE Make_StatementList ( Statement: Node ) : Node ;
```

One could then use generic routines (such as `Concat` or `Append`) to build up larger repetitions. In more recent GRAMPS-style SMSs (e.g., [Cam87] and my implementation of GRAFS), generic routines are provided to create repetition nodes; the class of `Node` desired is specified by passing in a `NodeDomain` parameter. For example, the generic routines

```
    PROCEDURE List0 ( rep_class: NodeDomain ) : Node ;
    PROCEDURE List1 ( rep_class: NodeDomain ; e1 : Node ) : Node ;
    PROCEDURE List2 ( rep_class: NodeDomain ; e1,e2 : Node ) : Node ;
```

create repetition nodes (of class `rep_class`) of zero, one, or two elements. Again, repetition nodes with more elements can be built up using `Concat` or `Append`. One advantage of the latter approach is that the resulting SMS can be significantly smaller, since there are only a small, fixed number of generic routines instead of two grammar-derived routines for each repetition rule. In addition, if the class of repetition node desired is to be determined dynamically, this approach is easier to use, since the desired `rep_class` can be the value of an expression.

The choice between grammar-derived routines and generic routines for providing some SMS facility has been raised here because it will reoccur when considering context-dependent SMSs.

CHAPTER 3
NURN, A NOTATION FOR DEFINING CONTEXT-DEPENDENT LANGUAGES

An SMS is derived (either manually or automatically) from a grammar for the target-language. To derive a context-dependent SMS, we need a notation for specifying the syntax of a context-dependent target-language. In this chapter, I introduce such a notation, called NURN, which formalizes descriptive techniques often found in informal language definitions. Section 3.1 presents the main ideas behind NURN, and Section 3.2 shows how these ideas can be applied to the scoping rules of statically scoped programming languages. Section 3.3 continues this example, showing how the various features of NURN can be used to formalize these scoping rules. Section 3.4 examines how NURN can be used to specify Pascal's type system.

3.1 Describing Context-Dependent Languages: the NURN Approach

NURN (a Notation Using Relations on Nodes) is a systematic means for describing context-dependent languages in terms of relationships between the nodes of abstract syntax-trees[1]. NURN is modelled after descriptive techniques found in informal definitions of context-dependent languages, but enforces and allows greater consistency. This section will explain these ideas in more detail.

The definition of a context-dependent language is almost always based on a context-free grammar that defines a context-free superset of the language. This superset will be called the ***super-language.*** The rest of the definition then specifies which sentences in the super-language are not sentences in the context-dependent language. This winnowing is done in various ways. An informal description of a programming language typically starts by introducing context-dependent entities such as ***defining occurrence***, ***scope***, ***type***, and ***value***. Next, the description shows how these context-dependent entities relate to each other and to the context-free entities defined by the context-free grammar. For example, the following relationships might be described:

- a defining occurrence is visible throughout its scope,
- an identifier may denote a type,
- two types may be compatible,
- a static expression denotes a value,
- a labelled statement may be the target of a goto-statement, and
- a statement may threaten a variable.

Finally, certain relationships between entities are required or prohibited. For example, the following

[1] Actually, there are three classes of entity that NURN deals with. Nodes are normally the focus of a NURN grammar, but integers and character sequences are also used.

might be asserted:

- an identifier is required to have exactly one defining occurrence,
- an argument and its corresponding parameter must have compatible types,
- a goto-statement is prohibited from having a target outside certain contexts, and
- statements in certain contexts are prohibited from threatening certain variables.

The frequent use of this descriptive technique (notably in official language standards, such as those for Pascal [ANSI83] and Ada [DoD83]) suggests that it is a natural way to define context-dependent languages, and thus that a formalism based on this technique would be relatively easy to use. Moreover, a context-dependent SMS based on such a description would be able to answer questions about entities and the relationships between them.

Existing formalisms for defining context-dependent languages do not follow this descriptive technique. For example, Attribute Grammars [Knu68] and the Vienna Definition Method [BjoJon82] are two of the most popular formalisms for specifying the syntax of context-dependent languages[2]; both ascertain the well-formedness of a syntactic object by consulting data structures which are constructed to encode just the information essential to this task.

In NURN, a GRAMPS-style grammar is used to define the context-free syntax, and thus, the syntactic domains, of the super-language. To define the domains of context-dependent entities, GRAMPS-style rules are also used. This uniform treatment of context-free and context-dependent entities is not new with NURN: many other researchers (e.g., [BjoJon82, DeRJul80, RepTei84, BahSne86]) have observed that context-dependent domains are described well by context-free grammars. However, in defining context-free syntax, they use their uniform notation to define *abstract* syntax only, and either ignore concrete syntax or use another formalism to define it. As with GRAMPS, NURN does not separate concrete and abstract syntax: a GRAMPS-style grammar specifies both.

Note that when context-dependent domains are defined using context-free rules, the distinction between "context-free" and "context-dependent" entities becomes blurred; one is less justified in making the distinction. Previous work on adding context-dependent facilities to existing GRAMPS-style SMSs [Cam87, Mer88] has erased the distinction even further by not having separate rules to define the context-dependent domains; instead, certain domains defined in the context-free syntax of the super-language are used to represent context-dependent domains, and nodes in the context-free syntax-tree are used to represent context-dependent entities. Viewed in a different way, there are no context-dependent entities *per se*; rather, some context-free entities have an additional context-dependent role. This strategy for representing context-dependent entities will be referred to as the ***double-duty***

[2] Both can be used more generally to specify dynamic semantics.

strategy.

For example, in Pascal, the context-dependent domain of *type* can be represented with the context-free domain of `TypeDefiner`. That is, for every type that might arise in determining the correctness of a `Program`, there is a corresponding `TypeDefiner` that the `Program` contains[3], and that `TypeDefiner` can be used to represent that type.

Double-duty is an attractive strategy, where the target-language permits its use, since it avoids the introduction of extraneous domains. The examples used to introduce NURN in this chapter will follow it.

To define relationships between entities, a NURN grammar contains rules that define mathematical relations on the domains of entities. An *instance* of a relation is a tuple of entities that satisfy the definition of the relation. The relations defined by a NURN grammar fall into two groups: *violation relations* and *support relations*. An instance of a violation relation indicates a particular violation of the context-dependent syntax. Support relations are used to define violation relations. The context-dependent language is then defined to be the set of sentences in the super-language that do not give rise to any instances of violation relations.

From the language-theoretic point of view, the support relations are only a means to an end: a relation is useless unless it ultimately participates in some constraint. However, from a descriptive point of view, the support relations can be just as important as the violation relations: even if a relation does not participate in a violation, it can still help define the semantics of the language. The use of support and violation relations will become clearer in the next section.

3.2 Extended Example of the NURN Approach

Let us consider an extended example, demonstrating how NURN can be used to define the scoping rules of a statically scoped programming language. This section will introduce the NURN approach, showing how the concepts in scoping rules can be formalized as entities and relations, and Section 3.3 will show how the relations can be defined for Pascal, as specified in the Pascal Standard [ANSI83].

The following Pascal program will be used to make the discussion more concrete:

[3] Actually, this is not quite the case, as we will see in Section 3.4.

```
01  PROGRAM example ;
02  VAR c : goop ;
03
04    PROCEDURE P ( a : real ) ;
05    TYPE a = boolean ;
06    VAR c : char ;
07    BEGIN
08      c := 'j' ;
09    END ;
10
11  BEGIN
12    P ( 3.14 )
13  END .
```

Suppose we want to know if the assignment on line 08 is valid. To answer this question, it is first necessary to know what the identifier `c` on line 08 denotes. This information is context-dependent, since it cannot be expressed by a context-free grammar. Any statically scoped programming language has, by definition, a set of scoping rules, which determine the entity that a lexeme (e.g., an identifier) denotes. The scoping rules for Pascal indicate that the identifier `c` on line 08 denotes a variable of type `char`, which allows other rules to determine that the assignment is valid. Most languages descended from Algol 60 [Nau60] have similar scoping rules, and, given a similar program, would agree that the assignment is valid. Thus, although the discussion in the following sections focusses on the scoping rules of Pascal, it could be applied with minor changes to many other languages.

To describe Pascal scoping rules using the NURN approach, the concepts involved must be formalized as entities and relations. We will see that the pertinent context-dependent entities are *region* and *declaration-point*, and the pertinent relations are

`is_effective_over`,

`is_the_defining_occurrence_of`,

`is_undefined`, and

`is_a_conflicting_declaration_point`.

The description will proceed by first examining the two entities and then each of the relations in turn.

3.2.1 The `declaration_point` *and* `region` *entities*

The bulk of Pascal's scoping rules specify how the occurrence of some lexeme (an identifier or label) in the context of a declaration associates the spelling of that lexeme with some context-dependent entity (e.g., a type, variable, or procedure) over the extent of some *region* of the program. We say that the lexeme is a *declaration-point* and that it is *effective over* that region.

We can formalize the entities *declaration-point* and *region* with two unary relations `is_a_declaration_point` and `is_a_region`.

The most familiar example of a region is the block. However, a formal–parameter–list, a record–type, the body of a with–statement, or the field–identifier of a selected–variable can also be a region. Each of these five possibilities for *region* identifies a context–free syntactic domain. Thus, the double–duty strategy of Section 3.1 is applicable; we can represent each region as a node in the syntax–tree of a Pascal program. There are 3 regions (instances of `is_a_region`) in the example program:

- the block of the program (lines 02 to 13)
- the formal–parameter–list on line 04
- the block of procedure `P` (lines 05 to 09)

Double–duty also applies to the context–dependent entity *declaration-point*, since every declaration–point is a lexeme, and every lexeme is certainly a context–free entity. There are 5 declaration–points (instances of `is_a_declaration`) in the example program:

- the identifier `c` on line 02
- the identifier `P` on line 04
- the identifier `a` on line 04
- the identifier `a` on line 05
- the identifier `c` on line 06

3.2.2 The `is_effective_over` *relation*

The connection between a declaration–point and a region over which it is effective can be represented with the binary support relation `is_effective_over`, so that

```
    dp is_effective_over r
```

denotes that the declaration–point `dp` is effective over the region `r`. Here, `dp` and `r` are place–holders, standing for nodes. One can liken them to the formal parameters of a procedure. There are 6 instances of this relation in the example program:

- the identifier `c` on line 02 `is_effective_over` the block of the program
- the identifier `P` on line 04 `is_effective_over` the block of the program
- the identifier `a` on line 04 `is_effective_over` the formal–parameter–list it appears in
- the identifier `a` on line 04 `is_effective_over` the block of procedure `P`
- the identifier `a` on line 05 `is_effective_over` the block of procedure `P`
- the identifier `c` on line 06 `is_effective_over` the block of procedure `P`

3.2.3 The `is_the_defining_occurrence_of` *relation*

Pascal scoping rules dictate that to determine the entity denoted by a given lexeme, one starts at the lexeme and moves outward (rootward in the syntax-tree) until one finds a region with a declaration-point effective over it whose spelling matches that of the lexeme. The declaration-point so found is the ***defining occurrence*** of that lexeme, and the lexeme denotes whatever its defining occurrence denotes. We can express this correspondence using another binary support relation:

```
d is_the_defining_occurrence_of x
```

In the example program, to discover what the `c` on line 08 denotes, we search for a region that has a declaration-point spelled "`c`" effective over it. There are two such regions, and the first we encounter, moving outward from the `c` on line 08, is the block of procedure `P` (lines 05 to 09). Thus, the `c` on line 06 `is_the_defining_occurrence_of` the `c` on line 08.

3.2.4 The `is_a_conflicting_declaration_point` *relation*

In Pascal, each lexeme must have exactly one defining occurrence. The Pascal Standard achieves this with two scoping rules; the first ensures that a lexemes has *at least* one defining occurrence, and the second ensures that it has *at most* one:

- Each lexeme must have a defining occurrence.
- The declaration-points effective over a given region must have distinct spellings.

Since these rules are expressing constraints that must be satisfied by programs, we formalize them as two violation relations:

```
x is_undefined
```

if `x` is a label or identifier without a defining occurrence, and

```
d is_a_conflicting_declaration_point
```

if `d` is a declaration-point that duplicates the spelling of another declaration-point effective over the same region. In the example program, there is one instance of the first relation:

- the identifier `goop` on line 02 `is_undefined`

because it has no defining occurrence. There are two instances of the second relation:

- the identifier `a` on line 04 `is_a_conflicting_declaration_point`, and
- the identifier `a` on line 05 `is_a_conflicting_declaration_point`,

because they have the same spelling and are both declaration-points effective over the block of procedure `P`.

3.2.5 Summary

This section has shown how the concepts involved in the scoping rules of Pascal (and thus, of many Algol-like programming languages) can be formalized with two context-dependent entities, two support relations, and two violation relations. The next section expands on this example, showing how to define these entities and relations using NURN.

3.3 Defining NURN Relations

A NURN grammar consists of GRAMPS-style rules, which define domains of entities, and *relation rules*, which define relations between these entities. Relations can be defined in two ways, using the three different kinds of relation rule. (See Appendix 1 for a GRAMPS-style grammar for NURN rules.) A relation can either be defined completely by a single `Definition` rule or else declared once with a `Declaration` rule and then "subdefined" any number of times with `Subdefinition` rules. Having two ways to define relations is simply a notational convenience for the grammar-writer; there is a straightforward equivalence between them. As another convenience, a `Subdefinition` rule may subdefine more than one relation, but for simplicity, we will ignore this.

A `Definition` rule (and conceptually, a `Declaration` rule and its corresponding `Subdefinition` rules) consists of two parts, a head and a body. The *head* is a `Declarator`, which:

1. includes a `Simple_Primary`, which names the relation that the rule defines and supplies identifiers to represent its parameters.
2. indicates the kind of the relation (whether it is primitive, derived, violation, maker, or normal)
3. indicates whether it is functional on any of its parameters.

These indicators are optional, with sensible defaults. The *body* gives a logical formula in terms of relations on these parameters and other variables, which can be introduced as necessary. Here is a somewhat contrived example of a `Definition` rule:

```
RELATION DEF NORMAL x_1 is_a_cousin_of x_2: FUNCTIONAL ON {x_1,x_2} :-
  gp is_the_parent_of p_1, p_1 is_the_parent_of x_1,
  gp is_the_parent_of p_2, p_2 is_the_parent_of x_2,
  NOT (p_1 is p_2)

(* Informally, two nodes x_1 and x_2 are cousins if
   they have the same grandparent gp, but
   different parents.
*)
```

Every relation rule begins with the keyword `RELATION`, followed by either `DEF`, `DECL`, or `SUBDEF`, indicating whether the rule is a `Definition`, `Declaration`, or `Subdefinition`, respectively. The head and body are separated by the "if" symbol "`:-`", and the end of the rule is indicated with a period.

Comments appear between "(*" and "*)".

In the example, the head of the rule is the `Declarator`:

```
    NORMAL x_1 is_a_cousin_of x_2: FUNCTIONAL ON {x_1,x_2}
```

The binary relation `is_a_cousin_of` is being defined, and its two parameters are denoted by `x_1` and `x_2`. It is a "normal" rule. Since this is the default, the keyword `NORMAL` could have been omitted. The relation is functional on the set of both its parameters. That is, one must bind both `x_1` and `x_2` to ensure that there is at most one instance of the relation satisfying the bindings. Since "functional only on the set of all parameters" is the default, this too could have been omitted.

The body of the rule in the example consists of a *conjunction* of five *primaries*, separated by commas. (In general, the body of a rule can be a *disjunction* of conjunctions, separated by "`OR`".) Four of the primaries use the relation `is_the_parent_of`, which is a primitive relation, relating any node to its parent (if it has one). The last primary is a *negation* involving the identity relation `is`, another primitive relation.

In this rule, the parameters (`x_1` and `x_2`) and the internal variables (`gp`, `p_1`, and `p_2`) are all node variables, and as such, can only be bound to nodes in abstract syntax-trees. Although node variables are the most used, there are also integer variables and character-sequence variables. That is, variables in a NURN grammar are "typed" according to the kind of entity they bind to. This typing is indicated by the first character of the variable's name: integer variables begin with `#`, character-sequence variables begin with `$`, and node variables have no special prefix.

Readers familiar with the programming languagc Prolog [SteSha86] should have no trouble with NURN's relation rules. They are both based on a subset of first-order logic, and both have a declarative semantics. There are also some differences: Prolog rules have a procedural semantics involving depth-first execution, whereas NURN's relation rules have no particular procedural interpretation, and are instead transformed into an equivalent form that can be evaluated efficiently. Also, Prolog allows the arguments of a relation to be structured with functions, whereas NURN requires that they be simple identifiers. In these two respects, relation rules resemble the deductive laws (extensional relations) of a deductive database [GMN84].

For definiteness, relation rules will be introduced by continuing the extended example of Section 3.2. Specifically, this section will show how to define the relations

```
    is_effective_over,
    is_a_declaration_point,
    is_a_region,
    is_the_defining_occurrence_of,
    is_undefined, and
    is_a_conflicting_declaration_point,
```

(in that order) for the programming language Pascal. Occasional minor simplifications have been made to ease the explanation here; see Appendix 2 for a complete NURN specification of Pascal.

There are various sources of leeway in writing a NURN grammar. One that we will see a couple of times involves syntactic objects within fragments. While language descriptions are usually not concerned with the correctness of any construct "smaller" than a program, such constructs commonly occur as fragments in SMSs, and there is the question of how a NURN grammar should treat them. For example, if the assignment-statement `c:='j'` occurs as a fragment, should the identifier `c` be an instance of the relation `is_undefined`? The Pascal Standard is not written to answer such questions, and the grammar-writer is free to do what he likes.

3.3.1 Predefined relations

Since a NURN grammar defines relations in terms of other relations, there must be some predefined relations that the grammar can use without defining. Similar to the division of routines of a GRAMPS-style SMS, these predefined relations can be divided into *grammar-derived* and *primitive.*

Here are declarations for the primitive relations that this section will use:

```
RELATION DECL PRIMITIVE n_1 is n_2: FUNCTIONAL ON {n_1}, {n_2} .
(* the identity relation on nodes *)

RELATION DECL PRIMITIVE p is_the_parent_of n: FUNCTIONAL ON {n} .
(* p is the parent of n *)

RELATION DECL PRIMITIVE a contains d: FUNCTIONAL ON {a,d} .
(* a is an ancestor of d *)

RELATION DECL
  PRIMITIVE $s is_the_character_sequence_of n: FUNCTIONAL ON {n} .
(* $s is the sequence of characters in lexeme n *)

RELATION DECL PRIMITIVE #1 is_the_length_of n: FUNCTIONAL ON {n} .
(* #1 is the number of elements in repetition node n *)

RELATION DECL
  PRIMITIVE $z IS_THE_BASE_TEN_REP_OF #i: FUNCTIONAL ON {$z}, {#i} .
(* $z is the base-ten representation of integer #i *)
```

As the name suggests, the grammar-derived relations are derived from the GRAMPS-style rules in the NURN grammar. From the NURN grammar for Pascal in Appendix 2, some of the GRAMPS-style rules relevant to the current discussion are as follows:

```
CONSTRUCT Block IS
  [ "label" <:LabelList> ";" ]
  [ "const" <:ConstantDefinitionList> ]
  [ "type" <:TypeDefinitionList> ]
  [ "var" <:VariableDeclarationList> ]
  <:RoutineDeclarationList>
  "begin"
    <:StatementSequence>
  "end"

LIST NONEMPTY LabelList OF Label SEPARATED_BY _ ","

LIST NONEMPTY ConstantDefinitionList OF ConstantDefinition
CONSTRUCT ConstantDefinition IS
   <Lhs:Identifier> "=" <Rhs:Constant> ";"

LIST NONEMPTY TypeDefinitionList OF TypeDefinition
CONSTRUCT TypeDefinition IS
   <Lhs:Identifier> "=" <Rhs:TypeDenoter> ";"
```

In the spirit of the GRAMPS derivation scheme presented in Section 2.2, these context-free rules yield four kinds of context-free relations:

1. *Domain relations:* These unary relations correspond to GRAMPS-style recognizers. For example, `b is_a_Block` is true when node `b` is a `Block`, and `tdl is_a_TypeDefinitionList` is satisfied when `tdl` is a `TypeDefinitionList`.
2. *Component relations:* These binary relations correspond to GRAMPS-style selectors. For example, `ll is_the_LabelList_of b` indicates that the node `ll` is the `LabelList` component of `b`, a `Block`; `id is_the_Lhs_of x` is satisfied when the node `id` is the `Lhs` component of `x`, which is a `ConstantDefinition` or a `TypeDefinition` or any other class of node that has a `Lhs` component.
3. *Element relations:* These binary relations have no direct analog among GRAMPS-style routines. For example, `l is_a_Label_in ll` is true if the node `l` is a `Label` in the `LabelList ll`.
4. *Element-selection relations:* These ternary relations correspond to the generic routine `GetElementAt`. For example, `l is_the #i th_Label_in ll` is true if node `l` is the `Label` at position `#i` in the `LabelList ll`.

3.3.2 Relation rules for `is_effective_over`

Recall that the support relation `is_effective_over` deals with the relationship between a declaration-point and a region over which it is effective. We can declare this relation as follows:

```
RELATION DECL dp is_effective_over r .
(* Declaration-point dp is effective over region r. *)
```

We can then subdefine this relation with a set of `Subdefinition` rules. In the NURN grammar for

Pascal in Appendix 2, there are 13 subdefinitions for `is_effective_over`; since they are somewhat repetitious, only 3 are presented here. The first subdefinition deals with the declaration of labels:

```
(* A Label in the LabelList of a Block
   is a declaration-point effective over the Block.
*)
RELATION SUBDEF l is_effective_over b :-
  l is_a_Label_in ll,
  ll is_the_LabelList_of b,
  b is_a_Block
.
```

This rule can be read in many ways. As a logical formula, it could be read fairly literally as: "for all nodes `l`, `b`, and `ll`: `l` is effective over `b` if `l` is a `Label` in `ll`, and `ll` is the `LabelList` component of `b`, and `b` is a `Block`." A more casual reading would abbreviate this to: "`l` is effective over `b` if `l` is a `Label` in the `LabelList` of `b`, a `Block`." The comment shows how it would be rendered in the style of the Pascal Standard. Note the close correspondence between the NURN definition and a careful wording of the same idea in English.

For example, consider the following Pascal program:

```
01    PROGRAM example ;
02    LABEL 10, 99 ;
03    BEGIN
04      10: Writeln ( 'hello world' ) ;
05      GOTO 10
06    END .
```

There are two instances of `is_effective_over` in this program. The first is that

> the label `10` on line 02 `is_effective_over` the block of the program

because

> the label `10` on line 02 `is_a_Label_in` the label-list,
> the label-list `is_the_LabelList_of` the block of the program, and
> the block of the program `is_a_Block`.

That is, the following bindings for the variables `l`, `ll`, and `b` satisfy the subdefintion of `is_effective_over` given above:

> `l` is the label `10` on line 02,
> `ll` is the label-list, and
> `b` is the block of the program.

The second instance of `is_effective_over` in the example program is given by

> `l` is the label `99` on line 02,
> `ll` is the label-list, and
> `b` is the block of the program.

which corresponds to the declaration of label `99`.

Another subdefinition of `is_effective_over`, similar to that for labels, deals with the declaration of constant-identifiers:

```
(* The Lhs of a ConstantDefinition in the ConstantDefinitionList
   of a Block is a declaration-point effective over the Block.
*)
RELATION SUBDEF id is_effective_over b :-
  id is_the_Lhs_of cd,
  cd is_a_ConstantDefinition_in cdl,
  cdl is_the_ConstantDefinitionList_of b,
  b is_a_Block
.
```

Both of the rules shown so far subdefine the `is_effective_over` relation entirely in terms of predefined relations. Similar rules exist for the declarations of type-identifiers, variable-identifiers, procedure-identifiers, function-identifiers, parameter-identifiers, and field-identifiers; little would be gained by presenting them all here.[4] Instead, it is instructive to examine a rule which is not in this vein. This rule deals with the identifiers of an enumerated-type. The pertinent GRAMPS-style rules are as follows:

```
CONSTRUCT EnumeratedTypeDefiner IS "(" <:IdentifierList> ")"

LIST NONEMPTY IdentifierList OF Identifier SEPARATED_BY _ ","
```

Normally, one might declare an enumerated-type as:

```
TYPE Primary = (Red, Green, Blue) ;
```

Not only does this declare the type-identifier `Primary`, it also declares the identifiers `Red`, `Green`, and `Blue` to denote the three values of the enumerated-type that `Primary` denotes. However, an enumerated-type need not be declared by binding it to an identifier in this way. One can also write

```
TYPE Colour = ARRAY [(Red, Green, Blue)] OF Intensity ;
```

or

```
VAR PrimarySet : SET OF (Red, Green, Blue) ;
```

There are many such contexts in which an enumerated-type can appear, but the effect is always that the identifiers of the enumerated-type are declared over the block that most closely contains the enumerated-type. Here is a rule that captures this idea:

[4] Actually, some of the rules dealing with procedure-identifiers, function-identifiers, and parameter-identifiers are complicated by forward-declarations. The rules are still fairly straightforward, but presenting them here would take too much time developing and explaining new relations, without adding much to the discussion.

```
(* An Identifier in the IdentifierList of an EnumeratedType
   is a declaration-point effective over
   the Block closest-containing the EnumeratedType.
*)
RELATION SUBDEF id is_effective_over b :-
  id is_an_Identifier_in idl,
  idl is_the_IdentifierList_of etd,
  etd is_a_EnumeratedTypeDefiner,
  b is_the_Block_closest_containing etd
.
```

In the body of this rule, the first three primaries involve predefined context-free relations, but the last uses `is_the_Block_closest_containing`, which is not predefined. To define it, we can write a recursive rule that starts at a node and climbs up the syntax-tree until it finds a `Block`:

```
RELATION DEF b is_the_Block_closest_containing x: FUNCTIONAL ON {x} :-
  IF (x is_a_Block) THEN
    b is x
  ELSE
    p is_the_parent_of x,
    b is_the_Block_closest_containing p
  END
.
```

This rules states that if `x` is a `Block`, the `Block` closest-containing `x` is `x` itself; otherwise, the `Block` closest-containing `x` is the `Block` closest-containing `p`, the parent of `x`. The phrase

```
FUNCTIONAL ON {x}
```

indicates that for any node `x`, there is at most one `Block` closest-containing `x`. This property of a relation could conceivably be deduced from the grammar, but someone reading the grammar should not have to perform this deduction, so it is better made explicit.[5]

The `IF-THEN-ELSE` construct is provided as another notational convenience. The above rule is equivalent to

```
RELATION DEF b is_the_Block_closest_containing x: FUNCTIONAL ON {x} :-
  x is_a_Block,
  b is x
OR
  NOT (x is_a_Block),
  p is_the_parent_of x,
  b is_the_Block_closest_containing p
.
```

which fails to show the intent of the rule as clearly.

One final subdefinition of `is_effective_over` will be presented, because it shows that the relation is in fact recursive. It concerns with-statements. Consider the following block:

[5] We will see in Chapter 4 that Ginger issues a warning if it cannot verify the stated functionality of a relation.

```
01  VAR rec: RECORD a: integer END ;
02  BEGIN
03    rec.a:=1 ;
04    WITH rec DO a:=1
05  END
```

The two assignments are equivalent, because the with-statement "opens up a scope" in which `a`, the field-identifier of the record-variable `rec`, can be referenced without the `rec.` prefix that is necessary in the first assignment. In fact, the `rec.` prefix in the first assignment itself opens up a scope (only one identifier "wide") in an analogous manner. Anyway, here is a somewhat simplified syntax for the with-statement[6] and two associated subdefinitions, one for the violation relation `has_an_inappropriate_type`, and one for `is_effective_over`:

```
CONSTRUCT WithStatement IS
  "with" <:VariableAccess> "do" <Body:Statement>

(*
The VariableAccess of a WithStatement
 shall be a variable possessing a record-type.
*)
RELATION SUBDEF va has_an_inappropriate_type :-
  ws is_a_WithStatement,
  va is_the_VariableAccess_of ws,
  rt is_the_type_of va,
  NOT (rt is_a_record_type)
.

(*
An Identifier that is effective over
 (i.e., is a field-identifier for)
 the record-type possessed by
 the VariableAccess of a WithStatement
 is a declaration-point
 effective over the region that is
 the Body of the WithStatement.
*)
RELATION SUBDEF id is_effective_over st :-
  ws is_a_WithStatement,
  st is_the_Body_of ws,
  va is_the_VariableAccess_of ws,
  rt is_the_type_of va,
  id is_effective_over rt
.
```

The variables in this subdefinition can be bound consistently to nodes in the example block as follows:

`ws`: `WITH rec DO a:=1` on line 04
`st`: `a:=1` on line 04
`va`: `rec` on line 04

---

[6] The simplification is to have a `VariableAccess` component rather than a `VariableAccessList`. The true syntax for the with-statement is dealt with in Subsection 3.3.7.

```
rt: RECORD a: integer END on line 01
id: a on line 01
```

Although relations `is_the_type_of` and `is_a_record_type` have not been seen, their meaning is as expected.

We have seen four subdefinitions of `is_effective_over`, and mentioned others as being similar. Together, they fully define the `is_effective_over` relation. A reader familiar with Pascal may be wondering how the predefined identifiers such as `char` and `ord` are covered by this definition, since according to the Pascal Standard, they should act as if they had declaration-points effective over the program. Subsection 3.3.6 will explain how.

3.3.3 Relation rules for `is_a_declaration_point` *and* `is_a_region`

To define the `is_a_declaration_point` relation, one could enumerate all the contexts in which a lexeme is a declaration-point:

```
RELATION DEF dp is_a_declaration_point :-
  dp is_a_Label_in ll,
  ll is_the_LabelList_of b,
  b is_a_Block
OR
  dp is_the_Lhs_of cd,
  cd is_a_ConstantDefinition
OR
  dp is_a_Identifier_in idl,
  idl is_the_IdentifierList_of etd,
  etd is_a_EnumeratedTypeDefiner
OR
  . . .
.
```

However, this is unnecessary; the definition of the relation `is_effective_over` has already identified all of these contexts. Thus, we can say that a node is a declaration-point if there is some region that it is effective over. Formally:

```
RELATION DEF dp is_a_declaration_point :-
  dp is_effective_over r
.
```

There is a small area of disagreement between the two definitions. For example, the `Lhs` of an orphan `ConstantDefinition` would be an instance of the first definition of `is_a_declaration_point`, but not of the second, because there is insufficient context (a `Block`) to establish the necessary instance of `is_effective_over`. Because this disagreement occurs only for nodes within fragments, it is outside the jurisdiction of the Pascal Standard. If the grammar-writer decides that the difference is unimportant, the second definition is preferable, since it is less repetitious and less error-prone than the first.

Similarly, for the `is_a_region` relation, we can write:

```
     RELATION DEF r is_a_region :-
       dp is_effective_over r
     .
```

3.3.4 A relation rule for `is_the_defining_occurrence_of`

Defining the binary support relation `is_the_defining_occurrence_of` is less straightforward. Recall that to find the defining occurrence of a lexeme, one starts at the lexeme and moves rootward in the syntax-tree until one finds a region with a declaration-point effective over it whose spelling matches that of the lexeme. The idea here is similar to that behind `is_the_Block_closest_containing`: in both cases, given a node `x`, we are looking for its "nearest ancestor" (call it `r`) that satisfies some criterion. With `is_the_Block_closest_containing`, the criterion is very simple:

```
     r is_a_Block
```

However, for `is_the_defining_occurrence_of`, the criterion is more complicated:

> there exists a `dp` such that:
> `dp is_effective_over r` and the spelling of `dp` equals the spelling of `x`.

The latter criterion involves the spelling of `x`, the node at which the rootward search begins. In the case of `is_the_Block_closest_containing`, this node is "left behind" as successively deeper invocations of the recursive rule travel rootward. However, with `is_the_defining_occurrence_of`, it is necessary for deep invocations to "remember" `x` or its spelling. Therefore, an additional argument is necessary: we need a ternary relation to do the work for the binary relation `is_the_defining_occurrence_of`. As it happens, the ternary relation also has a useful interpretation: we can say that

```
     d defines_the_spelling $s if_it_occurs_at x
```

when `d` would be the defining occurrence of a lexeme with spelling `$s` if that lexeme occurred in the context of node `x`.[7] Here, the name of the variable `$s` begins with a `$` to indicate that it binds to a character-sequence, not to a node. It is this variable that will carry along the spelling of `x` in `is_the_defining_occurrence_of`. The rule that defines the ternary relation is:

[7] This interpretation of the relation ignores the fact that `x` could be in a context that does not allow a simple lexeme to appear. One could probably find an interpretation that corresponds more closely to the syntax of the language, but it would certainly be more complex and probably less intuitive. Merks [Mer88] has a routine (`DefiningOccurrenceAt`) with much the same interpretation, so he is presumably not disturbed by its generality.

```
RELATION DEF
  d defines_the_spelling $s if_it_occurs_at x:
    FUNCTIONAL ON {$s,x}, {d,x}
:-
  IF (dp is_effective_over x, $s is_the_spelling_of dp) THEN
    d is dp
  ELSE
    p is_the_parent_of x,
    d defines_the_spelling $s if_it_occurs_at p
  END
.
```

Here we can see, in the condition of the `IF-THEN-ELSE` construct, the success criterion mentioned earlier.

Specifying `is_the_defining_occurrence_of` is now easy:

```
RELATION DEF d is_the_defining_occurrence_of x: FUNCTIONAL ON {x} :-
  $s is_the_spelling_of x,
  d defines_the_spelling $s if_it_occurs_at x
.
```

The observant reader may have noticed that the `is_the_spelling_of` relation has not been defined, although the word "spelling" has been used to suggest the sequence of characters in a lexeme. In fact, the spelling of a lexeme is a slightly more involved concept.

Section 6.1.1 of the Pascal Standard states: "The representation of any letter (upper-case or lower-case, differences of font, etc.) occurring anywhere outside a character-string ... shall be insignificant in that occurrence to the meaning of the program." For example, an identifier declared as "`foo`" can be the defining occurrence of identifiers with the character-sequences "`foo`", "`Foo`", "`FOO`", and so on. In addition, Section 6.1.6 of the Standard states: "Labels ... shall be distinguished by their apparent integral values." Thus, a label declared as "`099`" could be the defining occurrence of a label with the character-sequence "`099`", "`99`", "`0099`", and so on. The *spelling* of an identifier or label is a character-sequence that eliminates these differences of case and format. The way in which it does this is not prescribed by the Pascal Standard, so I have chosen the following definition:

```
LEXEME Identifier IS #letter { #letter | #digit }
LEXEME Label IS #digit { #digit }

(*
The spelling of an Identifier is
 the character-sequence of the Identifier,
 with any upper-case letters converted to lower-case.
The spelling of a Label is
 the character-sequence of the Label,
 with leading zeroes removed.
*)
```

```
    RELATION DEF $s is_the_spelling_of x: FUNCTIONAL ON {x} :-
      x is_a_Identifier,
      $a is_the_character_sequence_of x,
      $s is_the_lower_case_translation_of $a
    OR
      x is_a_Label,
      $z is_the_character_sequence_of x,
      $s is_the_unzeroed_translation_of $z
    .
```

The relations `is_the_lower_case_translation_of` and `is_the_unzeroed_translation_of` are treated as primitive relations because NURN does not include any notation to create or examine character-sequences, which would be necessary to define these relations. Such notation was not included in NURN because the lexical aspects are not being emphasized in this thesis. These relations and two other ad hoc and somewhat Pascal-specific relations:

`$c is_the_charseq_for_the_char_value_for_ordinal #ord`, and
`#n is_the_number_of_string_elements_in $s`

are included in my NURN grammar for Pascal.

3.3.5 Relation rules for `is_a_conflicting_declaration_point` *and* `is_undefined`

We turn finally to the violation relations `is_a_conflicting_declaration_point` and `is_undefined`.

Recall that `dp_1 is_a_conflicting_declaration_point` if `dp_1` is a declaration-point that duplicates the spelling of another declaration-point (`dp_2`) effective over the same region. This violation relation is easily expressed in the following definition:

```
    (* Two distinct declaration-points effective over the same region
       must not have the same spelling.
    *)
    RELATION DEF
      VIOLATION dp_1 is_a_conflicting_declaration_point :-
      $s is_the_spelling_of dp_1, dp_1 is_effective_over r,
      $s is_the_spelling_of dp_2, dp_2 is_effective_over r,
      NOT (dp_1 is dp_2)
    .
```

Recall that `x is_undefined` if `x` is a label or identifier without a defining occurrence. This is also easily expressed:

```
    (* A Label or Identifier must have a defining occurrence. *)
    RELATION DEF VIOLATION x is_undefined :-
      (x is_a_Label OR x is_a_Identifier),
      NOT (dp is_the_defining_occurrence_of x)
    .
```

One (possibly unintuitive) aspect of the semantics of NURN is that the logical interpretation of a negation includes an existential quantification of any variables in its body that are not referenced outside it. In the definition above, dp is such a variable. Thus, the definition could be read: "x is_undefined if x is a Label or Identifier, and there does not exist a node dp such that dp is_the_defining_occurrence_of x."

Now in fact, the Pascal Standard does not require *every* label or identifier to have a defining occurrence, just those within the block of a program.

```
(* A Label or Identifier within the Block of a Program
   must have a defining occurrence.
*)
RELATION DEF VIOLATION x is_undefined :-
  (x is_a_Label OR x is_a_Identifier),
  b is_the_Block_of p, p is_a_Program,
  b contains x,
  NOT (dp is_the_defining_occurrence_of x)
.
```

The Standard includes the restriction to the block of a program in order to relieve the name and parameters of a program from having defining occurrences[8], and the restriction has this effect because the Standard is not concerned with nodes within fragments. However, in a NURN grammar, which applies equally to such nodes, there is an added effect: any node not within a Program is also relieved from having a defining occurrence. For example, if the assignment–statement c:='j' occurs as a fragment, the identifier c is not required to have a defining occurrence, i.e., it is not an instance of the is_undefined relation. If this is not the desired effect, the name and parameters of a program can be excluded explicitly:

```
(* A Label or Identifier
   (other than the Name of a Program or
    an Identifier in the Parameters of a Program)
   must have a defining occurrence.
*)
RELATION DEF VIOLATION x is_undefined :-
  (x is_a_Label OR x is_a_Identifier),
  NOT
  (
    x is_the_Name_of p,
    p is_a_Program
  OR
    x is_a_Identifier_in idl,
    idl is_the_Parameters_of p,
    p is_a_Program
  ),
  NOT (dp is_the_defining_occurrence_of x)
.
```

[8] The name of a program has no significance and the parameters are dealt with specially.

3.3.6 Make Functions

In Subsection 3.3.1, I described how the GRAMPS-style rules of a NURN grammar implicitly define context-free relations. In addition, they also implicitly define context-free functions[9] called *Make functions*, corresponding to GRAMPS-style constructors[10]. For instance, the function `make_ConstantDefinition` specifies the construction of a `ConstantDefinition` node from its two arguments. A Make function can only be used in a `Make`, a NURN primary with the appearance of an assignment. Moreover, a `Make` can only appear in the definition of a maker relation. These restrictions are imposed because Make functions are inherently non-logical constructs, and it is desirable to isolate, as much as possible, their side-effects.

In general, Make functions are used to creates nodes to represent context-dependent entities for which there is no suitable representative in the program. Most programming languages have a set of such entities generally referred to as *predefined* entities, which exist independently from any program. These are denoted by predefined identifiers, whose scoping must be defined somehow. In explaining Pascal scoping rules in Section 3.1, and in defining them in Section 3.3, I avoided this matter, but I can now deal with it using Make functions.

In Pascal, the rule is that predefined identifiers "shall be used as if" their declaration-points are effective over a region enclosing the program. This is the only rule concerning the scoping of these predefined identifiers; otherwise, they behave like any other identifier.[11] Therefore, it would be convenient if they could be formally handled like any other identifier. However, in contrast to all the previous examples in this section, the declaration-points of predefined identifiers do not exist, nor does the region that they are supposedly effective over. That is, they do not and cannot exist *in any program.*

In previous work with ad hoc context-dependent SMSs, predefined identifiers have been handled in different ways. In [Cam87], the routine `DefiningOccurrence` returns `NIL` if no defining occurrence is found within the program. Thus, predefined identifiers and undefined identifiers cause the same response: whenever `DefiningOccurrence` returns `NIL`, one must check if the argument's spelling is that of a predefined identifier; and even if it is, one still does not have a node representing the defining

[9] The word "function" is used in several ways in this thesis. Here, "function" refers to a particular construct in a NURN grammar (see Appendix 1). These should be distinguished from functions defined in the host-language, such as the recognizers, selectors, and constructors seen in Section 2.2. Moreover, the target-language may have constructs called "functions", particularly if it is a programming language.

[10] Also, there is a single generic Make function named `tail`, which takes a repetition node as an argument and returns a copy of it with the first element removed.

[11] In particular, they can be redeclared for a nested region, as opposed to languages in which they are "reserved", thereby preventing a programmer from redefining them.

occurrence, merely the information that it is a predefined identifier. One could use an analogous technique in writing a NURN grammar, but this would hardly be a uniform treatment of identifiers.

On the other hand, in [Mer88], a special module is constructed to house the declaration-points of predefined identifiers. These nodes can then be returned by the routine `DefiningOccurrence`. The latter technique is more attractive because it distinguishes between predefined and undefined identifiers, provides a more uniform approach to the concept of defining occurrence, and corresponds more closely with informal language description. Using an analogous technique in a NURN grammar requires the ability to specify the creation of syntactic structures, and Make functions provide this ability.

We can use Make functions to build a special `Block` in which to house the declaration-points of the predefined identifiers.

```
RELATION DEF MAKER b is_the_predefined_Block :-
  b := make_Block ( ...... )
.
```

The actual expression used to create the predefined-block is large and difficult to grasp (see Appendix 2 under Section 6.2.2.10); rather than show it here, it is more helpful to give a textual representation of the predefined-block:

```
const maxint = 32767 ;

type
  boolean = (FALSE, TRUE) ;
  integer = TheIntegerType ;
  real    = TheRealType ;
  char    = TheCharType ;
  text    = file of char ;

procedure rewrite ( f : aFile ) ; PREDEFINED ;
procedure put     ( f : aFile ) ; PREDEFINED ;
procedure reset   ( f : aFile ) ; PREDEFINED ;
procedure get     ( f : aFile ) ; PREDEFINED ;
procedure read                  ; PREDEFINED ;
procedure readln                ; PREDEFINED ;
procedure write                 ; PREDEFINED ;
procedure writeln               ; PREDEFINED ;
procedure page                  ; PREDEFINED ;

procedure new     ; PREDEFINED ;
procedure dispose ; PREDEFINED ;

procedure pack
  (a: UnpackedArray; i: Ordinal; z: PackedArray) ; PREDEFINED;
procedure unpack
  (z: PackedArray; a: UnpackedArray; i: Ordinal) ; PREDEFINED;

function abs    ( x: Numeric ) : Numeric ; PREDEFINED ;
function sqr    ( x: Numeric ) : Numeric ; PREDEFINED ;
function sin    ( x: real ) : real ; PREDEFINED ;
function cos    ( x: real ) : real ; PREDEFINED ;
function exp    ( x: real ) : real ; PREDEFINED ;
function ln     ( x: real ) : real ; PREDEFINED ;
function sqrt   ( x: real ) : real ; PREDEFINED ;
function arctan ( x: real ) : real ; PREDEFINED ;

function trunc  ( x: real ) : integer ; PREDEFINED ;
function round  ( x: real ) : integer ; PREDEFINED ;

function ord  ( x: Ordinal ) : integer ; PREDEFINED ;
function chr  ( x: integer ) : char    ; PREDEFINED ;
function succ ( x: Ordinal ) : Ordinal ; PREDEFINED ;
function pred ( x: Ordinal ) : Ordinal ; PREDEFINED ;

function odd  ( x: integer ) : boolean ; PREDEFINED ;
function eof                 : boolean ; PREDEFINED ;
function eoln                : boolean ; PREDEFINED ;

begin end
```

Observe that syntactically, this is a `Block` like any other. The advantage of this is that many of the rules that apply to "normal" blocks also apply to this one. For example, the rules already given in Section 3.3 for the `is_effective_over` relation say that because of the contexts in which the predefined identifiers appear in the predefined-block, each is a declaration-point effective over this block. We can use this fact to revise the scoping rule for finding the defining occurrence of a lexeme: if the root of the syntax-tree is reached without finding a region having a declaration-point effective over it whose spelling matches that of the lexeme, continue the search at the predefined-block. We express this by revising the rule for the relation `d defines_the_spelling $s if_it_occurs_at x` as follows:

```
RELATION DEF d defines_the_spelling $s if_it_occurs_at x :-
  IF (dp is_effective_over x, $s is_the_spelling_of dp) THEN
    d is dp
  ELSE
    IF (p is_the_parent_of x) THEN
      cont is p
    ELSE
      cont is_the_predefined_Block,
      NOT ( x is_the_predefined_Block )
    END,
    d defines_the_spelling $s if_it_occurs_at cont
  END
.
```

If the parent p of x exists, the place to continue the search (cont) is p, and the rule recurses as before. If the parent does not exist, x is the root of a syntax-tree: the place to continue the search is with the predefined-block, unless we are already there, in which case the rule fails.

In the predefined-block, many of the declarations use identifiers that are not defined, e.g., `TheMaximumIntegerValue`, `TheIntegerType`, and `SomeFileType`. This is not surprising: predefined identifiers usually have definitions which cannot be expressed within the language. (Here, `boolean` is a notable exception.) Since practically every program gives rise to the predefined-block, we definitely do not want these undefined identifiers to be instances of the violation relation `is_undefined`. The first alternative definition for `is_undefined` in Subsection 3.3.5 will give us this assurance, since the nodes in the predefined-block are not contained by the block of a program, and are thus relieved of having a defining occurrence.

The operation of Make functions is determined by the maker relations in which they occur. Consider all parameters of a maker relation, and exclude those (the *made parameters*) that appear on the left-hand side of a `Make` within the definition of the relation. The remaining parameters are the relation's *unmade parameters*. For each distinct set of entities bound to the unmade parameters of a maker relation, there is at most one instance of the relation consistent with those bindings, and if it exists, the entities bound to the made parameters do not appear in any other instance of the relation. For example, in the `is_the_predefined_Block`, there is one made parameter and no unmade parameters. Thus, there can be only one distinct set of bindings for the unmade parameters (the empty set), and therefore, only one instance of the relation. That is, there is only one predefined-block, which is the desired effect.

3.3.7 The `IS_THE_EXPANSION_OF` *relation*

Informal descriptions of context-dependent languages sometimes employ "definition-by-equivalence". Using this technique, a complex construct of the language is not defined per se; rather, a transformation is given whereby any instance of this construct is expanded into an equivalent structure of simpler, more-easily-defined constructs.

For example, consider Pascal's with-statement. Here is its context-free syntax:

```
CONSTRUCT WithStatement IS
  "with" <:VariableAccessList> "do" <Body:Statement>

LIST VariableAccessList OF VariableAccess SEPARATED_BY _ ","
```

The context-dependent syntax of with-statements is defined only for with-statements having a single variable-access in their variable-access-list. This was given in Subsection 3.3.2. The Pascal Standard then says that the statement

```
with v1, v2, ..., vn do
  s
```

is equivalent to

```
with v1 do
  with v2 do
    ...
      with vn do
        s
```

This is not the only way to define the syntax of with-statements, but it is certainly the simplest. Without the use of definition-by-equivalence, the description would have to either start with a less clear context-free syntax for with-statements or else introduce more complicated scoping rules.

To accomodate this descriptive technique, NURN has a predeclared relation:

```
RELATION DECL MAKER ex IS_THE_EXPANSION_OF x: FUNCTIONAL ON {x} .
(* ex is an equivalent expansion of x into simpler constructs *)
```

For example, the above expansion for with-statements can be expressed as follows:

```
RELATION SUBDEF wex IS_THE_EXPANSION_OF w :-
  w is_a_WithStatement,
  st is_the_Body_of w,
  val is_the_VariableAccessList_of w,
  #len is_the_length_of val,
  #len IS_GREATER_THAN 1,
  va is_the 1 th_VariableAccess_in val,
  wex :=
    make_WithStatement
      ( make_VariableAccessList ( va ),
        make_WithStatement ( tail(val), st )
      )
.
```

Conceptually, any node for which an expansion can be found is replaced by that expansion. The replacement may also be subject to expansion, as in the example above. The rest of the NURN grammar can then be written as if these replacements have been made.[12]

There are two other cases in the Pascal Standard where the use of definition-by-equivalence allows a simple definition of context-dependent syntax. The array-type with a list of index-types

```
ARRAY [i1, i2, ..., in] OF b
```

is equivalent to

```
ARRAY [i1] OF ARRAY [i2] OF ... ARRAY [in] OF b
```

Similarly, the index-variable with a list of index-expressions

[12] Whether (and how) the replacements are performed in a resulting SMS is determined by the SMS derivation scheme used. The scheme-designer must decide whether SMSs derived according to that scheme will present the SMS-users with the idea that they are manipulating objects of the full language or the simpler language. If the SMS manipulates the full language, then the designer must decide how the SMS will handle queries involving non-simple objects. If the SMS manipulates the simpler language, then the designer must decide how the SMS will provide the conversion from the full language, and how it will handle attempts to construct non-simple objects.

```
    v[e1, e2, ..., en]
```
is equivalent to
```
    v[e1][e2]...[en]
```
In both cases, it is much easier to define the context-dependent syntax solely in terms of the second, expanded form; defining it in terms of the first form would require using a much less clear context-free syntax.

3.3.8 The `IS_A_VIOLATION` *relation*

NURN includes one other predeclared relation:
```
    RELATION DECL x IS_A_VIOLATION .
```
This relation is implicitly subdefined by each violation relation in the NURN grammar. For example, if the two violation relations seen so far were the only ones in a NURN grammar, the `IS_A_VIOLATION` relation would have the following (implicit) definition:
```
    RELATION DEF x IS_A_VIOLATION :-
      x is_undefined
    OR
      x is_a_conflicting_declaration_point
    .
```
The `IS_A_VIOLATION` relation would not normally be used in a NURN grammar, but it is useful in the resulting NURN SMS.

3.4 Pascal Types

All the features of NURN have been shown, but it is instructive to reinforce the presentation and emphasize NURN's generality with further examples. In this section, we will see how NURN can be used to describe some of the trickier aspects of the Pascal type system.

3.4.1 Multi-type literals: `nil` *and* `[]`

Section 6.4.4 of the Pascal Standard states: "The token `nil` shall denote the nil-value in all pointer-types. ... The token `nil` does not have a single type, but assumes a suitable pointer-type to satisfy the assignment-compatibility rules, or the compatibility rules for operators, if possible." For example, consider the following block:
```
    VAR
      p : @integer ;
      q : @real ;
    BEGIN
      IF p=nil THEN q:=nil
    END
```

The compatibility rules for the operator = require that p and the first occurrence of nil have the same type (@integer), and the assignment-compatibility rules require that q and the second occurrence of nil have the same type (@real). Thus, the two occurrences of nil have different types. Since most typed entities in Pascal have a single consistent type, this case is somewhat anomalous.

Rather than translating the Standard's description into NURN, we can use a different but equivalent approach whereby nil has a single consistent type. The idea is to invent a unique type (called the *null-type*, say) to be the type of all occurrences of nil, and modify the definition of compatibility to ensure that it is compatible with all pointer-types.[13]

The null-type is like a predefined type, in that it should exist for all programs to reference. However, it cannot be represented in the predefined-block because this would introduce an identifier to denote it, which would then become a predefined identifier, which would constitute an extension to Pascal. Instead, in the same way that the whole predefined-block was created, a single orphan node is created to represent the null-type. Since this node is used only in that capacity, and since the null-type has no structure (no subtypes), it could just be an empty node. However, since there may be occasion to display it concretely, an informative identifier (TheNullType) is used instead.

```
RELATION DEF MAKER t is_the_null_type :-
  t := make_Identifier("TheNullType")
.
```

Note that this does not introduce TheNullType as a predefined identifier, since it is not involved in any scoping rules.

The set-constructor [] is a similar case to nil. Section 6.7.1 of the Pascal Standard states: "The set-constructor [] shall denote that value in every set-type that contains no members." A satisfactory approach is to invent a type (the *empty-set-type*, say), represent it with a specially-created orphan identifier (TheEmptySetType), and ensure that it is compatible with any set-type.

3.4.2 Literal string-types

In Pascal, the type of a character-string with more than one character (e.g., "hello world") is an array-type exactly big enough to hold it (e.g., PACKED ARRAY [1..12] OF char). This is an *undeclared* type, since it does not appear in the program with the character-string. A clear and workable approach is to represent each such undeclared array-type with a created ArrayTypeDefiner, so that it can be treated like any other array-type. The only identifier in the created object is the identifier char, and it will scope as desired (i.e., it will denote the predefined char-type), because when the defines_the_spelling relation (as redefined in Subsection 3.3.6) reaches the root of the

---

[13] Modula-3 [Car*89] has this idea as part of a more uniform type system.

ArrayTypeDefiner, it will search the predefined-block for the defining occurrence of char. The relation to create these undeclared types is defined by the following rule:

```
RELATION DEF
 MAKER t is_the_string_type_corresponding_to cs: FUNCTIONAL ON {t}, {cs}
:-
  cs is_a_CharacterString,
  $s is_the_character_sequence_of cs,
  #n is_the_number_of_string_elements_in $s,
  $z IS_THE_BASE_TEN_REP_OF #n,
  t :=
    make_PackedStructuredType
      ( make_ArrayTypeDefiner
          ( make_TypeDenoterList
              ( make_SubrangeTypeDefiner
                  ( make_UnsignedIntegerLiteral("1"),
                    make_UnsignedIntegerLiteral($z)
                  )
              ),
            make_Identifier("char")
          )
      )
.
```

3.4.3 Canonical set-types

Section 6.4.3.4 of the Pascal Standard states that for each ordinal-type (other than a subrange-type), there exist (implicitly) two *canonical set-types*, packed and unpacked. These types are not declared and cannot be denoted. The type of a set-constructor or a dyadic expression with set operands is a canonical set-type. For example, consider the block:

```
TYPE
  Suit = (spade,club,heart,diamond) ;
  BlackSuits = PACKED SET OF spade .. club ;
  RedSuits = PACKED SET OF heart .. diamond ;
VAR
  s : Suit ;
  b : BlackSuits ;
  r : RedSuits ;
  is_in_b_or_r : ARRAY [Suit] OF boolean ;
BEGIN
  b := [spade] ;
  r := [heart,diamond] ;
  FOR s := spade TO diamond DO
    is_in_b_or_r[s] := ( s IN b+r ) ;
END
```

The type of the set-constructors [spade] and [heart,diamond], and of the expression b+r, is the packed canonical set-of-(spade,club,heart,diamond) type. There are two independent problem areas with canonical set-types: (1) creating them; and (2) determining from context whether the type of a set-constructor is packed or unpacked.

Canonical set-types are similar in many respects to the array-types of character-strings presented in Subsection 3.4.2. However, the only difference between the created array-types is the size of their index-types. Here, canonical set-types have a more significant difference: their base-type. A canonical set-type is represented with a `SetTypeDefiner` created by

```
   make_SetTypeDefiner ( bt )
```

where the `BaseTypeDenoter` argument `bt` is bound to the node representing the ordinal-type giving rise to the canonical set-type. This `SetTypeDefiner` is then related to `bt`, so that it is "accessible" from the program's syntax-tree.

```
RELATION DEF
  MAKER st is_the_packed_canonical_set_type_associated_with bt:
   FUNCTIONAL ON {st}, {bt}
:-
  bt is_an_ordinal_type,
  NOT (bt is_a_SubrangeTypeDefiner),
  st := make_PackedStructuredType ( make_SetTypeDefiner ( bt ) )
.

RELATION DEF
  MAKER st is_the_unpacked_canonical_set_type_associated_with bt:
   FUNCTIONAL ON {st}, {bt}
:-
  bt is_an_ordinal_type,
  NOT (bt is_a_SubrangeTypeDefiner),
  st := make_SetTypeDefiner ( bt )
.

RELATION DEF st is_a_canonical_set_type :-
  st is_the_packed_canonical_set_type_associated_with bt
OR
  st is_the_unpacked_canonical_set_type_associated_with bt
.
```

There are problems with this representation: since two nodes cannot share a child node because this would violate the "tree-ness" of the context-free node-structure, and since `bt`, the node representing the ordinal-type giving rise to the canonical set-type, is presumably a child of some node within a program's syntax-tree, the Make function `make_SetTypeDefiner(bt)` creates a `SetTypeDefiner` whose `BaseTypeDenoter` component is only a *copy* of `bt`. To see how this creates problems, we will assume for the moment that the original ordinal-type (and thus, the copy) is an `EnumeratedTypeDefiner`. Now two things about an `EnumeratedTypeDefiner` are that (1) it represents an ordinal-type, and (2) it denotes the type that it represents. Thus:

1. The copy also represents an ordinal-type, and will give rise to another pair of canonical set-types, each with a copy of the copy, etc., in an exponential explosion of canonical set-types.
2. The copy also denotes the type that it represents. Now a `SetTypeDefiner` represents a set-type,

and the base-type of a set-type is the type denoted by the `BaseTypeDenoter` of the `SetTypeDefiner`, so the base-type of the canonical set-type will be the type represented by the copy. However, the base-type *should* be the type represented by the original.

If, on the other hand, the original ordinal-type (and thus, the copy) is a predefined ordinal-type represented by an identifier (`TheIntegerType` or `TheCharType`), problems will occur because a copy of such an identifier does not denote anything; (the original only represents a predefined type by special dispensation).

The solution to all these problems is to deny the type-hood of the `BaseTypeDenoter` of a canonical set-type. That is, it does not represent a type; instead, it is made to denote the type represented by the original of which it is a copy. An alternative solution would use orphan identifiers to represent canonical set-types. This representation would require very little special handling, but a concrete display of a canonical set-type would be very uninformative.

Turning now to the second of the two independent problem areas, Section 6.7.1 of the Pascal Standard states that a non-empty set-constructor has an unpacked canonical set-type, or if the context requires, a packed canonical set-type. We have already seen examples (the empty set-constructor `[]` and `nil` in Subsection 3.4.1) where the Standard's description says that context determines type. There, it was deemed simpler to invent a type for the construct than to duplicate the Standard's description. A similar approach could be taken here: one could invent a third class of canonical set-types (*literal* canonical set-types, say) which are neither packed nor unpacked, but compatible with either. This would complicate the compatibility rule somewhat, and the expression-typing rule even more so, but not to an unmanageable extent. However, it is instructive to examine how NURN can express context-determined typing.

Here is an approximation to the rule wherein the type of a set-constructor is determined:

```
RELATION SUBDEF t is_the_type_of sc :-
  sc is_a_SetConstructor,
  rt is_the_type_of_all_member_designators_in sc,
  IF (sc is_in_a_context_requiring_a_packed_set_type) THEN
    t is_the_packed_canonical_set_type_associated_with rt
  ELSE
    t is_the_unpacked_canonical_set_type_associated_with rt
  END
.
```

Thus, it is now a question of how to define this relation:

```
RELATION DECL x is_in_a_context_requiring_a_packed_set_type .
```

The Standard does not explicitly state how a context can require a packed set-type, but examination of the Standard indicates that the constraints of compatibility and assignment-compatibility are operative here.

Assignment-compatibility is required between the expression and variable-access of an assignment-statement, and between an actual value parameter and its corresponding formal parameter. For example, the assignment-statement `b:=[spade]` appears in the block above. Since `b` has a packed set-type, `[spade]` must assume a packed set-type. We can capture these cases with the following rule:

```
RELATION SUBDEF x is_in_a_context_requiring_a_packed_set_type :-
  asmt is_a_AssignmentStatement,
  x is_the_Expression_of asmt,
  va is_the_VariableAccess_of asmt,
  tv is_the_type_of va,
  tv is_a_set_type,
  tv is_designated_packed
OR
  x is_a_actual_value_parameter,
  fp is_the_formal_for x,
  tfp is_the_type_of fp,
  tfp is_a_set_type,
  tfp is_designated_packed
.
```

Compatibility is required between the two operands of a dyadic expression. For example, if the expression `b+[club]` had occurred in the above block, the set-constructor `[club]` would have to assume a packed set-type in order to be compatible with `b`. Thus:

```
RELATION SUBDEF x is_in_a_context_requiring_a_packed_set_type :-
  expr is_a_dyadic_expression,
  (
    y is_the_LeftOperand_of expr, x is_the_RightOperand_of expr
  OR
    x is_the_LeftOperand_of expr, y is_the_RightOperand_of expr
  ),
  ty is_the_type_of y,
  ty is_a_set_type,
  ty is_designated_packed
.
```

To complete the definition of the relation, we need to recognize that if a dyadic expression is in a context requiring a packed set-type, so are each of its operands. Similarly, if a bracketted-expression is in such a context, so is its expression. For example, if `b+([club]+[spade])` appeared in the block above, both of these cases would be used to assert that the two set-constructors must assume packed set-types.

```
RELATION SUBDEF x is_in_a_context_requiring_a_packed_set_type :-
  expr is_a_dyadic_expression,
  expr is_in_a_context_requiring_packed,
  (x is_the_LeftOperand_of expr OR x is_the_RightOperand_of expr)
OR
  expr is_a_BrackettedExpression,
  expr is_in_a_context_requiring_packed,
  x is_the_Expression_of expr
.
```

3.4.4 The "treated as" rules

According to Section 6.7.1 of the Pascal Standard, an expression whose type is a subrange-type can be "treated as" being of the host-type, and an expression whose type is a (packed or unpacked) set-type can be "treated as" being of the corresponding (packed or unpacked) canonical set-type. These rules suggest that it might be necessary for an expression to have two types, but in fact, this is not the case. The "treated as" rules are only used to simplify the statement of the expression-typing rules. For example, with the "treated as" rules, the Standard can state that the operands of set union must have the same canonical set-type and the result will have that type. Without them, one must say something like the operands of set union must have set-types that correspond to the same canonical set-type, and the result is the canonical set-type corresponding to the type of either of the operands.[14]

This revolves around the concept of the most general type that is compatible with a given type, which I will represent here as `is_the_MG_type_for`, a relation between types. Now imagine that you want to say (in NURN) that an expression has the type integer or some subrange of integer. (Never mind the context of this NURN phrase.) You could say it as follows:

```
expr is_a_Expression,
te is_the_type_of expr,
t is_the_MG_type_for te,
t is_the_integer_type
```

If you group the second and third primaries, you get the "treated as" rule:

```
RELATION DEF expr can_be_treated_as_having_type t :-
  te is_the_type_of expr,
  t is_the_MG_type_for te
.
```

whereupon the NURN phrase can be written:

```
expr is_a_Expression,
expr can_be_treated_as_having_type t,
t is_the_integer_type
```

On the other hand, if you group the third and fourth primaries, you get the approach I took:

[14] Oddly enough, some of this alternative phrasing does appear in the Standard, as if the authors had forgotten that they had the "treated as" rules to fall back on. In fact, most of the Standard's definition of compatibility is redundant, given the "treated as" rules.

```
    RELATION DEF te is_an_integer_type :-
     t is_the_MG_type_for te,
     t is_the_integer_type
    .
```

whereupon the NURN phrase can be written:

```
      expr is_a_Expression,
      te is_the_type_of expr,
      te is_an_integer_type
```

3.4.5 Summary

As we have seen, NURN handles the type system of Pascal quite well, even the anomalous features. Also, NURN rules can be written using phrasing close to what one would write naturally in a language specification.

CHAPTER 4
SYNTACTIC MANIPULATION SYSTEMS BASED ON NURN GRAMMARS

A NURN SMS is a context-dependent SMS based on a NURN grammar. In Section 4.1, I introduce terms to describe the operations performed by a NURN SMS, and in Section 4.2, I describe how a NURN SMS operates. In Section 4.3, I argue that a context-dependent SMS should leave the construction of syntactic objects to context-free routines. In Section 4.4, I suggest various schemes for deriving the routines of a NURN SMS from a NURN grammar. In Section 4.5, I describe how the Ginger system automatically generates a NURN SMS.

4.1 Query Forms, Binding Patterns, Queries, and Investigators

A relation defined in a NURN grammar can be used to ask questions of different forms. For instance, given the relation `d is_effective_over r` discussed in Section 3.2, we can imagine asking:

1. Is declaration-point `d` effective over region `r`?
2. What declaration-points are effective over region `r`?
3. What regions is declaration-point `d` effective over?
4. What are all pairs of declaration-point and effective region?

Specifically, a *query form* is determined by a particular relation and for each of its parameters, an indication of whether that parameter is bound or not. Thus, for an n-ary relation, there are 2^n corresponding query forms. We can denote a query form by writing `B` for each bound parameter and `U` for each unbound parameter. Thus, the examples above correspond to the four query forms:

1. `B is_effective_over B`
2. `U is_effective_over B`
3. `B is_effective_over U`
4. `U is_effective_over U`

We define the ***binding pattern*** of a query form as its `U`s and `B`s taken in left-to-right order. For example, `U is_effective_over B` and `U is_the_defining_occurrence_of B` are distinct query forms, but they have the same binding pattern, `UB`.

A *query* is an instance of a query form for some relation, with an actual entity supplied for each bound parameter. Answering a query consists of finding every set of entities (if any) that can be bound to the query's unbound parameters such that they, along with the bindings supplied in the query, satisfy the relation. Equivalently, answering a query consists of finding each instance of the relation that matches the bindings supplied in the query.

An *investigator* for a query form is an algorithm that answers all queries that are instances of the query form. When Ginger generates investigators, it expresses them in a language closely related to NURN.[1] These investigators are interpreted by the SMS to answer queries with reasonable efficiency.

The *extension* of a relation is the set of instances of that relation; i.e., the set of entity–tuples that satisfy the definition of the relation. Calculating the extension of a relation is equivalent to answering the query for that relation with no parameters bound. We use the term "all–unbound" to designate the query and its investigator. For example, Subsections 3.2.1 through 3.2.4 enumerated the extensions of the relations involved in Pascal's scoping rules.

4.2 Overview of a NURN SMS

From a user's point of view, a NURN SMS consists of a set of host–language routines to construct and query syntactic objects of the target–language, as specified by a NURN grammar. It is useful to distinguish constructive and investigative routines. Any routine that changes syntactic objects is a *constructive* routine. Such routines are used for constructing, deleting, and editing syntactic objects. The constructive routines of a NURN SMS are just those that would be provided by a GRAMPS–style SMS for the context–free super–language.[2] On the other hand, an *investigative* routine is used to answer queries; it can be considered either context–free or context–dependent depending on whether the relation of the queries it answers is context–free (primitive or derived from GRAMPS–style rules) or context–dependent (defined by relation rules). The investigative routines of a NURN SMS subsume the recognizers and selectors that would be provided by a GRAMPS–style SMS.

Besides the routines seen by the user, the SMS includes investigators and an interpreter for executing these investigators. Since the interpreter interprets a NURN–like language, it is independent of the target–language. When an investigative routine is called by a user program, the routine finds the appropriate investigator in a table of investigators, prepares the arguments of the investigator, and invokes the interpreter with the investigator and arguments.

Generating a NURN SMS consists of three relatively independent tasks: generating the investigators, generating the investigative routines, and generating the constructive routines. The first two tasks are considered in Section 4.5 and Section 4.4, respectively. For the third task, see [Ter87].

Ideally, a NURN SMS would monitor the creation, deletion and editing of target–language objects and always keep the extensions of the NURN relations up–to–date (or at least appear to), using efficient

[1] A grammar for investigators appears in Appendix 1, along with a grammar for NURN.

[2] Section 4.3 will explain why there are no context–dependent constructors.

incremental algorithms when necessary. However, deriving these algorithms appears to be a difficult task, and the SMSs generated by Ginger do not meet the ideal yet. Instead, they must be used in a fairly static fashion:

1. Through calls to constructive routines, the user constructs one or more syntactic objects. (For example, the user might call a parsing routine to parse a text-file as a program.)
2. The user calls the routine `InstallSubject` to "install" each object, making it known to the "investigative" portion of the SMS.
3. The user calls the routine `FindAllExtensions`, which first forgets any previously gathered information, then calculates the extension of every relation with respect to the installed objects.
4. Thereafter, when the user calls an investigative routine, it merely finds those tuples in the extension of the appropriate relation that match any bound arguments in the query.[3]

At first, it might appear that calculating the *full* extension of *all* relations requires excessive effort, much more than just calculating and remembering instances of relations as needed to answer particular queries. For my purposes, this is not the case, since I always want a full syntax-check of the installed object(s), i.e., all instances of the relation `IS_A_VIOLATION`. Calculating the extension of this relation in the piecemeal manner described would produce almost the full extension of each relation, and would probably take longer than purposely calculating all extensions. Moreover, the full-extension method is far easier to program, because recursive relations are much more formidable when using the piecemeal method.

`FindAllExtensions` has two aspects:

1. Interpretation: Given the all-unbound investigator for a relation and the extensions of all the relations that it references, calculate the extension of the relation according to the semantics of NURN rules.
2. Control: Ensure that an investigator is processed after those that it relies on.

The characterization of Control appears paradoxical for recursive investigators, each of which relies (directly or indirectly) on itself. Control is especially important for such investigators. A *recursive cluster* is a maximal set of recursive investigators such that each investigator in the cluster relies (directly or indirectly) on every other investigator in the cluster. For instance, the investigator for `U is_the_Block_closest_containing B` is in a cluster by itself; `U is_effective_over U` is in a cluster with `U is_effective_over B`, `U is_the_type_of B`, and several other investigators. Control processes each recursive cluster as a group. Initially, each of the recursive relations in a cluster is given an empty extension as an approximation of its true extension. Then, assuming these approximations, better approximations to the extensions of the relations are calculated. These new approximations are

[3] Actually, that is an oversimplification. For some relations (those with huge extensions and/or simple definitions) it is generally faster to calculate the answer to a particular query than it would be to find it in the relation's extension. For these relations, the extension is never calculated.

used in the next iteration, and so on. This process continues until the approximations stop changing, at which point the true extensions have been found[4]. This is a common approach for dealing with sets of recursive definitions. For example, Aho, Sethi, and Ullman [ASU86] use it to solve the equations of global data-flow analysis.

4.3 Context-Dependent Constructors Considered Harmful

As outlined in Section 4.2, a NURN SMS comprises both constructive and investigative routines, but the constructive routines are just those that would be provided by a GRAMPS-style SMS. That is, with respect to the NURN grammar for the target-language, the constructive routines are based solely on the context-free GRAMPS-style rules and ignore the context-dependent relation rules. The reasons for not providing context-dependent constructors are discussed in this section.

Recall the context-free constructor `Make_ProcedureCall`, discussed in Section 2.2, which constructs a `ProcedureCall` and ensures the context-free validity of the result by requiring that its arguments be an `Identifier` and an `ExpressionList`. The corresponding context-dependent constructor might be called `CD_Make_ProcedureCall` and would also construct an instance of a `ProcedureCall`, but only if it could ensure the context-dependent validity of the result, using pertinent context-dependent rules. (Here, a syntactic object is deemed valid if it is free of violations or if it could appear in some context in which it is free of violations.) If the result would be invalid, the routine would have to raise an exception or perhaps cause a halt. Generic editing routines such as `Delete` or `Replace` could also have counterparts that ensure their context-dependent correctness of the result.

Given a complete specification of a context-dependent language, context-dependent constructors could conceivably be derived. However, such constructors would perform checking that could be redundant and obstructive.

4.3.1 Redundant

First, context-dependent correctness is not relevant to many of the manipulations that the software developer might perform on syntactic objects. For example, Merks [Mer87] constructed a Modula-2 compiler based on a GRAMPS-style SMS. The compiler first checks the context-dependent correctness of a program, and then performs a series of semantics-preserving (and thus, correctness-preserving) transformations which gradually reduce the original program into an equivalent one expressed in assembly-level constructs. If these correctness-preserving transformations were performed using

[4] It is unclear what conditions will guarantee that this process terminates. One necessary condition is that the relations be logically consistent. A sufficient condition (ignoring maker relations) is that no recursive relation contain a negation (or if-condition) referencing a relation in the same recursive cluster.

context-dependent constructors, all the checking done by the constructors would be redundant. Some benefit might result during the development of the transformations, when the failure of a context-dependent constructor would indicate that a transformation is not correctness-preserving and thus, not semantics-preserving. However, the same benefit can be obtained without using context-dependent constructors, simply by explicitly checking the validity of the result of each transformation.

4.3.2 Obstructive

Secondly, context-dependent constructors can be obstructive when performing transformations. Since transformations are an important application of SMSs, this case deserves careful consideration. The clearest, most efficient, and perhaps only way for a software developer to implement a non-atomic transformation may be as a series of editing operations where the intermediate objects violate the context-dependent syntax.

For example, consider a metaprogram that expands subprogram-calls. (Such expansion is also known as "inline-coding" and "unfolding".) A common problem arising in such a metaprogram occurs when the name of an entity referenced by the called subprogram denotes a different entity at the calling-point. This "naming-conflict" is a problem because if the subprogram call were to be expanded naively by replacing the subprogram-call with a copy of the subprogram-text, the wrong entity would be referenced by occurrences of the name in the copy of the subprogram-text. To resolve a naming-conflict, the metaprogram must rename one of the conflicting entities before the expansion takes place. The effect of this transformation is that all occurrences of the name of the entity are changed to a different name, a new one that does not conflict.

The entity-renaming transformation can be performed simply, using a context-dependent routine to find all occurrences of the entity's name, and a context-free editing routine to make each individual name-change. However, the intermediate stages of this method will generally violate the context-dependent syntax of the target-language: if an applied occurrence is edited first, it becomes undefined, since there is no declaration yet for the new name; if the defining occurrence is changed first, *all* the applied occurrences become undefined, since there is no longer a declaration for the old name. Thus, if this method were to be attempted using a context-dependent editing routine, it would fail at the very first change. It might be possible, depending on the target-language and the individual case, to introduce a declaration for the entity under the new name, change all the applied occurrences of the old name, then delete the declaration under the old name. Failing that, it is still possible that some more contrived sequence of context-dependent editing operations will succeed, but the point seems clear: performing a conceptually simple transformation becomes unnecessarily difficult or even impossible using context-dependent constructors.

As another example of how context-dependent constructors can be obstructive, a software developer might be writing an environment to guide a novice user through the subtleties of the target-language's context-dependent syntax. Such an environment would have to be able to endure syntactic objects with violations, so that these violations can be demonstrated and explained to the user. In this and other applications, the end-user does not use or interact with the SMS because there is another layer of software inbetween. Thus, the SMS should be as flexible as possible, not making decisions that unnecessarily restrict the applications that can use it. To disallow the creation of objects containing violations would be just such an unnecessary restriction.

4.3.3 Summary

In summary, context-free constructors are essential to an SMS, while context-dependent ones hinder the software developers using the SMS and make their code inefficient. Thus, context-dependent constructors are not provided in a NURN SMS. Instead, the SMS provides investigative routines with which one can obtain context-dependent information about objects, including whether they contain any violations of the context-dependent syntax.

4.4 Generating the Routines of a NURN SMS

In this section, I describe several schemes for generating the investigative routines of a NURN SMS. The routines of the different schemes provide equivalent capabilities, but they differ in their calling syntax, their genericity, and the amount of checking that they can force the host-language compiler to do at compile-time (rather than the routines at run-time). This section will concentrate on the user's view of the routines, i.e., their "headers", because there is little difference in their implementations[5]. Currently, Ginger does not generate any investigative routines, because the only application so far (a syntax-checker for Pascal) has only required a small, fixed interface to the investigators. I look forward to future research, involving substantial use of SMSs, to determine which of these schemes (or which combination of schemes) is the least unpleasant to use.

To generate the investigative routines, we need a scheme that will map query forms into the routine-call syntax of the host-language. Ideally, the scheme would preserve the readability of the simple-primaries in a NURN grammar. Unfortunately, most programming languages have a very restricted routine-call syntax: the routine-name followed by a list of arguments. Within this syntax, we need to specify the relation, the binding pattern, and entities for the bound parameters. The latter are certainly arguments to the routine, but each of the relation and binding pattern can be specified either in the name of the routine

[5] Mostly they just inspect their parameters, find the appropriate investigator, call the interpreter (passing it the investigator and the parameters), then unpack the results if necessary.

or in the argument list. Thus, schemes for generating SMS routines can be divided into four major kinds, one for each combination of choices:

1. Relation in name, binding pattern in name.
2. Relation in name, binding pattern in argument list.
3. Relation in argument list, binding pattern in name.
4. Relation in argument list, binding pattern in argument list.

Each of these four kinds of scheme will be considered in a separate subsection.

Each scheme will be demonstrated, using Modula-2 [Wir85] as the host-language, on a set of nine query forms:

```
B is_undefined
B is_the_defining_occurrence_of B
B is_the B th_Identifier_in B

U IS_A_VIOLATION
U is_a_Identifier_in B
B is_the_defining_occurrence_of U

U is_the_char_type
U is_the_defining_occurrence_of B
B is_a_Identifier_in U
```

The first three simply confirm whether the given bindings constitute an instance of the relation. The middle three result in a set of nodes, each of which can be bound to the unbound parameter. The last three are functional on the bound parameters, and thus result in a single node (if any). The examples will show how the routines generated for these query forms could be called in the context of a program using the SMS. For that purpose, assume the following variable declarations:

```
VAR
  violations, ids_in_idl, uses_of_d : EntityList ;
  x, d, id, idl, char_type : Node ;
```

Because NURN deals with three classes of entity (nodes, integers, and character-sequences), a NURN SMS defines the `Entity` data type as a discriminated union of `Node`, `Integer`, and `String`. An `EntityList` is a list of entities and an `Entity2List` is a list of pairs of them, etc. The SMS provides generic routines for extracting the individual entities from such structures. To avoid unnecessarily complicating the presentation, the code in this section will pretend that the `Entity` type is inter-assignable with the types it unites. Thus, for example, an `Integer` value can be assigned to an `Entity` variable or passed to an `Entity` value parameter, and an `Entity` value standing for an `Integer` value can be assigned/passed to an `Integer` variable. (This is possible in Algol68 and object-oriented languages and untyped languages.) The actual code needed in Modula-2 is somewhat more verbose. For example, one would need coercion routines such as the following:

```
PROCEDURE EntityFromInteger ( i : Integer ) : Entity ;
PROCEDURE EntityToInteger ( e: Entity ) : Integer ;
```

to effect the assignments or parameter-passing described above.

4.4.1 Relation in name, binding pattern in name

The first scheme for generating investigative routines puts both the name of the relation and the binding pattern in the name of the routine. Adapting the notation for query forms presented in Section 4.1, with blanks replaced by underscores, yields the following routines for the relation `is_effective_over`:

```
PROCEDURE B_is_effective_over_B (dp,region: Entity) : BOOLEAN ;
PROCEDURE U_is_effective_over_B (region: Entity) : EntityList ;
PROCEDURE B_is_effective_over_U (dp: Entity) : EntityList ;
PROCEDURE U_is_effective_over_U () : Entity2List ;
```

Since the query form is statically determined by the routine name, the nature of the query result is known at generation-time, so the routine can be declared with a fairly specific result-type (e.g., `BOOLEAN` vs. `EntityList` vs. `Entity2List` above).[6] If a relation is functional on some set of bound parameters, the corresponding routine is declared to return an `Entity` rather than an `EntityList`. (If an invocation of such a routine does not find a result, it returns the distinguished `Entity None`.) Demonstrating this scheme for the nine query forms:

```
IF B_is_undefined (x) THEN ...
IF B_is_the_defining_occurrence_of_B (d, x) THEN ...
IF B_is_the_B_th_Identifier_in_B (id, 4, idl) THEN ...

violations := U_IS_A_VIOLATION () ;
ids_in_idl := U_is_a_Identifier_in_B (idl) ;
uses_of_d  := B_is_the_defining_occurrence_of_U (d) ;

char_type  := U_is_the_char_type () ;
d          := U_is_the_defining_occurrence_of_B (x) ;
idl        := B_is_a_Identifier_in_U (id) ;
```

The approach just outlined is sufficient, but in some cases it is possible to provide a routine with a more readable call syntax.

1. For the binding pattern `B`, we can simply capitalize the first letter of the relation's name. For example, instead of `B_is_undefined`, we would have:

   ```
   PROCEDURE Is_undefined (x:Entity) : BOOLEAN ;
   ```

   This could be used as follows:

   ```
   IF Is_undefined(x) THEN ...
   ```

   Applying this subscheme to the grammar-derived domain relations introduced in Subsection 3.3.1 yields the GRAMPS-style recognizers presented in Subsection 2.2.2. For example, the

[6] Moreover, the routine could be declared with more specific parameter-types (e.g., `Node` instead of `Entity` above).

GRAMPS-style construction rule for `Block` yields the domain relation `n is_a_Block`, which yields the routine:

```
PROCEDURE Is_a_Block (n: Node) : BOOLEAN ;
```

2. For the binding patterns `U` and `UB`, the bound variable (if any) occupies the same position with respect to the name of the relation in a `Simple_Primary` as does the argument-list with respect to the name of the routine. For non-functional query forms, we can prefix the relation's name with "`Every_entity_that_`", to obtain somewhat more readable routine-calls. For example, instead of the routines `U_is_undefined` and `U_is_effective_over_B`, we could have

```
PROCEDURE Every_entity_that_is_undefined () : EntityList ;
PROCEDURE Every_entity_that_is_effective_over
  (region: Entity) : EntityList ;
```

Moreover, if the relation's name begins with "`is_a`", we can replace this with "`Every`" to achieve greater abbreviation. For example, for the query forms `U is_a_declaration_point` and `U is_a_Identifier_in B`, we could have

```
PROCEDURE Every_declaration_point () : EntityList ;
PROCEDURE Every_Identifier_in (p:Entity) : EntityList ;
```

Analogously, for relations whose `U` or `UB` query form is functional (e.g., `U is_the_char_type`, `U is_the_defining_occurrence_of B`), we can prefix the relation's name with "`The_entity_that`":

```
PROCEDURE The_entity_that_is_the_char_type () : Entity ;
PROCEDURE The_entity_that_is_the_defining_occurrence_of
  (x:Entity) : Entity ;
```

or, if (as in these cases, as in most cases) the name of the relation begins with "`is_the_`", we can replace this with "`The`" to achieve greater abbreviation:

```
PROCEDURE The_char_type () : Entity ;
PROCEDURE The_defining_occurrence_of (x:Entity) : Entity ;
```

Applying this last subscheme to the component relations introduced in Subsection 3.3.1 yields the selectors that would be derived according to the GRAMPS scheme presented in Section 2.2. For example, the construction rule for `Block` yields the component relation `n is_the_LabelList_of p`, which yields the routine:

```
PROCEDURE The_LabelList_of (p: Entity) : Entity ;
```

4.4.2 Relation in name, binding pattern in argument list

In the second scheme, the name of the relation[7] is used as the investigative routine's name, and the binding pattern is indicated in the argument list, which includes arguments for all of the relation's parameters. With this approach, only one routine is generated for each relation; this routine works for all of the

[7] A ternary relation has a two-part name; this is converted to a single name by joining the parts with "__".

relation's binding patterns. For example, the routine for the relation `dp is_effective_over region` is

```
     PROCEDURE is_effective_over (dp,region:Entity) : Result ;
```

Each call to a relation's investigative routine specifies a query form and simultaneously a query by passing in entities for the bound parameters, and the distinguished `Entity` value `Unknown` for the unbound parameters. For example, the call

```
     is_effective_over ( Unknown, r )
```

corresponds to the query form `U is_effective_over B`, and returns a `Result` representing all defining-points effective over `r`.

In contrast to the first scheme, a routine does not correspond to a single query form, and thus, little can be known about the query result at generation-time. Thus, the data type `Result` has to be a discriminated union of all the possible result types, i.e., `BOOLEAN`, `Entity`, `EntityList`, and so on. The consequent need for run-time discrimination means a drop in efficiency and some clumsiness in dealing with `Results`.

The demonstration queries come out as follows:

```
     IF ResultToBoolean ( is_undefined(x) ) THEN ...
     IF ResultToBoolean ( is_the_defining_occurrence_of(d,x) ) THEN ...
     IF ResultToBoolean ( is_the__th_Identifier_in (id,4,idl) ) THEN ...

     violations := ResultToEntityList ( IS_A_VIOLATION (Unknown) ) ;
     ids_in_idl := ResultToEntityList ( is_a_Identifier_in (Unknown,idl) ) ;
     uses_of_d  :=
       ResultToEntityList ( is_the_defining_occurrence_of (d,Unknown) ) ;

     char_type  :=
       ResultToEntity ( is_the_char_type (Unknown) ) ;
     d :=
       ResultToEntity ( is_the_defining_occurrence_of (Unknown,x) ) ;
     idl :=
       ResultToEntity ( is_a_Identifier_in (id,Unknown) ) ;
```

4.4.3 Relation in argument list, binding pattern in name

In the third scheme, the binding pattern appears in the name of the routine, and the relation is indicated in the argument list. Thus, there is one routine for each binding pattern, and this routine works for all relations that can have that binding pattern. The relation of interest is specified by name in the argument list as a value of the data type `Reln`, which could be `String` or a language-specific enumeration.[8]

[8] If `String` is used, then the routines are generic, and need not be generated for each target-language, provided that the target-language does not have relations with more than three parameters.

```
PROCEDURE B ( reln:Reln; x:Entity ) : BOOLEAN ;
PROCEDURE U ( reln:Reln           ) : EntityList ;

PROCEDURE BB ( reln:Reln; x1,x2:Entity ) : BOOLEAN ;
PROCEDURE BU ( reln:Reln; x1   :Entity ) : EntityList ;
PROCEDURE UB ( reln:Reln;    x2:Entity ) : EntityList ;
PROCEDURE UU ( reln:Reln               ) : Entity2List ;

PROCEDURE BBB ( reln:Reln; x1,x2,x3:Entity ) : BOOLEAN ;
PROCEDURE BBU ( reln:Reln; x1,x2   :Entity ) : EntityList ;
PROCEDURE BUB ( reln:Reln; x1,   x3:Entity ) : EntityList ;
PROCEDURE UBB ( reln:Reln;    x2,x3:Entity ) : EntityList ;
PROCEDURE BUU ( reln:Reln; x1      :Entity ) : Entity2List ;
PROCEDURE UBU ( reln:Reln;    x2   :Entity ) : Entity2List ;
PROCEDURE UUB ( reln:Reln;       x3:Entity ) : Entity2List ;
PROCEDURE UUU ( reln:Reln                  ) : Entity3List ;
```

(In my NURN grammar for Pascal, no relation has more than three parameters, so this set of routines would suffice.) In addition, for functional query forms with one unbound parameter, the SMS can provide more convenient routines that return an `Entity` rather than an `EntityList`

```
PROCEDURE Uf ( reln:Reln ) : Entity ;

PROCEDURE BUf ( reln:Reln; x1   :Entity ) : Entity ;
PROCEDURE UBf ( reln:Reln;    x2:Entity ) : Entity ;

PROCEDURE BBUf ( reln:Reln; x1,x2   :Entity ) : Entity ;
PROCEDURE BUBf ( reln:Reln; x1,   x3:Entity ) : Entity ;
PROCEDURE UBBf ( reln:Reln;    x2,x3:Entity ) : Entity ;
```

For the demonstration example, we will assume that `Reln` is a language-specific enumeration, e.g.,

```
TYPE Reln =
  ( IS_A_VIOLATION,
    is_a_Identifier_in,
    is_the__th_Identifier_in,
    is_the_char_type,
    is_the_defining_occurrence_of,
    is_undefined,
    ...
  ) ;
```

Using this enumeration, we can write the queries as follows:

```
IF B   (is_undefined, x) THEN ...
IF BB  (is_the_defining_occurrence_of, d, x) THEN ...
IF BBB (is_the__th_Identifier_in, id, 4, idl) THEN ...

violations := U  (IS_A_VIOLATION) ;
ids_in_idl := UB (is_a_Identifier_in, idl) ;
uses_of_d  := BU (is_the_defining_occurrence_of, d) ;
```

```
char_type    := Uf  (is_the_char_type) ;
d            := UBf (is_the_defining_occurrence_of, x) ;
idl          := BUf (is_a_Identifier_in, id) ;
```

4.4.4 Relation in argument list, binding pattern in argument list

In the fourth scheme, both the relation and the binding pattern are indicated in the argument list. The binding pattern is encoded using `Unknown`, and the relation is specified using the `Reln` type. Because all this information appears in the argument list, this is the one scheme in which the arguments can appear in their "proper" place with respect to the name of the relation. Thus, the format of the argument-lists reflects the syntax of simple-primaries:

```
PROCEDURE UnaryQuery
  ( arg: Entity; rel: Reln ) : Result ;
PROCEDURE BinaryQuery
  ( left: Entity ; rel: Reln ; right: Entity ) : Result ;
PROCEDURE TernaryQuery
  ( left     : Entity ;
    rel_left : Reln ;
    centre   : Entity ;
    rel_right: Reln ;
    right    : Entity ) : Result ;
PROCEDURE NaryQuery ( rel: Reln; args: EntityList ) : Result ;
```

The caveats regarding the `Result` type discussed in Subsection 4.4.2 also apply here. Also, as in Subsection 4.4.3, if the `Reln` type is `String` these routines would be generic and could be made to work for all target-languages. If the `Reln` type were an enumerated-type, it would be declared slightly differently than in Subsection 4.4.3, because ternary relations have their two-part names in two parts here. For the demonstration example, it would appear as

```
TYPE Reln =
  ( IS_A_VIOLATION,
    is_a_Identifier_in,
    is_the,
    is_the_char_type,
    is_the_defining_occurrence_of,
    is_undefined,
    th_Identifier_in,
    ...
  ) ;
```

and the queries would appear as follows:

```
IF ResultToBoolean
  ( UnaryQuery(x,is_undefined) ) THEN ...
IF ResultToBoolean
  ( BinaryQuery(d,is_the_defining_occurrence_of,x) ) THEN ...
IF ResultToBoolean
  ( TernaryQuery(id,is_the,4,th_Identifier_in,idl) ) THEN ...
```

```
       violations :=
        ResultToEntityList
          ( UnaryQuery(Unknown,IS_A_VIOLATION) ) ;
      ids_in_idl  :=
        ResultToEntityList
          ( BinaryQuery(Unknown,is_a_Identifier_in,idl) ) ;
      uses_of_d  :=
        ResultToEntityList
          ( BinaryQuery(d,is_the_defining_occurrence_of,Unknown) ) ;

      char_type :=
        ResultToEntity
          ( UnaryQuery  (Unknown,is_the_char_type) ) ;
      d :=
        ResultToEntity
          ( BinaryQuery (Unknown,is_the_defining_occurrence_of,x) ) ;
      idl :=
        ResultToEntity
          ( BinaryQuery (id,is_a_Identifier_in,Unknown) ) ;
```

4.5 Generating the Investigators of a NURN SMS

A NURN grammar is a declarative specification of a context-dependent language. A corresponding SMS includes investigators which answer queries about syntagms with respect to the specification. Conceivably, these investigators could be derived from the NURN grammar by hand, but for a typical target-language, the SMS would be quite large and its investigators fairly complex. Therefore, rather than build a NURN SMS for a particular language, it is wiser to construct a system to generate a NURN SMS for any given language. This section describes Ginger, an implemented approach to the problem of generating reasonably efficient investigators from the rules of a NURN grammar.

Ginger generates a NURN SMS according to the following outline:

1. Generate the context-free SMS.
2. Construct the predefined relations.
3. Perform static consistency checks on the relation rules.
4. Reduce the relation rules to a canonical form.
5. Annotate the canonical rules with information used in the next step.
6. Construct investigators from the canonical rules and their annotations.

4.5.1 Generating the context-free SMS

The input NURN grammar consists of context-free GRAMPS-style rules and context-dependent relation rules. These two sets of rules are separated, and the context-free rules are given to my implementation of GRAFS, which generates a GRAMPS-style SMS from them.[9]

4.5.2 Constructing the predefined relations

As we saw in Subsection 3.3.6, it is useful when manipulating Pascal to create declarations for its predefined entities. Analogously, Ginger (which manipulates NURN) finds it useful to construct `Definition` rules for its predefined relations (see Subsection 3.3.1). There is a fixed set of primitive relations and a language-specific set of derived relations; Ginger derives definitions for the latter relations from the GRAMPS-style subgrammar for the target-language.

4.5.3 Static consistency checks

NURN performs static consistency checks on the relation rules to detect syntax errors which would preclude the suitability of the rules for further (extensive) processing.

Most of the checks deal with `Declarators`, which appear (one each) in `Definitions` and `Declarations`. For convenience of reference, here is the syntax for a `Declarator`, as it appears in Appendix 1:

```
CONSTRUCT Declarator IS
  [ <:Kind_Indicator> ] <:Simple_Primary>
  [ _ ":" "FUNCTIONAL" "ON" <FunctionalBoundVarSets:VarSet_List> ]
  [ "WITH" <Profiles:Profile_List> ]
```

and here are the constraints on `Declarators` that Ginger enforces:

- The `Simple_Primary` must declare a unique relation.
- The parameters in the `Simple_Primary` must be distinct.
- A parameter in the `Simple_Primary` must be a `Var`, not a `Literal`.
- A `Var` appearing in the `FunctionalBoundVarSets` must reference a parameter in the `Simple_Primary`.
- A violation relation must have an arity of 1.
- A user-written rule cannot declare a derived or primitive relation.
- A user-written rule must have empty `Profiles`.

Here are the static constraints on a `Simple_Primary` appearing anywhere other than within a `Declarator`:

- The relation it references must have been declared, either explicitly or implicitly.

[9] Because the context-free subgrammar can remain fairly stable while the relation rules are being developed, a context-free SMS is generated only if the subgrammar has changed since the last generation.

- Its arity and parameter–types (node, integer, or string) must agree with the declaration of the relation referenced.
- If it occurs in the head of a `Subdefinition`, the relation it references must have been declared in a `Declaration`, not in a `Definition`.

There are two constraints dealing with maker relations and `Make` primaries:

- A maker relation must contain at least one `Make`;
- A `Make` cannot occur outside of a maker relation.

It is interesting to note that all of these constraints could be expressed in a NURN grammar for NURN grammars.

4.5.4 Reduction to canonical form

The relation rules are transformed so that each relation is defined by a `Definition` rule. That is, if a relation has a distributed definition consisting of a `Declaration` and `Subdefinitions`, the bodies of its various `Subdefinitions` are joined into a single disjunction, which forms the body of a new `Definition` rule[10]. The head of the new rule is the `Declarator` appearing in the `Declaration`, with the kind and functionality indicators made explicit if they were omitted. For example, assume for the sake of illustration that the three subdefinitions given for the `is_effective_over` relation in Section 3.3 constitute its full definition. Then the canonical rule for `is_effective_over` is:

```
RELATION DEF NORMAL dp is_effective_over r: FUNCTIONAL ON {dp,r} :-
  dp is_a_Label_in ll,
  ll is_the_LabelList_of r,
  r is_a_Block
OR
  dp is_the_Lhs_of cd,
  cd is_a_ConstantDefinition_in cdl,
  cdl is_the_ConstantDefinitionList_of r,
  r is_a_Block
OR
  dp is_an_Identifier_in idl,
  idl is_the_IdentifierList_of etd,
  etd is_a_EnumeratedTypeDefiner,
  r is_the_Block_closest_containing etd
.
```

The reduction to canonical form allows Ginger to treat all rules uniformly, since they are all now `Definition` rules.

[10] This often involves renaming the `Subdefinition`'s parameters to achieve consistency with the `Declaration`.

4.5.5 Annotation

Recall that in a relation rule, a conjunction comprises a list of primaries. One of the major problems in generating investigators is to find an efficient ordering (in a sense that will be defined in the next subsection) for the primaries in each conjunction. To this end, Ginger *annotates* each primary with information that will aid this search.[11] In fact, Ginger annotates many other nodes of the canonical relation rules (essentially, everything from definitions down to simple–primaries), because the annotation of a primary usually depends on the relation it references (if it is a simple–primary) or its children (if it is a structured–primary). Such dependencies tend to be recursive, and the Control aspect introduced in Section 4.2 is in fact general enough to deal with recursive relations (as well as recursive investigators), propagating calculated information around the nodes of the relations rules.

We can identify three major annotations: *upvars*, *profiles*, and *basics*.

`upvars`: The *upvars* of a primary are the variables referenced within it that are also referenced outside it[12]. Informally, a primary's upvars are the variables by which it communicates with its neighbouring primaries. For example, recall the following rule from Subsection 3.3.5:

```
(* A Label or Identifier
   (other than the Name of a Program or
    an Identifier in the Parameters of a Program)
   must have a defining occurrence.
*)
RELATION DEF VIOLATION x is_undefined :-
  (x is_a_Label OR x is_a_Identifier),
  NOT
  (
    x is_the_Name_of p,
    p is_a_Program
  OR
    x is_a_Identifier_in idl,
    idl is_the_Parameters_of p,
    p is_a_Program
  ),
  NOT (dp is_the_defining_occurrence_of x)
.
```

This definition contains a single (top–level) conjunction comprising three primaries: the first is a group (a parenthesized disjunction), and the second and third are negations. For each of the three primaries, the only upvar is `x`; neither `p` in the second primary nor `dp` in the third is used outside the primary. As a further example, within the body of the second top–level primary, the upvars of each primary is the set of

[11] Ginger uses the annotation facility introduced in my implementation of GRAFS, which allows arbitrary information to be associated with any `Node`. The facility is much like that provided by property lists in Lisp [WinHor89].

[12] This is one annotation that does not entail recursive propagation.

variables it references.

`profiles`: A primary may have many *profiles*, each of which is a consistent mapping from its upvars (those that are node-variables) to target-language node-classes. For example, the simple-primary `id is_the_Callee_of c` has two profiles:

```
(id:Identifier, c:FunctionCall)
(id:Identifier, c:ProcedureStatement)
```

This reflects the fact that both `FunctionCalls` and `ProcedureStatements` have a `Callee` component, which in either case is an `Identifier`. The profiles of derived relations can be derived from the GRAMPS-style subgrammar for the target-language. These profiles are then propagated through the nodes of the canonical relation rules.

`basics`: A *base* for a primary is a subset of its upvars assumed (for whatever purpose) to be bound.[13] In particular, each base of a simple-primary corresponds to a query form for the relation referenced by the simple-primary. For example, the simple-primary `id is_a_Identifier_in idl` has four possible bases, corresponding to the four query forms for the relation `is_a_Identifier_in`:

```
{id,idl} -- B is_a_Identifier_in B
{id}     -- B is_a_Identifier_in U
{idl}    -- U is_a_Identifier_in B
{}       -- U is_a_Identifier_in U
```

In the next subsection we will need to have a rough idea of the behaviour of a primary for various bases. This behaviour will be used heuristically as an estimate of the size of the result of investigating the primary with particular bindings for the variables in the base. The *basics* of a primary tell Ginger the behaviour of a primary for any of its bases. Ginger distinguishes the following behaviours (in order, from "best" to "worst"): *confirmative*, *functional*, *direct*, *feasible*, and *infeasible*. These behaviours are defined in terms of a hypothetical investigation of the primary with particular bindings for the variables in the base:

1. `confirmative`: This is a primary's behaviour for the base with all upvars bound. The hypothetical investigation can merely confirm or deny whether the bindings satisfy the (sub)relation that the primary defines.
2. `functional`: The investigation can result in at most one binding for each unbound upvar in the base.
3. `direct`: The investigation can avoid involving the `U` investigator for any domain relation (see Subsection 3.3.1).
4. `feasible`: The investigation can avoid involving any infeasible investigators.
5. `infeasible`: The investigation cannot avoid involving an infeasible investigator.

Note that the first four behaviours form an inclusive series. That is, if a primary is confirmative on some base, it is also functional, direct and feasible on that base. However, this is normally left unstated. For

[13] The word "base" arose as an abbreviation for "bound argument set".

example, the predefined relation `n is_a_Identifier_in p` has the following basics:

```
{n,p}: confirmative
{n}: functional
{p}: direct
{}: feasible
```

Certain primitive investigators are designated *a priori* as infeasible. These correspond to operations with respect to primitive relations that Ginger does not implement. Often this is because the results would normally be very large. For instance, `U is U`, `U is_the_parent_of U`, and `U is_before B` would each result in roughly as many instances as there are nodes installed in the SMS; `U contains U` and `U is_before U` would result in even more instances. On the other hand, `B is_the_character_sequence_of U` involves going from a character–sequence to all lexemes having an equal character–sequence. This would not normally be an excessively large result, but it is something that Ginger simply does not currently do. Ginger complains if it is forced to generate an infeasible investigator.

Starting with the *a priori* basics of primitives, the basics of nodes in the canonical relations rules are calculated by combining the basics of nodes according to their class. For example, we will consider calculating the basics of a disjunction. A disjunction is confirmative (or direct or feasible) on some base if all of its conjunctions are confirmative (or direct or feasible) on that base. A disjunction is infeasible on a base if *any* of its conjunctions is infeasible on that base. The trickiest case for a disjunction occurs when trying to determine whether it is functional on some base. It is necessary that all of its conjunctions be functional on that base, but this is not sufficient; it is also necessary that the conjunctions be mutually exclusive on that base. That is, it is necessary that for any possible binding of entities to the variables of the base, at most one conjunction can supply an instance matching those bindings. For example, consider the disjunction:

```
x is_the_LeftOperand_of expr OR x is_the_RightOperand_of expr
```

and the base `(expr)`. (Assume that both `x` and `expr` are upvars of the disjunction.) Both conjunctions are functional on this base (a node `expr` can have at most one `LeftOperand` and at most one `RightOperand`), but the disjunction is not functional on `expr` because if `expr` is bound to a `dyadic_expression`, both conjunctions can supply a binding for `x`. On the other hand, the disjunction *is* functional on `x`, because a given node `x` cannot be both the `LeftOperand` of an `expr` and the `RightOperand` of an `expr`. For completeness, we observe that the disjunction is confirmative on the full base `(x,expr)` and feasible on the empty base `()`. Thus, the basics of this disjunction are:

```
{x,expr}: confirmative
{x}     : functional
{expr}  : direct
{}      : feasible
```

Profiles are used to help determine whether two conjunctions are mutually exclusive on some base. If we project the profiles of a conjunction onto the base's variables, we get consistent sets of node–classes for the base's variables. If these projections are disjoint between conjunctions, then the conjunctions must be mutually exclusive. However, the converse is not always true. For example, consider the disjunction:

```
    c is_a_FunctionCall, id is_the_Callee_of c
    OR
    c is_a_ProcedureStatement, id is_the_Callee_of c
```

Both conjunctions are functional on `c` and on `id`. Each conjunction has a single profile; the first conjunction's profile is `(c:FunctionCall,id:Identifier)` and the second's is `(c:ProcedureStatement,id:Identifier)`. The projections onto `c` are `(c:FunctionCall)` and `(c:ProcedureStatement)`, which are disjoint, and thus we can conclude that conjunctions are mutually exclusive on `c`, and thus that the disjunction is functional on `c`. On the other hand, the projections onto `id` are `(id:Identifier)` and `(id:Identifier)`, which are not disjoint. Thus, we cannot conclude from this test that the conjunctions are mutually exclusive on `id` or that the disjunction is functional on `id`, although both assertions are in fact true. Ginger is currently fairly dumb on this point, and would have to be told explicitly that this disjunction is functional on `id`.

Since relations are declared with explicit functionalities (using the optional `FUNCTIONAL ON ...` part of declarators), there is an opportunity for a consistency check, comparing the stated and deduced functionalities. If the two differ, a diagnostic is produced. There are three possible reasons for such a discrepancy:

1. The grammar–writer made a mistake in the grammar.
2. Ginger is not smart enough to deduce the relation's functionality correctly.
3. The relation's functionality is *intended* to be different (better) than is derivable. For example, the body of the definition of the relation `dp is_the_defining_occurrence_of x` does not define a relation that is functional on `x`, but any non–functional extension of this relation constitutes a violation of the context–dependent syntax of Pascal, and would be caught by the violation relation `is_a_conflicting_declaration_point`.

When the user has determined the cause for the discrepancy, the appropriate actions are, respectively:

1. Correct the grammar.
2. Ignore the diagnostic, and hope that Ginger's error does not have serious repercussions when generating investigators. (Or currently, use the ad hoc facility mentioned above to inform Ginger of the correct functionality.)
3. Make sure that the grammar defines a violation relation corresponding to non–functional behaviour of the relation, and then ignore the diagnostic. (It is not going to go away.)

4.5.6 Construction of investigators

Recall that an *investigator* for a query form is an algorithm that answers all queries that are instances of the query form. Ginger expresses investigators using a syntax quite similar to that for relation rules. For example, consider the hypothetical canonical rule for `is_effective_over` that was created in Subsection 4.5.4:

```
RELATION DEF NORMAL dp is_effective_over r: FUNCTIONAL ON {dp,r} :-
  dp is_a_Label_in ll,
  ll is_the_LabelList_of r,
  r is_a_Block
OR
  dp is_the_Lhs_of cd,
  cd is_a_ConstantDefinition_in cdl,
  cdl is_the_ConstantDefinitionList_of r,
  r is_a_Block
OR
  dp is_an_Identifier_in idl,
  idl is_the_IdentifierList_of etd,
  etd is_a_EnumeratedTypeDefiner,
  r is_the_Block_closest_containing etd
.
```

For the query form `U is_effective_over B`, Ginger could generate the following investigator:

```
TO GET ALL {dp} SUCH THAT dp is_effective_over r,
  CONFIRM THAT r is_a_Block
  GET {ll} SUCH THAT ll is_the_LabelList_of r,
  GET ALL {dp} SUCH THAT dp is_a_Label_in ll
  PROJECT AWAY {ll}
  OR
  CONFIRM THAT r is_a_Block,
  GET {cdl} SUCH THAT cdl is_the_ConstantDefinitionList_of r,
  GET ALL {cd} SUCH THAT cd is_a_ConstantDefinition_in cdl,
  GET {dp} SUCH THAT dp is_the_Lhs_of cd,
  PROJECT AWAY {cd, cdl}
  OR
  GET ALL {etd} SUCH THAT r is_the_Block_closest_containing etd,
  CONFIRM THAT etd is_a_EnumeratedTypeDefiner,
  GET {idl} SUCH THAT idl is_the_IdentifierList_of etd,
  GET ALL {dp} SUCH THAT dp is_an_Identifier_in idl
(FORGET)
.
```

An investigator should express a reasonable method of calculating the answer to a query in terms of the bound parameters of the query and the answers to subqueries. We will consider the calculation problem in a top–down manner, with reference to the the GRAMPS–style grammar for NURN rules found in Appendix 1. To do so, it is necessary to generalize some ideas. Here, any node of a definition rule (down to a simple–primary) defines a relation on its upvars, and can supply instances of its extension to answer a query. I give a theoretical statement of the extension of a node (NURN semantics), followed by a practical statement of how such a node can answer a query (Ginger implementation). It is this latter statement that the investigators and their subparts express.

`Definition`: A definition's extension (and thus, the extension of the relation it defines) is just the extension of its body (a disjunction). Thus, a definition's answer to a query is the answer of its body to the same query.

`Disjunction`: A disjunction's extension is the union of the extensions of its conjunctions. Thus, a disjunction's answer to a query is the union of its conjunctions' answers to same query. The conjunctions' answers are obtained in the order they appear in the disjunction, but this order makes little difference: each will add its answer to the disjunction's answer.

`Conjunction`: A conjunction's extension is the join of the extensions of its primaries, projected onto the conjunction's upvars. Thus, a conjunction's answer to a query is that join, selected for the bindings of the query. Ginger's technique for arriving at this answer is fairly involved, and is described later.

`Simple_Primary`: A simple–primary's extension is the extension of the relation (definition) it references, with a name–change for each parameter corresponding to a variable argument and a selection–projection for each parameter corresponding to a literal argument. Thus, a simple–primary's answer to a query is obtained as follows:

1. Translate the query's bindings to bindings for the parameters of the referenced definition.
2. Add bindings for any literals appearing in the simple–primary.
3. Give the resulting query to the referenced definition.
4. Translate each instance in the answer back into the simple–primary's variables.

`Group`: A group's extension is simply the extension of its body (a disjunction). Thus, a group's answer to a query is its body's answer to the same query.

`Negation`: A negation's extension is the complement of the extension of its body. (The complement of an extension is the set of all possible bindings for the variables of the extension, minus the bindings that are instances in the extension.) This extension is typically huge. Even projecting it onto a single variable base usually results in most of the nodes in existence. Therefore, a negation is infeasible for anything less than a fully–bound query; for the fully–bound query, it is confirmative, and it answers yes iff the body's answer to the same query is no.

`If`: Ifs were expanded during the reduction of relation rules to canonical form. Thus, they are not considered here.

`Make`: The extension of a make is easier to understand in terms of the extension of the maker relation in which it occurs (see Subsection 3.3.6).

We deferred the discussion of how to calculate a conjunction's answer to a query because it is fairly involved. In the discussion that follows, we refer to the primaries of a conjunction by their position in a predetermined order that depends on which of the conjunction's upvars are bound. Answering a query on a conjunction with `n` primaries can be viewed as enumerating the paths of a `n+1`-level tree (a *query-tree*). Each level `i` in the tree has a corresponding set of bound variables `BV[i]`: `BV[0]` is the set of variables bound in the query to the conjunction, and for `i>0`, `BV[i]` is the union of `BV[i-1]` and the upvars of the `i`th primary. Each vertex at level `i` represents a set of bindings for the variables of `BV[i]`. The root of the tree (level `0`) represents the set of bindings supplied in the query to the conjunction. For `i>0`, a vertex `u` at level `i-1` is adjacent to a vertex `v` at level `i` iff the bindings of `v` are compatible with both the `i`th primary and the bindings of `u`. Thus, the vertices at level `n` (if any) represent bindings compatible with the initial bindings and all primaries. These can then be projected onto the upvars of the conjunction, to obtain the query's answer.

In effect, to answer a query on a conjunction, the NURN interpreter traverses the corresponding query-tree. At each vertex, it uses the set of bindings there to formulate a query to the appropriate primary, and uses the primary's answer to determine the adjacent vertices (sets of bindings) on the next level. If the vertex is on the bottom level, the interpreter projects the set of bindings onto the conjunction's upvars and saves the result as an instance in the conjunction's answer to the query.

Since the time required to answer the query is roughly proportional to the size (number of vertices) of the query-tree, we would like the query-tree to be as small as possible. Using the stated algorithm, the only way to influence the size of the query-tree is by the ordering of the primaries. Ideally, for each base of the conjunction, we would like an ordering that, for every possible query on that base, minimizes the size of the resultant query-tree. In general, however, there cannot be an ordering that is optimal in this sense. Given sufficient statistical information about subject syntagms and queries, one could conceivably find an ordering that minimizes the *expected* size of the query-tree, but such information is not usually available or reliable. Lacking an optimality criterion by which to judge orderings, it seems that the most one can ask is that the chosen ordering seem reasonable to someone who is familiar with the target-language.

The technique that Ginger uses to find reasonable orderings for the primaries of a conjunction is a heuristic approach using the information gathered in the previous subsection. In particular, Ginger uses the basics of each primary. For each base of a conjunction, Ginger starts with the variables of the base as the only bound variables, and selects the primary that has the "best" behaviour on this set of variables, according to the confirmative-to-infeasible rating scheme. (If a tie occurs, Ginger selects the leftmost of the primaries in the tie.) This becomes the first primary in the ordering, its upvars are added to the set of bound variables, and it is removed from further consideration. The process repeats until each primary has been appended to the ordering. This greedy method is an attempt to minimize the out-degree of vertices in the query-tree as early as possible. It seems to work fairly well.

4.6 Introductory Examples Revisited

We examine again the questions considered in Section 1.1, and show how the investigative routines of a NURN SMS can be used to answer these questions. We will use the routines generated by the scheme outlined in Subsection 4.4.1, including its convenient abbreviations.

1. "What is the type of this variable?"

   Pertinent relation:

   ```
   t is_the_type_of x: FUNCTIONAL ON {x}
   ```

   Pertinent query form and routine:

   ```
   U is_the_type_of B
   PROCEDURE The_type_of (x:Entity): Entity ;
   ```

   Let `v` represent the variable referred to in the question. Then

   ```
   The_type_of(v)
   ```

   returns a `Node` representing its type.

2. "Has NEW been redefined for this scope?"

   Pertinent relations:

   ```
   d defines_the_spelling $s if_it_occurs_at x:
     FUNCTIONAL ON {$s, x}, {d, x}
   r is_a_required_routine: FUNCTIONAL ON {r}
   ```

   Pertinent query forms and routines:

   ```
   U defines_the_spelling B if_it_occurs_at B
   B is_a_required_routine
   PROCEDURE U_defines_the_spelling_B_if_it_occurs_at_B
     (s,x:Entity): Entity
   PROCEDURE Is_a_required_routine (r:Entity): BOOLEAN ;
   ```

   Let `p` represent a point in the program where the definition of NEW is in question. Then

   ```
   U_defines_the_spelling_B_if_it_occurs_at_B ("new",p)
   ```

   returns the defining occurrence that `NEW` would have if it were inserted at `p`, and thus

   ```
   NOT Is_a_required_routine
    ( U_defines_the_spelling_B_if_it_occurs_at_B ("new",p) )
   ```

   returns `TRUE` iff `NEW` has been redefined.

3. "Will this identifier conflict with any existing identifiers?"

   Pertinent relations:

   ```
   $s is_the_spelling_of x: FUNCTIONAL ON {x}
   d defines_the_spelling $s if_it_occurs_at x:
     FUNCTIONAL ON {$s, x}, {d, x}
   ```

   Pertinent query forms and routines:

```
U is_the_spelling_of B
U defines_the_spelling B if_it_occurs_at B
PROCEDURE The_spelling_of (x:Entity): Entity ;
PROCEDURE U_defines_the_spelling_B_if_it_occurs_at_B
  (s,x:Entity): Entity ;
```

Let id represent the identifier in question, and let p represent the point at which it is to be introduced. Then

```
The_spelling_of(id)
```

returns the spelling of id, and

```
U_defines_the_spelling_B_if_it_occurs_at_B(The_spelling_of(id),p)
```

returns the defining occurrence that id would have if it were inserted p, and thus equals None iff there is no existing declaration for id to conflict with.

4. "Where is the resolution for this forward declaration?"

   Pertinent relation:

```
d_2 is_the_resolution_of d_1: FUNCTIONAL ON {d_2}, {d_1}
```

   Pertinent query form and routine:

```
U is_the_resolution_of B
PROCEDURE The_resolution_of (x:Entity): Entity ;
```

   Let f represent the forward-declared routine. Then

```
The_resolution_of(f)
```

   returns the resolution of f.

5. "Is this argument assignment-compatible with that parameter?"

   Pertinent relations:

```
t is_the_type_of x: FUNCTIONAL ON {x}
t_2 is_assignable_to t_1: FUNCTIONAL ON {t_1, t_2}
```

   Pertinent query forms and routines:

```
U is_the_type_of B
B is_comparable_with B
PROCEDURE The_type_of (x:Entity): Entity ;
PROCEDURE B_is_assignable_to_B (t_2,t_1:Entity): BOOLEAN ;
```

   Let a represent the argument and p represent the parameter. Then

```
B_is_assignable_to_B (The_type_of(a), The_type_of(p))
```

   returns TRUE iff a is assignment-compatible with p.

6. "What is the value of this constant-expression?"

   Pertinent relation:

```
val is_the_value_denoted_by c: FUNCTIONAL ON {c}
```

   Pertinent query form and routine:

```
U is_the_value_denoted_by B
PROCEDURE The_value_denoted_by (c:Entity): Entity
```

Let ce represent the constant-expression. Then

```
The_value_denoted_by(ce)
```

returns a node representing the value denoted by ce.

7. "Where are all the calls to this procedure?"

Pertinent relations:

```
n is_the_Name_of p: FUNCTIONAL ON {n}, {p}
x is_a_applied_occurrence_of d: FUNCTIONAL ON {x}
n is_the_Callee_of p: FUNCTIONAL ON {n}, {p}
fp is_the_formal_for ap: FUNCTIONAL ON {ap}
```

Pertinent query forms and routines:

```
U is_the_Name_of B
B is_the_Name_of U
U is_a_applied_occurrence_of B
B is_the_Callee_of U
U is_the_formal_for B
PROCEDURE The_Name_of (p:Entity) : Entity ;
PROCEDURE B_is_the_Name_of_U (n:Entity): Entity ;
PROCEDURE Every_applied_occurrence_of (c:Entity): EntityList ;
PROCEDURE B_is_the_Callee_of_U (n:Entity): Entity ;
PROCEDURE The_formal_for (ap:Entity): Entity ;
```

Let p represent the procedure. (Procedures are represented by ProcedureDeclarations.) Then

```
Every_applied_occurrence_of(The_Name_of(p))
```

returns a list of all applied occurrences of the name of the procedure. However, in a language with procedural parameters, not every applied occurrence is a call to the procedure. To find all the ProcedureStatements that call p, we iterate through the applied occurrences of p.

```
iter := GetIterator ( Every_applied_occurrence_of(The_Name_of(p)) )
WHILE NextEntity ( iter, applied_occ ) DO
  call := B_is_the_Callee_of_U (applied_occ) ;
  IF call<>None THEN
    (* <call> is a call to <p> *)
  END
END
```

If we are interested in all places where p *might* be called, including calls as a procedural parameter, then we can write the following recursive procedure:

```
PROCEDURE Find_all_possible_calls_to_routine (r:Entity) ;
VAR
  iter : Iterator ;
  applied_occ : Entity ;
  call : Entity ;
BEGIN
  iter :=
    GetIterator ( Every_applied_occurrence_of(The_Name_of(r)) ) ;
  WHILE NextEntity ( iter, applied_occ ) DO
    call := B_is_the_Callee_of_U (applied_occ) ;
    IF call=None THEN
      (* <applied_occ> is a procedural or functional parameter *)
      Find_all_possible_calls_to_routine
        ( B_is_the_Name_of_U(The_formal_for(applied_occ)) )
    ELSE
      (* <call> is a call to <p> *)
      (* do something with <call>! *)
    END
  END
END Find_all_possible_calls_to_routine_named ;
```

and invoke it with

```
Find_all_possible_calls_to_routine (p) ;
```

(Alternatively, we could urge the grammar-writer to include the auxiliary relation `is_a_possible_call_to` in the grammar, and use `Every_possible_call_to(p)`.)

Thus, I have demonstrated that a NURN SMS provides routines which allow queries about context-dependent entities and their relationships to be conveniently posed.

CHAPTER 5
CONCLUSIONS

5.1 Accomplishments

I have coined the term *syntactic manipulation system* (SMS) as a generalization of *metaprogramming system* (MPS). In my view, an MPS is a context-free SMS for a programming language. I have proposed and justified a distinction between two kinds of alternation rule in GRAMPS-style grammars, and suggested how this distinction should be reflected in GRAMPS-style SMSs. I have validated Terry's concept of GRAFS by using it extensively in my own work. At the same time, I have improved the implementation of GRAFS, taking it from a prototype to a sturdy implementation. Two notable additions are a more powerful parsing algorithm and an annotation facility.

I have devised and presented NURN, a new technique for defining the context-dependent syntax of target-languages, with particular emphasis on its use for defining programming languages. NURN features rules which use logical formulas to define relations on the nodes of abstract syntax-trees. I have recognized and named the *double-duty* strategy for representing context-dependent entities using context-free entities. This strategy was employed in previous ad hoc context-dependent extensions to context-free SMSs.

NURN has two major advantages over existing formalisms for specifying context-dependent syntax: its emphasis on relationships between nodes and its simplicity. In emphasizing relationships, it encourages formal language descriptions that use familiar concepts such as *defining-occurrence*, *argument-parameter binding*, and *naming conflict*. NURN's simplicity is reflected by its relatively simple syntax; one expresses relationships using logical formulas that are fairly close to English phrases. The combination of these two advantages makes NURN particularly useful for formalizing informal descriptions of context-dependent languages: given a document such as the Pascal Standard, one can translate it practically sentence-for-sentence, preserving most of the readability of the original, into a NURN grammar for the language. Such formalization is desirable because it brings out and resolves the incompleteness and inconsistency common to informal descriptions.

I have demonstrated NURN's usefulness by writing a complete NURN grammar for the syntax of Standard Pascal, with all its exceptions and anomalies. This grammar appears in Appendix 2. It is roughly 4800 lines long, including about 140 rules for the context-free syntax, and definitions for about 150 context-dependent relations.

A NURN SMS is a context-dependent SMS derived from a NURN grammar. I have characterized a NURN SMS as comprising *constructive* and *investigative* routines, and argued that the constructive routines should be derived solely from the context-free subgrammar for the target-language. I have identified various schemes for deriving the investigative routines from a NURN grammar, and shown how these routines can be used to answer context-dependent queries, such as those presented in Section 1.1.

I have designed and implemented Ginger, a system for generating context-dependent SMSs from NURN grammars. Ginger itself is about 11,000 lines of Modula-2 code; this count does not include GRAFS or the many application-independent utilities that they use. Of particular use was a general-purpose utility to control the propagation of values through a set of recursive definitions, which was gradually abstracted from Ginger-specific code.

To demonstrate the feasibility of automatically constructing a NURN SMS, I have applied Ginger to my NURN grammar for Standard Pascal. Although Ginger does not generate any host-language routines, it does generate investigators, which can be used by calling routines in a small fixed interface. Using these routines, I wrote an unsophisticated but technically complete syntax-checker, which detects violations of the context-free and context-dependent syntax of Standard Pascal. The syntax-checker has been validated using the Pascal Validation Suite, a large set of Pascal programs more generally used for validating and certifying Pascal processors such as interpreters and compilers (see Appendix 3).

5.2 Further Research

5.2.1 Improvements to NURN

Notation should bc added to NURN for creating and examining character-sequences.

One extension to NURN that might make it even more readable would be to allow the arguments of simple-primaries (currently required to be variables or literals) to be structured with functions. For instance, instead of saying

```
    RELATION SUBDEF id is_effective_over b :-
      id is_the_Lhs_of cd,
      cd is_a_ConstantDefinition_in cdl,
      cdl is_the_ConstantDefinitionList_of b,
      b is_a_Block
    .
```

one might say

```
    RELATION SUBDEF id is_effective_over b :-
      id is_the_Lhs_of
        ( a_ConstantDefinition_in
            ( the_ConstantDefinitionList_of b ) ),
      b is_a_Block
    .
```

Here, the elision of intermediate variables allows a smoother reading. One could even allow functions in rule heads, similar to the "pattern-matching" feature of modern functional languages. For example, one might say

```
RELATION SUBDEF
  the_Lhs_of
    ( a_ConstantDefinition_in
        ( the_ConstantDefinitionList_of b ) )
  is_effective_over b
:-
  b is_a_Block
.
```

which is almost identical in phrasing to the original comment:

```
(* The Lhs of a ConstantDefinition in the ConstantDefinitionList
   of a Block is a declaration-point effective over the Block.
*)
```

Whether this is more readable, however, is open to debate.

5.2.2 Improvements to Ginger

There are lots of opportunities for improving the speed of Ginger and the SMSs that Ginger generates:

1. Use faster data-structures. Many of the data-structures that Ginger uses were chosen for flexibility and ease of use. They could probably be replaced by more efficient structures.
2. Generate and compile host-language source-code equivalent to investigators. Currently, investigators are expressed using a syntax similar to that for relation rules and are interpreted. An alternative approach would have Ginger translate the investigators into equivalent host-language source-code, which would thcn be compiled into machine code. The code would be loaded as part of the SMS and executed directly when an interface routine was invoked. The current approach (interpretation) is generally applicable since it avoids most limitations of the host-language, but translation to source-code would be preferable if the operation of investigators can be expressed easily in the host-language. Host-language investigators are potentially more efficient, easier to debug, easier to make small ad hoc changes to, and easier to connect to the host-language routines that appear in the SMS interface.
3. Annotate nodes with their images under particular relations and, where applicable, use this readily available information rather than performing a lookup in a possibly large extension. For example, for the relation `dp is_effective_over r`, one could annotate each `dp` node with a list of the `r` nodes that it is effective over, and vice versa.
4. Transform rules and investigators to remove redundant calculation. For instance, consider the following investigator:

```
TO GET ALL {dp} SUCH THAT dp is_effective_over r,
  CONFIRM THAT r is_a_Block
  GET {ll} SUCH THAT ll is_the_LabelList_of r,
  GET ALL {dp} SUCH THAT dp is_a_Label_in ll
  PROJECT AWAY {ll}
  OR
  CONFIRM THAT r is_a_Block,
  GET {cdl} SUCH THAT cdl is_the_ConstantDefinitionList_of r,
  GET ALL {cd} SUCH THAT cd is_a_ConstantDefinition_in cdl,
  GET {dp} SUCH THAT dp is_the_Lhs_of cd,
  PROJECT AWAY {cd, cdl}
  OR
  GET ALL {etd} SUCH THAT r is_the_Block_closest_containing etd,
  CONFIRM THAT etd is_a_EnumeratedTypeDefiner,
  GET {idl} SUCH THAT idl is_the_IdentifierList_of etd,
  GET ALL {dp} SUCH THAT dp is_an_Identifier_in idl
(FORGET)
.
```

In all three conjunctions, r is required to be a Block, and it will be tested repeatedly for this property. A more efficient investigator would factor out the common test, as follows:

```
TO GET ALL {dp} SUCH THAT dp is_effective_over r,
  CONFIRM THAT r is_a_Block,
  (
    GET {ll} SUCH THAT ll is_the_LabelList_of r,
    GET ALL {dp} SUCH THAT dp is_a_Label_in ll
    PROJECT AWAY {ll}
    OR
    GET {cdl} SUCH THAT cdl is_the_ConstantDefinitionList_of r,
    GET ALL {cd} SUCH THAT cd is_a_ConstantDefinition_in cdl,
    GET {dp} SUCH THAT dp is_the_Lhs_of cd,
    PROJECT AWAY {cd, cdl}
    OR
    GET ALL {etd} SUCH THAT r is_the_Block_closest_containing etd,
      CONFIRM THAT etd is_a_EnumeratedTypeDefiner,
    GET {idl} SUCH THAT idl is_the_IdentifierList_of etd,
    GET ALL {dp} SUCH THAT dp is_an_Identifier_in idl
  )
(FORGET)
.
```

5. Update extensions incrementally. Even within a single call to FindAllExtensions, the extensions of recursive relations may be calculated many times, as (typically small) changes in extensions propagate their way through the cluster. It would save significant amounts of time if the interpreter could calculate the new extension simply by adding or subtracting the affected instances from the old extension, rather than recalculating the extension from scratch. However, I suspect this is a difficult problem.
6. Transform the NURN grammar to an equivalent attribute grammar and then implement that, using any of the many techniques available.

To make a NURN SMS really useful for interactive environments, it must be able to cope with changing objects. That is, it should be able to maintain the extensions of the relations as objects are created, destroyed, and edited. Incremental update algorithms would be really useful here as well.

Occasionally, a user of a NURN SMS might wish for a relation in addition to those of the NURN grammar for the target-language. Such a relation would not be necessary to define the target-language, but would be a convenience for the SMS user. We have already seen in Section 4.6 how the relation `is_a_possible_call_to` would be useful. As another example, consider a relation which identifies unused declarations:

```
RELATION DEF d is_a_unused_declaration :-
 d is_a_declaration_point,
 NOT (a is_a_applied_occurrence_of d)
.
```

It should be possible to augment the "necessary" relations with these user-defined "convenience" relations.

5.2.3 Open questions

How should a NURN grammar handle fragments? For example, for a programming language, how should we deal with the correctness of objects that do not occur in a program? Such objects are common in metaprogramming systems, but language descriptions are not usually concerned with any context less than a program, so they give us no direction in this matter. To make informed judgements, we need experience writing NURN grammars and using the resulting SMSs.

It would be interesting to see an equivalence transformation established from NURN grammars to attribute grammars.

Ginger's interpreter uses a successive-approximation technique to calculate the extensions of relations in a recursive cluster (see Section 4.2). What are the minimal conditions necessary to ensure that the approximations converge to a solution?

Section 4.4 sets out four different schemes for generating the investigative routines of a NURN SMS. Deciding which of these schemes are preferable will also require some practical experience.

APPENDIX 1: A GRAMPS-STYLE GRAMMAR FOR RELATION RULES AND INVESTIGATORS

This appendix presents a GRAMPS-style grammar for NURN's relation rules and the investigators generated by Ginger. A complete NURN grammar also includes GRAMPS-style rules; see [Ter87] for a grammar for these. There are a few differences between the syntax presented there and the syntax used in this thesis; the major differences have been discussed in Subsection 2.1.4.

```
GRAMMAR NURN IS

LIST Rule_List OF Rule SEPARATED_BY /

ALTERNATE CLOSED Rule IS Definition | Declaration | Subdefinition

CONSTRUCT Definition IS
  "RELATION" "DEF" <Head:Declarator> ":-" / <Body:Disjunction> "."

CONSTRUCT Declaration IS
  "RELATION" "DECL" <:Declarator> "."

  CONSTRUCT Declarator IS
    [ <:Kind_Indicator> ] <:Simple_Primary>
    [ _ ":" "FUNCTIONAL" "ON" <FunctionalBoundVarSets:VarSet_List> ]
    [ "WITH" <Profiles:Profile_List> ]

  ALTERNATE CLOSED Kind_Indicator IS
   KI_Normal | KI_Violation | KI_Maker | KI_External | KI_Derived |
   KI_Primitive

    CONSTRUCT KI_Normal IS "NORMAL"

    CONSTRUCT KI_Violation IS "VIOLATION"

    CONSTRUCT KI_Maker IS "MAKER"

    CONSTRUCT KI_External IS "EXTERNAL"

    CONSTRUCT KI_Derived IS "DERIVED"

    CONSTRUCT KI_Primitive IS "PRIMITIVE"

  LIST VarSet_List OF VarSet SEPARATED_BY _ ","

    CONSTRUCT VarSet IS "{" _ <:Var_List> _ "}"

      LIST Var_List OF Var SEPARATED_BY _ ","

  LIST Profile_List OF Profile SEPARATED_BY _ "," %
```

```
    CONSTRUCT Profile IS "{" _ <:ClassConstraint_List> _ "}"

      LIST ClassConstraint_List OF ClassConstraint SEPARATED_BY _ ","

        CONSTRUCT ClassConstraint IS <:Var> _ ":" _ <ClassName:Id>

CONSTRUCT Subdefinition IS
  "RELATION" "SUBDEF" <Head:Simple_Primaries> ":-" / <Body:Disjunction> "."

  LIST Simple_Primaries OF Simple_Primary SEPARATED_BY _ "," %

LIST Disjunction OF Conjunction SEPARATED_BY % "OR" %

LIST Conjunction OF Primary SEPARATED_BY _ "," %

ALTERNATE OPEN Primary IS Simple_Primary | Structured_Primary

ALTERNATE CLOSED Simple_Primary IS Unary | Binary | Ternary | Nary

  CONSTRUCT Unary IS <:Arg> <Relation:Id>

  CONSTRUCT Binary IS
    <Left_Arg:Arg> <Relation:Id> <Right_Arg:Arg>

  CONSTRUCT Ternary IS
    <Left_Arg:Arg> <Rel_Left:Id> <Centre_Arg:Arg> <Rel_Right:Id>
    <Right_Arg:Arg>

  CONSTRUCT Nary IS <Relation:Id> "(" <Args:Arg_List> ")"

    LIST Arg_List OF Arg SEPARATED_BY _ ","

  ALTERNATE OPEN Arg IS Var | Literal

  ALTERNATE OPEN Var IS Id | Integer_Id | String_Id

    LEXEME Id IS #id

    LEXEME Integer_Id IS '#' #id

    LEXEME String_Id IS '$' #id

    SUBLEXEME #id IS #Letter { #Letter | #Digit | '_' }

    CHARACTER-SET #Letter IS
      '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'
      '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'

    CHARACTER-SET #Digit IS '0' '1' '2' '3' '4' '5' '6' '7' '8' '9'

  ALTERNATE CLOSED Literal IS Integer_Literal | String_Literal
```

```
    LEXEME Integer_Literal IS #Digit { #Digit }

    LEXEME String_Literal IS '"' { #Letter | #Digit | '_' } '"'

ALTERNATE OPEN Structured_Primary IS Group | Negation | If | Maker

  CONSTRUCT Group IS "(" <Body:Disjunction> ")"

  CONSTRUCT Negation IS
    "NOT" <Body:Group>

  CONSTRUCT If IS
    "IF" <Condition:Group> "THEN"
      % <Consequent:Conjunction>
    "ELSE"
      % <Alternate:Conjunction>
    "END"

  LIST Id_List OF Id SEPARATED_BY _ ","

  CONSTRUCT Maker IS <:Id> ":=" <:Function_Call>

    CONSTRUCT Function_Call IS
      <Function_Name:Id> % "(" <Arguments:Expr_List> ")"

    LIST Expr_List OF Expr SEPARATED_BY _ "," %

    ALTERNATE OPEN Expr IS Function_Call | Arg


LIST Investigator_List OF Investigator SEPARATED_BY /

CONSTRUCT Investigator IS
  "TO" <:I_Simple>  _ "," /
    <Body:Investigator_Body> /
    "(" _ <:MemoStrategy> _ ")" /
  "."

ALTERNATE CLOSED MemoStrategy IS SaveWhole | Accumulate | Forget

  CONSTRUCT SaveWhole IS "SAVE" "WHOLE"

  CONSTRUCT Accumulate IS "ACCUMULATE"

  CONSTRUCT Forget IS "FORGET"

ALTERNATE OPEN Investigator_Body IS
  Pending_Body | Primitive_Body | External_Body | Copier | Selector |
  I_Disjunction

  CONSTRUCT Pending_Body IS "PENDING"
  CONSTRUCT Primitive_Body IS "PRIMITIVE"

  CONSTRUCT External_Body IS "EXTERNAL"

  CONSTRUCT Copier IS "[" <:I_Simple> "]" _ "," "COPY" "ACCUMULATION"

  CONSTRUCT Selector IS
    "[" <:I_Simple> "]" _ "," "THEN" "SELECT" "WITH" <:VarSet>

LIST I_Disjunction OF I_Conjunction SEPARATED_BY % "OR" %

LIST I_Conjunction OF I_Primary SEPARATED_BY _ "," %

ALTERNATE OPEN I_Primary IS I_Simple | I_Structured | Projector | SavePoint

  ALTERNATE CLOSED I_Simple IS Confirmer | Generator

    CONSTRUCT Confirmer IS "CONFIRM" "THAT" <:Simple_Primary>

    CONSTRUCT Generator IS
      "GET" [ <:NFI> ] <GenVars:VarSet> "SUCH" "THAT" <:Simple_Primary>

      CONSTRUCT NFI IS "ALL"

  ALTERNATE OPEN I_Structured IS I_Group | I_Negation | I_If | Maker

    CONSTRUCT I_Group IS "(" <Body:I_Disjunction> ")"

    CONSTRUCT I_Negation IS "ATTEMPT" "TO" <Body:I_Group> "BUT" "FAIL"

    CONSTRUCT I_If IS
      "IF" "YOU" "CAN" <Condition:I_Group> "THEN"
        % <Consequent:I_Conjunction>
      "ELSE"
        % <Alternate:I_Conjunction>
      "END"

  CONSTRUCT Projector IS "PROJECT" "AWAY" <:VarSet>

  CONSTRUCT SavePoint IS "SAVEPOINT"

LIST BasesMap OF BaseMap SEPARATED_BY _ ","

  CONSTRUCT BaseMap IS <:VarSet> _ ":" _ <:Id>
```

APPENDIX 2: A NURN GRAMMAR FOR PASCAL

In this appendix, the context-dependent syntax of Standard Pascal is specified in a NURN grammar.

The Pascal Standard [ANSI83] is not concerned with specifying (just) a context-dependent language, so it intermixes context-free, context-dependent, implementation-dependent, and data-dependent restrictions that a Pascal processor should enforce. Therefore, the term "the context-dependent syntax of Pascal" needs clarification. Section 5.1(e) of the Pascal Standard states (in part) that "[a Pascal processor shall] be able to determine whether or not a program violates any requirement of this standard, where such a violation is not designated an error, and report the result of this determination to the user of the processor". The standard does not have a term for "a violation that is not designated an error," but I choose to call it a *static violation*, because a note on the definition of "error" (in Section 3.1 of the Standard) indicates that these violations can be detected statically, i.e., by examining just the text of the program, without "knowledge of the data read by the program or the implementation definition of implementation-defined features." Given this definition, the following NURN grammar specifies the set of Pascal programs without static violations.

```
GRAMMAR Pasc IS

CASELESS-KEYWORDS

(*
Single-line comments beginning with "S" or "/S"
 indicate the start or end (respectively)
 of the corresponding section in the Pascal Standard.
*)

(* General violation. *)
RELATION DECL VIOLATION x is_not_allowed_in_this_context .

(*S  6.1: LEXICAL TOKENS. *)

(*S  6.1.1: General. *)

CHARACTER-SET #letter IS
 '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' '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'

CHARACTER-SET #digit IS '0' '1' '2' '3' '4' '5' '6' '7' '8' '9'

(*/S 6.1.1: General. *)

(*S  6.1.2: Special-Symbols. *)
```

```
(*/S 6.1.2: Special-Symbols. *)

(*S  6.1.3: Identifiers. *)

LEXEME Identifier IS #letter { #letter | #digit }

RELATION SUBDEF
  $s is_the_spelling_of id
:-
  id is_a_Identifier,
  $sid is_the_character_sequence_of id,
  $s is_the_lower_case_translation_of $sid
.
RELATION DECL
  EXTERNAL $l is_the_lower_case_translation_of $s: FUNCTIONAL ON {$s}
.

(*/S 6.1.3: Identifiers. *)

(*S  6.1.4: Directives. *)

CONSTRUCT Directive IS <:Identifier>

RELATION SUBDEF
  $s is_the_spelling_of d
:-
  d is_a_Directive,
  id is_the_Identifier_of d,
  $sid is_the_character_sequence_of id,
  $s is_the_lower_case_translation_of $sid
.

(*/S 6.1.4: Directives. *)

(*S  6.1.5: Numbers. *)

LEXEME UnsignedIntegerLiteral IS #digit_sequence
(*
An UnsignedIntegerLiteral shall denote in decimal notation
 a value of integer-type.
*)
RELATION SUBDEF
  t is_the_type_of expr
:-
  expr is_a_UnsignedIntegerLiteral,
  t is_the_integer_type
.
(*
The value denoted by an UnsignedIntegerLiteral
 must be in the closed interval 0 to MAXINT.
*)
RELATION DEF
  VIOLATION uil is_a_illegal_UnsignedIntegerLiteral
```

```
:-
  uil is_a_UnsignedIntegerLiteral,
  v is_the_value_denoted_by uil,
  zero is_the_integer_value_for_ordinal 0,
  maxint is_the_integer_value_maxint,
  (zero is_greater_than v OR v is_greater_than maxint)
.

LEXEME UnsignedRealLiteral IS
  #digit_sequence
  ( '.' #digit_sequence   #scale_factor |
    '.' #digit_sequence | #scale_factor )

SUBLEXEME #scale_factor IS ('e'|'E') [ '+' | '-' ] #digit_sequence

SUBLEXEME #digit_sequence IS #digit { #digit }

(*
An UnsignedRealLiteral shall denote in decimal notation
 a value of real-type.
The letter 'E' preceding a scale Factor shall mean
 'times ten to the power of'.
*)
RELATION SUBDEF
  t is_the_type_of expr
:-
  expr is_a_UnsignedRealLiteral,
  t is_the_real_type
.

(*/S 6.1.5: Numbers. *)

(*S  6.1.6: Labels. *)

CONSTRUCT Label IS <:UnsignedIntegerLiteral>

(*
The apparent integral value of a Label
 shall be in the closed interval 0 to 9999.
(Context-free takes care of lower bound.)
*)
RELATION DEF
  VIOLATION lab is_a_illegal_Label
:-
  lab is_a_Label,
  ui is_the_UnsignedIntegerLiteral_of lab,
  v is_the_value_denoted_by ui,
  v is_greater_than max,
  max is_the_integer_value_for_ordinal 9999
.

RELATION SUBDEF
  $s is_the_spelling_of lab
```

```
:-
  lab is_a_Label,
  ui is_the_UnsignedIntegerLiteral_of lab,
  $sui is_the_character_sequence_of ui,
  $s is_the_unzeroed_translation_of $sui
.
RELATION DECL
  EXTERNAL $u is_the_unzeroed_translation_of $z: FUNCTIONAL ON {$z}
.

(*/S 6.1.6: Labels. *)

(*S  6.1.7: Character-Strings. *)

LEXEME CharacterString IS ''' #string_element { #string_element } '''

SUBLEXEME #string_element IS #apostrophe_image | #string_character

SUBLEXEME #apostrophe_image IS ''' '''

CHARACTER-SET #string_character IS
  ' ' '!' '"' '#' '$' '%' '&' '(' ')' '*' '+' ',' '-' '.' '/' '0' '1'
  '2' '3' '4' '5' '6' '7' '8' '9' ':' ';' '<' '=' '>' '?' '@' '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' '[' '\' ']'     '_'     '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' '{' '|' '}'

(*
A CharacterString containing a single string-element
 shall denote a value of the required char-type (see 6.4.2.2).
A CharacterString containing more than one string-element
 shall denote a value of a string-type (see 6.4.3.2)
  with the same number of components
  as the CharacterString contains string-elements.
  ("packed array [1..<#n>] of char")
*)
RELATION DECL
  EXTERNAL #n is_the_number_of_string_elements_in $s: FUNCTIONAL ON {$s}
.

RELATION SUBDEF
  t is_the_type_of cs
:-
  cs is_a_CharacterString,
  $s is_the_character_sequence_of cs,
  IF (1 is_the_number_of_string_elements_in $s) THEN
    t is_the_char_type
  ELSE
    t is_the_string_type_corresponding_to cs
  END
.
RELATION DEF
```

```
  MAKER t is_the_string_type_corresponding_to cs: FUNCTIONAL ON {t},{cs}
:-
  cs is_a_CharacterString,
  $s is_the_character_sequence_of cs,
  #n is_the_number_of_string_elements_in $s,
  $z IS_THE_BASE_TEN_REP_OF #n,
  t :=
    make_ArrayTypeDefiner
      ( make_TypeDenoterList
          ( make_SubrangeTypeDefiner
              ( make_UnsignedIntegerLiteral("1"),
                make_UnsignedIntegerLiteral($z)
              )
          ),
        make_Identifier("char")
      ),
  p := make_PackedStructuredTypeDefiner(t)
.

(*/S 6.1.7: Character-Strings. *)

(*S  6.1.8: Token-Separators. *)

COMMENT-DELIMITERS "{" "}"

(*
There shall be at least one separator (comment,space,newline)
 between any pair of consecutive tokens made up of
 Identifiers, keywords, or UnsignedLiterals.
The one deviant case that will not be rejected for other reasons
 is an UnsignedLiteral followed by a keyword.
Examples:
 10div 2
 begin x:=10.3end
*)
LEXEME DeviantLexeme IS
  #digit_sequence [ '.' #digit_sequence ] [ #scale_factor ]
  #letter { #letter }

(*/S 6.1.8: Token-Separators. *)

(*S  6.1.9: Lexical Alternatives. *)

COMMENT-DELIMITERS "(*" "*)"
COMMENT-DELIMITERS "(*"  "}"
COMMENT-DELIMITERS "{"  "*)"

(*/S 6.1.9: Lexical Alternatives. *)

(*/S 6.1: LEXICAL TOKENS. *)

(*S 6.2: BLOCKS, SCOPE AND ACTIVATIONS. *)
```

```
(*S 6.2.1: Block. *)

CONSTRUCT Block IS
  [ "label" <:LabelList> ";" ]
  [ "const" <:ConstantDefinitionList> ]
  [ "type" <:TypeDefinitionList> ]
  [ "var" <:VariableDeclarationList> ]
  <:RoutineDeclarationList>
  "begin"
    <:StatementSequence>
  "end"

LIST NONEMPTY LabelList OF Label SEPARATED_BY _ ","

LIST NONEMPTY ConstantDefinitionList OF ConstantDefinition

LIST NONEMPTY TypeDefinitionList OF TypeDefinition

LIST NONEMPTY VariableDeclarationList OF VariableDeclaration

LIST RoutineDeclarationList OF RoutineDeclaration

ALTERNATE CLOSED RoutineDeclaration IS
  ProcedureDeclaration | FunctionDeclaration

(*
Utility:
The Block closest-containing x
 is that Block which contains x
 but does not contain another Block containing that x.
*)
RELATION DEF
  b is_the_Block_closest_containing x: FUNCTIONAL ON {x}
:-
  p is_the_parent_of x,
  IF (p is_a_Block) THEN
    b is p
  ELSE
    b is_the_Block_closest_containing p
  END
.

(*
Label Declarations:
*)

(*
A Label in the LabelList of a Block
 is a declaration-point
 (as a Label)
 effective over the region that is the Block.
*)
RELATION DECL lab declares_a_label .
RELATION SUBDEF
  lab is_a_declaration_point,
  lab declares_a_label,
  lab is_effective_over b
:-
  lab is_a_Label_in ll,
  ll is_the_LabelList_of b,
  b is_a_Block
.

(*
The Label of a LabelledStatement
 is the 'site' for the defining occurrence of that Label.
*)
RELATION DEF
  slab is_the_site_for lab: FUNCTIONAL ON {slab},{lab}
:-
  slab is_the_Label_of s, s is_a_LabelledStatement,
  lab is_the_defining_occurrence_of slab,
  lab declares_a_label
.

(*
A Label must have exactly one site,
 and the Block over which the Label is declared must be
  the Block closest-containing the site.
*)
RELATION DEF
  VIOLATION lab declares_a_label_with_no_site
:-
  lab declares_a_label,
  NOT (slab is_the_site_for lab)
.
RELATION DEF
  VIOLATION slab_2 is_a_duplicate_label_site
:-
  lab declares_a_label,
  slab_1 is_the_site_for lab,
  slab_2 is_the_site_for lab,
  NOT (slab_1 is slab_2)
.
RELATION SUBDEF
  slab is_not_allowed_in_this_context
:-
  lab declares_a_label,
  slab is_the_site_for lab,
  lab is_effective_over b,
  NOT (b is_the_Block_closest_containing slab)
.

(*/S 6.2.1: Block. *)

(*S  6.2.2: Scope. *)
```

```
RELATION DECL dp is_a_declaration_point .

RELATION DECL dp is_effective_over region .

(*
6.2.2.1:
Each Identifier or Label contained by the Block of the Program
 (other than the Identifier of a Directive)
 must have a defining-occurrence.
*)
RELATION DEF
  x is_a_scoped_entity
:-
  x is_a_Label OR
  ( x is_a_Identifier,
    NOT (x is_the_Identifier_of dir, dir is_a_Directive) )
.
RELATION DECL VIOLATION x is_undefined .
RELATION SUBDEF
  x is_undefined
:-
  x is_a_scoped_entity,
  b is_the_Block_of p, p is_a_Program,
  b contains x,
  NOT (d is_the_defining_occurrence_of x)
.

RELATION DECL $s is_the_spelling_of x: FUNCTIONAL ON {x} .

(*
6.2.2.{2,3,4,5,6}
*)
RELATION DEF
  d defines_the_spelling $s if_it_occurs_at x: FUNCTIONAL ON {$s,x}, {d,x}
:-
  IF (d_2 defines_the_spelling $s over x) THEN
    d is d_2
  ELSE
    IF (x is_a_orphan) THEN
      NOT (x is_the_required_Block),
      p is_the_required_Block
    ELSE
      p is_the_parent_of x
    END,
    d defines_the_spelling $s if_it_occurs_at p
  END
.
(*
for efficiency?:
*)
RELATION DEF
  d defines_the_spelling $s over x: FUNCTIONAL ON {$s,x}, {d,x}
```

```
:-
  d is_effective_over x, $s is_the_spelling_of d
.

(*
6.2.2.7
*)
RELATION DECL VIOLATION d_2 is_a_conflicting_declaration_point .
RELATION SUBDEF
  d_2 is_a_conflicting_declaration_point
:-
  d_1 defines_the_spelling $s over x,
  d_2 defines_the_spelling $s over x,
  NOT (d_1 is d_2)
.

(*
6.2.2.8
*)
RELATION DEF
  d is_the_defining_occurrence_of e: FUNCTIONAL ON {e}
:-
  e is_a_scoped_entity,
  $se is_the_spelling_of e,
  d defines_the_spelling $se if_it_occurs_at e
.
RELATION DEF
  x is_a_applied_occurrence_of d: FUNCTIONAL ON {x}
:-
  d is_the_defining_occurrence_of x,
  NOT (d is x)
.

(*
6.2.2.9:
The defining occurrence of an Identifier or Label
 must precede all applied occurrences of that Identifier or Label.
*)
RELATION DECL VIOLATION x cannot_be_used_yet .
RELATION SUBDEF
  x cannot_be_used_yet
:-
  x is_a_applied_occurrence_of d,
  x is_before d,
  NOT (x is_a_use_exception)
.

(*
6.2.2.10:
Identifiers that denote required types and routines
 shall be "used as if" their defining-occurrences
 have a region enclosing the Program.
*)
```

```
RELATION DEF
  MAKER b is_the_required_Block: FUNCTIONAL ON {}
:-
  maxint is_the_integer_value_maxint,
  the_boolean_type is_the_boolean_type,
  the_integer_type is_the_integer_type,
  the_real_type is_the_real_type,
  the_char_type is_the_char_type,
  the_textfile_type is_the_textfile_type,

  fpl_OneFile :=
    make_FormalParameterList
    ( make_VariableParameterSection
      ( make_IdentifierList(make_Identifier("f")),
        make_Identifier("aFile")
      )
    ),
  fpl_OneNumeric :=
    make_FormalParameterList
    ( make_ValueParameterSection
      ( make_IdentifierList(make_Identifier("x")),
        make_Identifier("Numeric")
      )
    ),
  fpl_OneReal :=
    make_FormalParameterList
    ( make_ValueParameterSection
      ( make_IdentifierList(make_Identifier("x")),
        make_Identifier("real")
      )
    ),
  fpl_OneOrdinal :=
    make_FormalParameterList
    ( make_ValueParameterSection
      ( make_IdentifierList(make_Identifier("x")),
        make_Identifier("Ordinal")
      )
    ),
  fpl_OneInteger :=
    make_FormalParameterList
    ( make_ValueParameterSection
      ( make_IdentifierList(make_Identifier("x")),
        make_Identifier("integer")
      )
    ),
  pre is_the_directive_predefined,
  b :=
    make_Block
    ( make_Empty (),

      make_ConstantDefinitionList
      ( make_ConstantDefinition ( make_Identifier("maxint"), maxint ) ),
make_TypeDefinitionList
( make_TypeDefinition ( make_Identifier("boolean"), the_boolean_type ),
  make_TypeDefinition ( make_Identifier("integer"), the_integer_type ),
  make_TypeDefinition ( make_Identifier("real"),    the_real_type ),
  make_TypeDefinition ( make_Identifier("char"),    the_char_type ),
  make_TypeDefinition ( make_Identifier("text"),    the_textfile_type )
),

make_Empty (),

make_RoutineDeclarationList
(
  make_ProcedureDeclaration
  ( make_Identifier("rewrite"), fpl_OneFile, pre ),
  make_ProcedureDeclaration
  ( make_Identifier("put"),     fpl_OneFile, pre ),
  make_ProcedureDeclaration
  ( make_Identifier("reset"),   fpl_OneFile, pre ),
  make_ProcedureDeclaration
  ( make_Identifier("get"),     fpl_OneFile, pre ),

  make_ProcedureDeclaration
  ( make_Identifier("read"),    make_Empty(), pre ),
  make_ProcedureDeclaration
  ( make_Identifier("readln"),  make_Empty(), pre ),
  make_ProcedureDeclaration
  ( make_Identifier("write"),   make_Empty(), pre ),
  make_ProcedureDeclaration
  ( make_Identifier("writeln"), make_Empty(), pre ),
  make_ProcedureDeclaration
  ( make_Identifier("page"),    make_Empty(), pre ),

  make_ProcedureDeclaration
  ( make_Identifier("new"),     make_Empty(), pre ),
  make_ProcedureDeclaration
  ( make_Identifier("dispose"), make_Empty(), pre ),

  make_ProcedureDeclaration
  ( make_Identifier("pack"),
    make_FormalParameterList
    ( make_VariableParameterSection
      ( make_IdentifierList(make_Identifier("a")),
        make_Identifier("UnpackedArray")
      ),
      make_ValueParameterSection
      ( make_IdentifierList(make_Identifier("i")),
        make_Identifier("Ordinal")
      ),
      make_VariableParameterSection
      ( make_IdentifierList(make_Identifier("z")),
        make_Identifier("PackedArray")
      )
    ),
  pre ),
make_ProcedureDeclaration
( make_Identifier("unpack"),
  make_FormalParameterList
  ( make_VariableParameterSection
    ( make_IdentifierList(make_Identifier("z")),
      make_Identifier("PackedArray")
    ),
    make_VariableParameterSection
    ( make_IdentifierList(make_Identifier("a")),
      make_Identifier("UnpackedArray")
    ),
    make_ValueParameterSection
    ( make_IdentifierList(make_Identifier("i")),
      make_Identifier("Ordinal")
    )
  ),
  pre ),

make_FunctionDef
( make_Identifier("abs"),
  fpl_OneNumeric,
  make_Identifier("Numeric"),
  pre ),
make_FunctionDef
( make_Identifier("sqr"),
  fpl_OneNumeric,
  make_Identifier("Numeric"),
  pre ),
make_FunctionDef
( make_Identifier("sin"),
  fpl_OneReal,
  make_Identifier("real"),
  pre ),
make_FunctionDef
( make_Identifier("cos"),
  fpl_OneReal,
  make_Identifier("real"),
  pre ),
make_FunctionDef
( make_Identifier("exp"),
  fpl_OneReal,
  make_Identifier("real"),
  pre ),
make_FunctionDef
( make_Identifier("ln"),
  fpl_OneReal,
  make_Identifier("real"),
  pre ),
make_FunctionDef
( make_Identifier("sqrt"),
  fpl_OneReal,
  make_Identifier("real"),
  pre ),
make_FunctionDef
( make_Identifier("arctan"),
  fpl_OneReal,
  make_Identifier("real"),
  pre ),

make_FunctionDef
( make_Identifier("trunc"),
  fpl_OneReal,
  make_Identifier("integer"),
  pre ),
make_FunctionDef
( make_Identifier("round"),
  fpl_OneReal,
  make_Identifier("integer"),
  pre ),

make_FunctionDef
( make_Identifier("ord"),
  fpl_OneOrdinal,
  make_Identifier("integer"),
  pre ),
make_FunctionDef
( make_Identifier("chr"),
  fpl_OneInteger,
  make_Identifier("char"),
  pre ),
make_FunctionDef
( make_Identifier("succ"),
  fpl_OneOrdinal,
  make_Identifier("Ordinal"),
  pre ),
make_FunctionDef
( make_Identifier("pred"),
  fpl_OneOrdinal,
  make_Identifier("Ordinal"),
  pre ),

make_FunctionDef
( make_Identifier("odd"),
  fpl_OneInteger,
  make_Identifier("boolean"),
  pre ),
make_FunctionDef
( make_Identifier("eof"),
  make_Empty(),
  make_Identifier("boolean"),
  pre ),
make_FunctionDef
( make_Identifier("eoln"),
  make_Empty(),
  make_Identifier("boolean"),
            pre )
        ),

        make_StatementSequence ( make_EmptyStatement() )
    )
.

RELATION DEF
  MAKER dir is_the_directive_predefined: FUNCTIONAL ON {}
:-
  dir := make_Directive ( make_Identifier("PREDEFINED") )
.
RELATION DEF
  rd is_the_required_routine_named $s: FUNCTIONAL ON {rd}, {$s}
:-
  $s is_the_spelling_of id,
  id is_the_Name_of rd,
  rd is_a_RoutineDeclaration_in rdl,
  rdl is_the_RoutineDeclarationList_of b,
  b is_the_required_Block
.

(*
6.2.2.11:
An applied occurrence of an Identifier or Label
 denotes whatever its defining occurrence denotes.
*)

(*/S 6.2.2: Scope. *)

(*S  6.2.3: Activations. *)

(*/S 6.2.3: Activations. *)

(*/S 6.2: BLOCKS, SCOPE AND ACTIVATIONS. *)

(*S  6.3: CONSTANT-DEFINITIONS. *)

RELATION DECL id declares_a_constant_identifier .
RELATION DECL VIOLATION x should_be_a_constant_identifier_but_isnt .
RELATION DEF
  id is_a_constant_identifier
:-
  d is_the_defining_occurrence_of id,
  d declares_a_constant_identifier
.

CONSTRUCT ConstantDefinition IS <Lhs:Identifier> "=" <Rhs:Constant> ";"

(*
The Lhs of
 a ConstantDefinition in
 the ConstantDefinitionList of
 a Block
 is a declaration-point
 as a constant-identifier
 over the region that is the Block.
*)
RELATION SUBDEF
  id is_a_declaration_point,
  id declares_a_constant_identifier
:-
  id is_the_Lhs_of cd,
  cd is_a_ConstantDefinition
.
RELATION SUBDEF
  id is_effective_over b
:-
  id is_the_Lhs_of cd,
  cd is_a_ConstantDefinition_in cdl,
  cdl is_the_ConstantDefinitionList_of b,
  b is_a_Block
.

(*
The Rhs of a ConstantDefinition
 must not contain an applied-occurrence of
 the Lhs of the ConstantDefinition.
*)
RELATION SUBDEF
  ao cannot_be_used_yet
:-
  ao is_a_applied_occurrence_of do,
  do is_the_Lhs_of cd,
  cd is_a_ConstantDefinition,
  c is_the_Rhs_of cd,
  c contains ao
.

RELATION DECL val is_the_value_denoted_by c: FUNCTIONAL ON {c} .

(*
The Lhs of a ConstantDefinition
 denotes the value denoted by
 (and possesses the type possessed by)
 the Rhs of the ConstantDefinition.
*)
RELATION SUBDEF
  val is_the_value_denoted_by id
:-
  cd is_a_ConstantDefinition,
  id is_the_Lhs_of cd,
  c is_the_Rhs_of cd,
  val is_the_value_denoted_by c
.
RELATION SUBDEF
```

```
  t is_the_type_of id
:-
  cd is_a_ConstantDefinition,
  id is_the_Lhs_of cd,
  c is_the_Rhs_of cd,
  t is_the_type_of c
.
(*
Each applied occurrence of a constant-identifier
 denotes the value denoted by the constant-identifier.
*)
RELATION SUBDEF
  val is_the_value_denoted_by id
:-
  id is_a_applied_occurrence_of d,
  val is_the_value_denoted_by d
.

ALTERNATE OPEN Constant IS
  SignedIntegerLiteral | UnsignedIntegerLiteral |
  SignedRealLiteral | UnsignedRealLiteral |
  SignedIdentifier | Identifier |
  CharacterString

CONSTRUCT SignedIntegerLiteral IS <:Sign> _ <:UnsignedIntegerLiteral>
CONSTRUCT SignedRealLiteral    IS <:Sign> _ <:UnsignedRealLiteral>
CONSTRUCT SignedIdentifier     IS <:Sign> _ <:Identifier>

ALTERNATE OPEN Sign IS PlusOp | MinusOp

(*
Anything in a Constant context must denote a value.
This is automatic for the numeric and character literals.
*)
RELATION SUBDEF
  id should_be_a_constant_identifier_but_isnt
:-
  id is_a_Identifier,
  id is_in_a_Constant_context,
  NOT (id is_a_constant_identifier)
OR
  id is_the_Identifier_of si, si is_a_SignedIdentifier,
  NOT (id is_a_constant_identifier)
.

(*
The Identifier of a SignedIdentifier
 must denote a value of real-type or of integer-type.
*)
RELATION SUBDEF
  id has_a_inappropriate_type
:-
  si is_a_SignedIdentifier,
  id is_the_Identifier_of si,
  t is_the_type_of id,
  NOT (t is_the_real_type OR t is_the_integer_type)
.
(*
The type of a signed-thing shall be
 the type of the unsigned-thing of the signed-thing.
*)
RELATION SUBDEF
  t is_the_type_of sx
:-
  (
    sx is_a_SignedIntegerLiteral, ux is_the_UnsignedIntegerLiteral_of sx
  OR
    sx is_a_SignedRealLiteral, ux is_the_UnsignedRealLiteral_of sx
  OR
    sx is_a_SignedIdentifier, ux is_the_Identifier_of sx
  ),
  t is_the_type_of ux
.

(*
The value denoted by a signed-thing
 is either the value denoted by the unsigned-thing of the signed-thing,
 or that value's sign-inverse,
according to whether the Sign of the signed-thing is a PlusOp or a MinusOp,
 respectively.
*)
RELATION SUBDEF
  vsx is_the_value_denoted_by sx
:-
  (
    sx is_a_SignedIntegerLiteral, ux is_the_UnsignedIntegerLiteral_of sx
  OR
    sx is_a_SignedRealLiteral, ux is_the_UnsignedRealLiteral_of sx
  OR
    sx is_a_SignedIdentifier, ux is_the_Identifier_of sx
  ),
  s is_the_Sign_of sx,
  vux is_the_value_denoted_by ux,
  (
    s is_a_PlusOp,  vsx is vux
  OR
    s is_a_MinusOp, vsx is_the_sign_inverse_of vux
  )
.

RELATION DECL v_1 is_the_sign_inverse_of v_2: FUNCTIONAL ON {v_1}, {v_2} .

(*/S 6.3: CONSTANT-DEFINITIONS. *)

(*S  6.4: TYPE-DEFINITIONS. *)
```

```
(*
Type shall be an attribute that is possessed
 by every value, variable, and expression.
*)
RELATION DECL t is_the_type_of x: FUNCTIONAL ON {x} .

RELATION DEF
  VIOLATION x should_have_a_type_but_doesnt
:-
  (
    x is_a_genuine_expression
  OR
    x denotes_a_variable,
    NOT
      (x is_a_formal_parameter_in fpl,
       fpl is_the_FormalParameterList_of rd,
       rd is_a_required_RoutineDeclaration
      )
  ),
  NOT (t is_the_type_of x)
.

RELATION DECL VIOLATION x has_a_inappropriate_type .

(*S  6.4.1: General. *)

RELATION DECL id declares_a_type_identifier .
RELATION DECL id is_a_type_identifier .

RELATION SUBDEF
  id is_a_type_identifier
:-
  d is_the_defining_occurrence_of id,
  d declares_a_type_identifier
.

CONSTRUCT TypeDefinition IS <Lhs:Identifier> "=" <Rhs:TypeDenoter> ";"

(*
The Lhs of a TypeDefinition
 in the TypeDefinitionList of a Block
 is a declaration-point
 effective over the region that is the Block.
!: Overlap
*)

RELATION SUBDEF
  id is_a_declaration_point,
  id declares_a_type_identifier
:-
  id is_the_Lhs_of tdef,
  tdef is_a_TypeDefinition
.
```

```
RELATION SUBDEF
  id is_effective_over b
:-
  id is_the_Lhs_of tdef,
  tdef is_a_TypeDefinition_in tdl,
  tdl is_the_TypeDefinitionList_of b,
  b is_a_Block
.

RELATION DECL t is_the_type_denoted_by x: FUNCTIONAL ON {x} .
RELATION DECL VIOLATION td denotes_a_inappropriate_type .

(*
The type denoted by the Lhs of a TypeDefinition
 is the type denoted by the Rhs of the TypeDefinition.
*)
RELATION SUBDEF
  t is_the_type_denoted_by id
:-
  tdef is_a_TypeDefinition,
  id is_the_Lhs_of tdef,
  td is_the_Rhs_of tdef,
  t is_the_type_denoted_by td
.

(*
The Rhs of a TypeDefinition
 must not contain an applied-occurrence of
 the Lhs of the TypeDefinition,
except for applied-occurrences
 in the DomainTypeIdentifier of a PointerTypeDefiner.
*)
RELATION SUBDEF
  ao cannot_be_used_yet
:-
  tdef is_a_TypeDefinition,
  id is_the_Lhs_of tdef,
  td is_the_Rhs_of tdef,
  ao is_a_applied_occurrence_of id,
  td contains ao,
  NOT (ao is_a_use_exception)
.

(*
Each applied occurrence of a type-identifier
 denotes the type denoted by the type-identifier.
*)
RELATION SUBDEF
  t is_the_type_denoted_by id
:-
  id is_a_Identifier,
  id is_a_applied_occurrence_of d,
  t is_the_type_denoted_by d
```

```
.

(*
The type of an applied occurrence
  (of a constant-identifier, variable-identifier, whatever)
 is the type of its defining occurrence.
*)
RELATION SUBDEF
  t is_the_type_of id
:-
  id is_a_Identifier,
  id is_a_applied_occurrence_of d,
  t is_the_type_of d
.

ALTERNATE OPEN TypeDenoter IS TypeIdentifier | TypeDefiner

ALTERNATE OPEN TypeIdentifier IS Identifier

(*
A TypeDenoter must denote a type.
(For TypeDefiners, this is automatic.)
Exceptions:
  a ParameterTypeIdentifier for a required routine,
  the ResultTypeIdentifier of a required routine.
*)
RELATION DEF
  VIOLATION td should_denote_a_type_but_doesnt
:-
  td is_a_Identifier,
  td is_in_a_TypeIdentifier_context,
  NOT
    (
      td is_the_ParameterTypeIdentifier_of fps,
      fps is_a_FormalParameterSection_in fpl,
      fpl is_the_FormalParameterList_of rd,
      rd is_a_required_RoutineDeclaration
    OR
      td is_the_ResultTypeIdentifier_of rd,
      rd is_a_required_RoutineDeclaration
    ),
  NOT (t is_the_type_denoted_by td)
.

RELATION DECL t is_a_type .

(*
A TypeDefiner
 (other than a PackedStructuredTypeDefiner or
  the BaseTypeDenoter of a canonical set type)
 shall be/represent a type.
*)
RELATION SUBDEF
```

```
  t is_a_type
:-
  t is_in_the_TypeDefiner_domain,
  NOT (t is_a_PackedStructuredTypeDefiner OR
       t is_the_BaseTypeDenoter_of st, st is_a_canonical_set_type)
.
(*
A type shall denote itself.
*)
RELATION SUBDEF
  t is_the_type_denoted_by t
:-
  t is_a_type
.

ALTERNATE OPEN TypeDefiner IS
  OrdinalTypeDefiner | StructuredTypeDefiner | PointerTypeDefiner
(*
"OPEN" because StructuredTypeDefiner is open.
*)

(*/S 6.4.1: General. *)

(*S  6.4.2: Simple-Types. *)

(*S  6.4.2.1: General. *)

RELATION DECL t is_a_ordinal_type .

ALTERNATE CLOSED OrdinalTypeDefiner IS
  EnumeratedTypeDefiner | SubrangeTypeDefiner

RELATION SUBDEF
  t is_a_ordinal_type
:-
  t is_a_OrdinalTypeDefiner
.

(*
An ordinal-type shall determine an finite set of values.
Each value of an ordinal-type shall have an integer ordinal number.
The values of an ordinal-type shall be ordered by their ordinal numbers.
*)
RELATION DECL lo is_the_smallest_value_of ot: FUNCTIONAL ON {ot} .
RELATION DECL hi is_the_largest_value_of  ot: FUNCTIONAL ON {ot} .
RELATION DECL
  #ord is_the_ordinal_number_of v in ot: FUNCTIONAL ON {v,ot}, {#ord,ot} .

RELATION DEF
  val is_a_value_of ot
:-
  ot is_a_ordinal_type,
  #ord_lo is_the_ordinal_number_of lo in ot, lo is_the_smallest_value_of ot,
  #ord_hi is_the_ordinal_number_of hi in ot, hi is_the_largest_value_of ot,
  #ord_val IS_IN_THE_RANGE_FROM #ord_lo UP_TO #ord_hi,
  #ord_val is_the_ordinal_number_of val in ot
.

RELATION DEF
  #n is_the_number_of_values_defined_by ot: FUNCTIONAL ON {ot}
:-
  ot is_a_ordinal_type,
  #ord_lo is_the_ordinal_number_of lo in ot, lo is_the_smallest_value_of ot,
  #ord_hi is_the_ordinal_number_of hi in ot, hi is_the_largest_value_of ot,
  #diff IS #ord_hi MINUS #ord_lo,
  #n IS #diff PLUS 1
.

RELATION DEF
  v_1 is_greater_than v_2
:-
  ot is_the_type_of v_1,
  ot is_the_type_of v_2,
  ot is_a_ordinal_type,
  #ord_1 is_the_ordinal_number_of v_1 in ot,
  #ord_2 is_the_ordinal_number_of v_2 in ot,
  #ord_1 IS_GREATER_THAN #ord_2
.

(*/S 6.4.2.1: General. *)

(*S  6.4.2.2: Required Simple-Types. *)

RELATION SUBDEF
  t is_a_ordinal_type
:-
  t is_the_integer_type
OR
  t is_the_boolean_type
OR
  t is_the_char_type
.

(* 1: The Integer-Type ------------------------- *)
RELATION DEF
  MAKER t is_the_integer_type: FUNCTIONAL ON {}
:-
  t := make_Identifier ( "TheIntegerType" )
.
RELATION SUBDEF
  t is_a_type_identifier,
  t is_a_type
:-
  t is_the_integer_type
.
RELATION DEF
```

```
  t is_a_integer_type
:-
  rt is_the_range_type_of t,
  rt is_the_integer_type
.

(*
(from 6.7.2.2:)
All integral values in the closed interval from -MAXINT to +MAXINT
 shall be values of the integer-type.
*)
RELATION SUBDEF
  lo is_the_smallest_value_of t
:-
  t is_the_integer_type,
  maxint is_the_integer_value_maxint,
  lo is_the_sign_inverse_of maxint
.
RELATION SUBDEF
  hi is_the_largest_value_of t
:-
  t is_the_integer_type,
  hi is_the_integer_value_maxint
.

RELATION SUBDEF
  #ord is_the_ordinal_number_of v in t
:-
  t is_the_integer_type,
  v is_the_integer_value_for_ordinal #ord
.
RELATION DEF
  MAKER sil is_the_integer_value_for_ordinal #ord:
    FUNCTIONAL ON {sil}, {#ord}
:-
  IF (0 IS_GREATER_THAN #ord) THEN
    sign := make_MinusOp (), #abs IS 0 MINUS #ord
  ELSE
    sign := make_PlusOp (),  #abs EQUALS #ord
  END,
  $a IS_THE_BASE_TEN_REP_OF #abs,
  sil := make_SignedIntegerLiteral ( sign, make_UnsignedIntegerLiteral($a) )
.
RELATION SUBDEF
  val is_the_value_denoted_by x
:-
  t is_the_integer_type,
  x is_a_UnsignedIntegerLiteral,
  $i is_the_character_sequence_of x,
  $i IS_THE_BASE_TEN_REP_OF #ord,
  #ord is_the_ordinal_number_of val in t
.
RELATION SUBDEF
```

```
  v_1 is_the_sign_inverse_of v_2
:-
  v_1 is_the_integer_value_for_ordinal #ord_1,
  v_2 is_the_integer_value_for_ordinal #ord_2,
  #ord_1 IS 0 MINUS #ord_2
.

(* 2: The Real-Type ---------------------------- *)
RELATION DEF
  MAKER t is_the_real_type: FUNCTIONAL ON {}
:-
  t := make_Identifier ( "TheRealType" )
.
RELATION SUBDEF
  t is_a_type_identifier,
  t is_a_type
:-
  t is_the_real_type
.

(* 3: The Boolean-Type ------------------------- *)
RELATION DEF
  MAKER t is_the_boolean_type: FUNCTIONAL ON {}
:-
  t :=
    make_EnumeratedTypeDefiner
      ( make_IdentifierList
          ( make_Identifier ( "FALSE" ),
            make_Identifier ( "TRUE" )
          )
      )
.
RELATION DEF
  t is_a_boolean_type
:-
  rt is_the_range_type_of t,
  rt is_the_boolean_type
.

(* 4: The Char-Type ---------------------------- *)
RELATION DEF
  MAKER t is_the_char_type: FUNCTIONAL ON {}
:-
  t := make_Identifier ( "TheCharType" )
.
RELATION SUBDEF
  t is_a_type_identifier,
  t is_a_type
:-
  t is_the_char_type
.
RELATION DEF
  t is_a_char_type
```

```
:-
  rt is_the_range_type_of t,
  rt is_the_char_type
.

(*
The ordinal number of the smallest value of the char-type shall be 0.
The rest are implementation-defined.
*)
RELATION SUBDEF
  lo is_the_smallest_value_of t
:-
  t is_the_char_type,
  32 is_the_ordinal_number_of lo in t
.
RELATION SUBDEF
  hi is_the_largest_value_of t
:-
  t is_the_char_type,
  126 is_the_ordinal_number_of hi in t
.
RELATION SUBDEF
  #ord is_the_ordinal_number_of v in t
:-
  t is_the_char_type,
  v is_the_char_value_for_ordinal #ord
.
RELATION DEF
  MAKER cs is_the_char_value_for_ordinal #ord: FUNCTIONAL ON {cs}, {#ord}
:-
  $c is_the_charseq_for_the_char_value_for_ordinal #ord,
  cs := make_CharacterString ( $c )
.
RELATION SUBDEF
  val is_the_value_denoted_by cs
:-
  t is_the_char_type,
  cs is_a_CharacterString,
  $c is_the_character_sequence_of cs,
  1 is_the_number_of_string_elements_in $c,
  $c is_the_charseq_for_the_char_value_for_ordinal #ord,
  #ord is_the_ordinal_number_of val in t
.
RELATION DECL
  EXTERNAL $c is_the_charseq_for_the_char_value_for_ordinal #ord:
    FUNCTIONAL ON {$c}, {#ord}
.

(*/S 6.4.2.2: Required Simple-Types. *)

(*S  6.4.2.3: Enumerated-Types. *)

CONSTRUCT EnumeratedTypeDefiner IS "(" <:IdentifierList> ")"
```

```
LIST NONEMPTY IdentifierList OF Identifier SEPARATED_BY _ ","

(*
An Identifier in the IdentifierList of an EnumeratedTypeDefiner
 is a declaration-point
 effective over
 the Block closest containing the EnumeratedTypeDefiner.
!: Overlap
*)
RELATION SUBDEF
  id is_a_declaration_point,
  id declares_a_constant_identifier,
  id is_the_value_denoted_by id,
  etd is_the_type_of id
:-
  id is_a_Identifier_in idl,
  idl is_the_IdentifierList_of etd,
  etd is_a_EnumeratedTypeDefiner
.
RELATION SUBDEF
  id is_effective_over b
:-
  id is_a_Identifier_in idl,
  idl is_the_IdentifierList_of etd,
  etd is_a_EnumeratedTypeDefiner,
  b is_the_Block_closest_containing etd
.


RELATION SUBDEF
  lo is_the_smallest_value_of etd
:-
  etd is_a_EnumeratedTypeDefiner,
  idl is_the_IdentifierList_of etd,
  lo is_the 1 th_Identifier_in idl
.
RELATION SUBDEF
  hi is_the_largest_value_of etd
:-
  etd is_a_EnumeratedTypeDefiner,
  idl is_the_IdentifierList_of etd,
  #n is_the_length_of idl,
  hi is_the #n th_Identifier_in idl
.
RELATION SUBDEF
  #ord is_the_ordinal_number_of id in etd
:-
  id is_the #i th_Identifier_in idl,
  idl is_the_IdentifierList_of etd,
  etd is_a_EnumeratedTypeDefiner,
  #ord IS #i MINUS 1
.
```

```
(*/S 6.4.2.3: Enumerated-Types. *)

(*S  6.4.2.4: Subrange-Types. *)

CONSTRUCT SubrangeTypeDefiner IS
  <LowerBound:Constant> ".." <UpperBound:Constant>

(*
The LowerBound and the UpperBound of a SubrangeTypeDefiner
 must have ordinal types,
 and the type of the UpperBound must be
 the same as that of the LowerBound.
*)
RELATION SUBDEF
  c has_a_inappropriate_type
:-
  st is_a_SubrangeTypeDefiner,
  (c is_the_LowerBound_of st OR c is_the_UpperBound_of st),
  t is_the_type_of c,
  NOT (t is_a_ordinal_type)
.
RELATION SUBDEF
  st has_children_with_conflicting_types
:-
  st is_a_SubrangeTypeDefiner,
  lt is_the_type_of lb, lb is_the_LowerBound_of st,
  ut is_the_type_of ub, ub is_the_UpperBound_of st,
  NOT (lt is ut)
.

(*
The value denoted by the LowerBound of a SubrangeTypeDefiner
 must be less than or equal to
 the value denoted by the UpperBound of the SubrangeTypeDefiner.
*)
RELATION DEF
  VIOLATION st is_a_empty_subrange
:-
  st is_a_SubrangeTypeDefiner,
  lbv is_the_value_denoted_by lb, lb is_the_LowerBound_of st,
  ubv is_the_value_denoted_by ub, ub is_the_UpperBound_of st,
  lbv is_greater_than ubv
.

(*
Every ordinal-type has a range-type.
The range-type of a subrange-type is the type of its bounds.
The range-type of any other ordinal-type is the type itself.
*)
RELATION DEF
  rt is_the_range_type_of ot: FUNCTIONAL ON {ot}
:-
```

```
  rt is_a_ordinal_type,
  ot is_a_ordinal_type,
  IF (ot is_a_SubrangeTypeDefiner) THEN
    lb is_the_LowerBound_of ot,
    rt is_the_type_of lb
  ELSE
    rt is ot
  END
.

RELATION SUBDEF
  lo is_the_smallest_value_of st
:-
  st is_a_SubrangeTypeDefiner,
  lb is_the_LowerBound_of st,
  lo is_the_value_denoted_by lb
.
RELATION SUBDEF
  hi is_the_largest_value_of st
:-
  st is_a_SubrangeTypeDefiner,
  ub is_the_UpperBound_of st,
  hi is_the_value_denoted_by ub
.
RELATION SUBDEF
  #ord is_the_ordinal_number_of val in st
:-
  st is_a_SubrangeTypeDefiner,
  rt is_the_range_type_of st,
  #ord is_the_ordinal_number_of val in rt
.

(*/S 6.4.2.4: Subrange-Types. *)

(*/S 6.4.2: Simple-Types. *)

(*S  6.4.3: Structured-Types. *)

(*S  6.4.3.1: General. *)

ALTERNATE OPEN StructuredTypeDefiner IS
  PackedStructuredTypeDefiner | UnpackedStructuredTypeDefiner

CONSTRUCT PackedStructuredTypeDefiner IS
  "packed" <:UnpackedStructuredTypeDefiner>

ALTERNATE CLOSED UnpackedStructuredTypeDefiner IS
  ArrayTypeDefiner | RecordTypeDefiner | SetTypeDefiner |
  FileTypeDefiner

(*
The type denoted by a PackedStructuredTypeDefiner is
 the UnpackedStructuredTypeDefiner of the PackedStructuredTypeDefiner.
It shall be designated "packed".
*)
RELATION DECL t is_designated_packed .
RELATION SUBDEF
  t is_the_type_denoted_by pst,
  t is_designated_packed
:-
  t is_the_UnpackedStructuredTypeDefiner_of pst,
  pst is_a_PackedStructuredTypeDefiner
.

(*/S 6.4.3.1: General. *)

(*S  6.4.3.2: Array-Types. *)

RELATION DECL t is_a_array_type .

CONSTRUCT ArrayTypeDefiner IS
  "array" "[" <IndexTypeList:TypeDenoterList> "]" "of"
  <ComponentTypeDenoter:TypeDenoter>

LIST NONEMPTY TypeDenoterList OF TypeDenoter SEPARATED_BY _ ","

RELATION SUBDEF
  t is_a_array_type
:-
  t is_a_ArrayTypeDefiner
.

(*
The (unpacked) ArrayTypeDefiner
  array [td1,td2,...,tdn] of ctd
is equivalent to
  array [td1] of array [td2] of ... of array [tdn] of ctd

The PackedStructuredTypeDefiner
  packed array [td1,td2,...,tdn] of ctd
is equivalent to
  packed array [td1] of
    packed array [td2] of
      ... of
        packed array [tdn] of ctd
*)
RELATION SUBDEF
  atdex IS_THE_EXPANSION_OF atd
:-
  atd is_a_ArrayTypeDefiner,
  itl is_the_IndexTypeList_of atd,
  #len is_the_length_of itl,
  #len IS_GREATER_THAN 1,
  td_1 is_the 1 th_TypeDenoter_in itl,
  ctd is_the_ComponentTypeDenoter_of atd,
  ctdex := make_ArrayTypeDefiner ( tail(itl), ctd ),
  IF (atd is_designated_packed) THEN
    ctdexp := make_PackedStructuredTypeDefiner ( ctdex )
  ELSE
    ctdexp is ctdex
  END,
  atdex := make_ArrayTypeDefiner ( make_TypeDenoterList ( td_1 ), ctdexp )
.

(*
The index-type of an array-type (ArrayTypeDefiner)
 is the type denoted by
 the [only] TypeDenoter in the IndexTypeList of the ArrayTypeDefiner.
*)
RELATION DEF
  it is_the_index_type_of at: FUNCTIONAL ON {at}
:-
  at is_a_ArrayTypeDefiner,
  tdl is_the_IndexTypeList_of at,
  itd is_the 1 th_TypeDenoter_in tdl,
  it is_the_type_denoted_by itd
.

(*
The index-type of an array-type
 must be an ordinal-type.
*)
RELATION SUBDEF
  itd denotes_a_inappropriate_type
:-
  at is_a_ArrayTypeDefiner,
  tdl is_the_IndexTypeList_of at,
  itd is_the 1 th_TypeDenoter_in tdl,
  it is_the_type_denoted_by itd,
  NOT (it is_a_ordinal_type)
.

(*
The component-type of an array-type (ArrayTypeDefiner)
  is the type denoted by the ComponentTypeDenoter of the ArrayTypeDefiner.
*)
RELATION DEF
  ct is_the_array_component_type_of at: FUNCTIONAL ON {at}
:-
  at is_a_ArrayTypeDefiner,
  ctd is_the_ComponentTypeDenoter_of at,
  ct is_the_type_denoted_by ctd
.

RELATION DEF
  #n is_the_number_of_components_of at: FUNCTIONAL ON {at}
:-
  at is_a_ArrayTypeDefiner,
  it is_the_index_type_of at,
  #n is_the_number_of_values_defined_by it
.

(*
Any type designated packed and denoted by an array-type
 having as its index-type a denotation of a subrange-type
 specifying a smallest value of 1
 and a largest value of greater than 1,
 and having as its component-type
 a denotation of the char-type,
 shall be designated a string-type.
*)
RELATION DEF
  t is_a_string_type
:-
  t is_designated_packed,
  t is_a_ArrayTypeDefiner,
  it is_the_index_type_of t,
  it is_a_SubrangeTypeDefiner,
  it is_a_integer_type,
  lb is_the_LowerBound_of it, one is_the_value_denoted_by lb,
  ub is_the_UpperBound_of it, ubv is_the_value_denoted_by ub,
  one is_the_integer_value_for_ordinal 1,
  ubv is_greater_than one,
  ct is_the_array_component_type_of t,
  ct is_the_char_type
.

(*/S 6.4.3.2: Array-Types. *)

(*S  6.4.3.3: Record-Types. *)

RELATION DECL t is_a_record_type .

CONSTRUCT RecordTypeDefiner IS "record" <:FieldList> [";"] "end"

ALTERNATE OPEN FieldList IS FixedPart | VariantPart | FixedPartAndVariantPart

CONSTRUCT FixedPartAndVariantPart IS <:FixedPart> ";" <:VariantPart>

LIST FixedPart OF RecordSection SEPARATED_BY ";"

CONSTRUCT RecordSection IS <:IdentifierList> ":" <:TypeDenoter>

RELATION SUBDEF
  t is_a_record_type
:-
  t is_a_RecordTypeDefiner
.

(*
Utility
*)
```

```
RELATION DEF
  rt is_the_RecordTypeDefiner_closest_containing x: FUNCTIONAL ON {x}
:-
  p is_the_parent_of x,
  IF (p is_a_RecordTypeDefiner) THEN
    rt is p
  ELSE
    rt is_the_RecordTypeDefiner_closest_containing p
  END
.

RELATION DECL t is_a_field_type_denoter_of rt: FUNCTIONAL ON {t} .
RELATION DECL id declares_a_field_identifier .

RELATION DEF
  id is_a_field_identifier
:-
  d is_the_defining_occurrence_of id,
  d declares_a_field_identifier
.

(*
An Identifier in the IdentifierList of a RecordSection
 is a declaration-point
 as a field-identifier
 effective over the region that is the RecordTypeDefiner
   closest-containing the RecordSection,
and has the type denoted by
 the TypeDenoter of the RecordSection.
!: Overlap
*)
RELATION SUBDEF
  id is_a_declaration_point,
  id declares_a_field_identifier
:-
  id is_a_Identifier_in idl,
  idl is_the_IdentifierList_of rs,
  rs is_a_RecordSection
.
RELATION SUBDEF
  id is_effective_over rt
:-
  id is_a_Identifier_in idl,
  idl is_the_IdentifierList_of rs,
  rs is_a_RecordSection,
  rt is_the_RecordTypeDefiner_closest_containing rs
.
RELATION SUBDEF
  t is_the_type_of id
:-
  id is_a_Identifier_in idl,
  idl is_the_IdentifierList_of rs,
  rs is_a_RecordSection,
  td is_the_TypeDenoter_of rs,
  t is_the_type_denoted_by td
.
RELATION SUBDEF
  td is_a_field_type_denoter_of rt
:-
  td is_the_TypeDenoter_of rs,
  rs is_a_RecordSection,
  rt is_the_RecordTypeDefiner_closest_containing rs
.

CONSTRUCT VariantPart IS
  "case" [ <TagField:Identifier> ":" ] <TagTypeIdentifier:TypeIdentifier>
  "of" <:VariantList>

(*
If unempty, the TagField of a VariantPart
 is a declaration-point
 as a field-identifier
 effective over the region that is the RecordTypeDefiner
  closest-containing the VariantPart,
 and has the type denoted by
  the TagTypeIdentifier of the VariantPart.
!: Overlap
*)
RELATION SUBDEF
  id is_a_declaration_point,
  id declares_a_field_identifier
:-
  id is_a_Identifier,
  id is_the_TagField_of vp,
  vp is_a_VariantPart
.
RELATION SUBDEF
  id is_effective_over rt
:-
  id is_a_Identifier,
  id is_the_TagField_of vp,
  vp is_a_VariantPart,
  rt is_the_RecordTypeDefiner_closest_containing vp
.
RELATION SUBDEF
  tt is_the_type_of id
:-
  id is_a_Identifier,
  id is_the_TagField_of vp,
  vp is_a_VariantPart,
  tid is_the_TagTypeIdentifier_of vp,
  tt is_the_type_denoted_by tid
.
RELATION SUBDEF
  tid is_a_field_type_denoter_of rt
:-
```

```
  tid is_the_TagTypeIdentifier_of vp,
  vp is_a_VariantPart,
  rt is_the_RecordTypeDefiner_closest_containing vp
.

(*
The TagTypeIdentifier of a VariantPart must denote an ordinal-type.
*)
RELATION SUBDEF
  tid denotes_a_inappropriate_type
:-
  tid is_the_TagTypeIdentifier_of vp,
  vp is_a_VariantPart,
  tt is_the_type_denoted_by tid,
  NOT (tt is_a_ordinal_type)
.

LIST NONEMPTY VariantList OF Variant SEPARATED_BY ";"

CONSTRUCT Variant IS <:ConstantList> ":" "(" <:FieldList> [";"] ")"

LIST NONEMPTY ConstantList OF Constant SEPARATED_BY _ ","

(*
A Constant in the ConstantList of a Variant
 in the VariantList of a VariantPart
 is a selector-constant of the VariantPart.
(Useful for next two defs.)
*)
RELATION DEF
  c is_a_selector_constant_of vp: FUNCTIONAL ON {c}
:-
  c is_a_Constant_in cl,
  cl is_the_ConstantList_of v,
  v is_a_Variant_in vl,
  vl is_the_VariantList_of vp,
  vp is_a_VariantPart
.

(*
The type of each selector-constant of a VariantPart
 must be the range-type of
 the type denoted by the TagTypeIdentifier of the VariantPart
*)
RELATION SUBDEF
  c has_a_inappropriate_type
:-
  vp is_a_VariantPart,
  id is_the_TagTypeIdentifier_of vp,
  tt is_the_type_denoted_by id,
  c is_a_selector_constant_of vp,
  ct is_the_type_of c,
  NOT (ct is_the_range_type_of tt)
```

```
.

(*
The value of a selector-constant of a VariantPart
 must be different from all other such values
 (for the VariantPart).
*)
RELATION DEF
  VIOLATION c_2 denotes_a_duplicated_selector_value
:-
  vp is_a_VariantPart,
  c_1 is_a_selector_constant_of vp,
  c_2 is_a_selector_constant_of vp,
  NOT (c_1 is c_2),
  v is_the_value_denoted_by c_1,
  v is_the_value_denoted_by c_2
.

(*
The set of values denoted by all selector-constants of a VariantPart
 must be equal to the set of values
 specified by the type denoted by the TagTypeIdentifier of the VariantPart.
*)
RELATION DEF
  VIOLATION vp doesnt_have_selector_constants_for_all_selector_values
:-
  vp is_a_VariantPart,
  id is_the_TagTypeIdentifier_of vp,
  tt is_the_type_denoted_by id,
  val is_a_value_of tt,
  NOT (c is_a_selector_constant_of vp, val is_the_value_denoted_by c)
.
RELATION SUBDEF
  c is_not_allowed_in_this_context
:-
  vp is_a_VariantPart,
  id is_the_TagTypeIdentifier_of vp,
  tt is_the_type_denoted_by id,
  c is_a_selector_constant_of vp,
  val is_the_value_denoted_by c,
  NOT (val is_a_value_of tt)
.

(*
*)
RELATION DEF
  fl is_the_FieldList_associated_with val in vp:
    FUNCTIONAL ON {val,vp},{fl,val}
:-
  vp is_a_VariantPart,
  vl is_the_VariantList_of vp,
  v is_a_Variant_in vl,
  fl is_the_FieldList_of v,
  cl is_the_ConstantList_of v,
  c is_a_Constant_in cl,
  val is_the_value_denoted_by c
.

(*/S 6.4.3.3: Record-Types. *)

(*S  6.4.3.4: Set-Types. *)

CONSTRUCT SetTypeDefiner IS "set" "of" <BaseTypeDenoter:TypeDenoter>

(*
A SetTypeDefiner defines a set-type.
*)
RELATION DEF
  st is_a_set_type
:-
  st is_a_SetTypeDefiner
  OR
  st is_the_empty_set_type
.
RELATION SUBDEF
  st is_a_type
:-
  st is_a_set_type
.

(*
The BaseTypeDenoter of a SetTypeDefiner
 must denote an ordinal type.
*)
RELATION SUBDEF
  btd denotes_a_inappropriate_type
:-
  st is_a_SetTypeDefiner,
  btd is_the_BaseTypeDenoter_of st,
  bt is_the_type_denoted_by btd,
  NOT (bt is_a_ordinal_type)
.

(*
The base-type of a set-type defined by a SetTypeDefiner
 is the type denoted by the BaseTypeDenoter of the SetTypeDefiner.
The empty-set-type does not have a base-type.
*)
RELATION DEF
  bt is_the_base_type_of st: FUNCTIONAL ON {st}
:-
  st is_a_SetTypeDefiner,
  btd is_the_BaseTypeDenoter_of st,
  bt is_the_type_denoted_by btd
.
```

```
(*
The BaseTypeDenoter of a canonical set type
 shall denote the ordinal type that gave rise to the canonical set type.
*)
RELATION SUBDEF
  bt is_the_type_denoted_by btd
:-
  btd is_the_BaseTypeDenoter_of st,
  (
    st is_the_packed_canonical_set_type_associated_with bt
    OR
    st is_the_unpacked_canonical_set_type_associated_with bt
  )
.

(*
For every non-subrange ordinal-type T,
 there exist two canonical set-of-T types,
 packed and unpacked.
*)
RELATION DEF
  st is_a_canonical_set_type
:-
  st is_the_packed_canonical_set_type_associated_with bt
  OR
  st is_the_unpacked_canonical_set_type_associated_with bt
.
RELATION DEF
  MAKER st is_the_packed_canonical_set_type_associated_with bt:
  FUNCTIONAL ON {st},{bt}
:-
  bt is_a_ordinal_type,
  NOT ( bt is_a_SubrangeTypeDefiner ),
  st := make_SetTypeDefiner ( bt ),
  p := make_PackedStructuredTypeDefiner ( st )
.
RELATION DEF
  MAKER st is_the_unpacked_canonical_set_type_associated_with bt:
  FUNCTIONAL ON {st},{bt}
:-
  bt is_a_ordinal_type,
  NOT ( bt is_a_SubrangeTypeDefiner ),
  st := make_SetTypeDefiner ( bt )
.

RELATION DEF
  cst is_the_canonical_set_type_corresponding_to st: FUNCTIONAL ON {st}
:-
  bt is_the_base_type_of st,
  rt is_the_range_type_of bt,
  IF (st is_designated_packed) THEN
    cst is_the_packed_canonical_set_type_associated_with rt
  ELSE
```

```
    cst is_the_unpacked_canonical_set_type_associated_with rt
  END
.

(*/S 6.4.3.4: Set-Types. *)

(*S  6.4.3.5: File-Types. *)

RELATION DECL t is_a_file_type .

CONSTRUCT FileTypeDefiner IS "file" "of" <ComponentTypeDenoter:TypeDenoter>

RELATION SUBDEF
  t is_a_file_type
:-
  t is_a_FileTypeDefiner
.

(*
The component-type of a file-type (FileTypeDefiner)
 is the type denoted by the ComponentTypeDenoter of the FileTypeDefiner.
*)
RELATION DEF
  ct is_the_file_component_type_of ftd: FUNCTIONAL ON {ftd}
:-
  ftd is_a_FileTypeDefiner,
  ctd is_the_ComponentTypeDenoter_of ftd,
  ct is_the_type_denoted_by ctd
.

(*
A type
 shall not be permissible as the component-type of a file-type
 if it is either a file-type
 or a structured-type having any component-type that
 is not permissible as the component-type of a file-type.
*)
RELATION SUBDEF
  ctd denotes_a_inappropriate_type
:-
  ctd is_the_ComponentTypeDenoter_of ft, ft is_a_file_type,
  t is_the_type_denoted_by ctd,
  t is_a_invalid_file_component_type
.
RELATION DEF
  t is_a_invalid_file_component_type
:-
  t is_a_file_type
OR
  t is_a_array_type,
  ct is_the_array_component_type_of t,
  ct is_a_invalid_file_component_type
OR
```

```
  t is_a_record_type,
  ftd is_a_field_type_denoter_of t,
  ft is_the_type_denoted_by ftd,
  ft is_a_invalid_file_component_type
.

(*
There shall be a file-type
 that is denoted by the required structured-type-Identifier TEXT.
A variable that possesses
 the type denoted by the required structured-type-Identifier TEXT
  shall be designated a <&ital('textfile')>.
*)
RELATION DEF
  MAKER t is_the_textfile_type: FUNCTIONAL ON {}
:-
  t := make_FileTypeDefiner ( make_Identifier("char") )
.

(*/S 6.4.3.5: File-Types. *)

(*/S 6.4.3: Structured-Types. *)

(*S  6.4.4: Pointer-Types. *)

RELATION DEF
  t is_a_pointer_type
:-
  t is_a_PointerTypeDefiner
  OR
  t is_the_null_type
.

CONSTRUCT PointerTypeDefiner IS "@" _ <DomainTypeIdentifier:TypeIdentifier>

(*
6.2.2.9:
A TypeIdentifier may be the DomainTypeIdentifier of *any* PointerTypeDefiner
 contained by the TypeDefinitionList
 that contains the defining-occurrence of the TypeIdentifier.
*)
RELATION DEF
  id is_a_use_exception
:-
  id is_the_DomainTypeIdentifier_of pt,
  pt is_a_PointerTypeDefiner,
  tdl contains pt,
  tdl is_a_TypeDefinitionList,
  dp is_the_defining_occurrence_of id,
  dp is_the_Lhs_of td,
  td is_a_TypeDefinition_in tdl
.
```

```
(*/S 6.4.4: Pointer-Types. *)

(*S  6.4.5: Compatible Types. *)

(*
Used only in 6.4.6 and 6.7.2.
*)
RELATION DEF
  t_1 is_comparable_with t_2
:-
  t_1 is_a_ordinal_type, t is_the_range_type_of t_1,
  t_2 is_a_ordinal_type, t is_the_range_type_of t_2
OR
  t_1 is_the_real_type,
  t_2 is_the_real_type
OR
  t_1 is_a_pointer_type,
  t_2 is_a_pointer_type,
  ( t_1 is_the_null_type OR t_2 is_the_null_type OR t_1 is t_2 )
OR
  t_1 is_a_set_type,
  t_2 is_a_set_type,
  (
    t_1 is_the_empty_set_type
  OR
    t_2 is_the_empty_set_type
  OR
    cst is_the_canonical_set_type_corresponding_to t_1,
    cst is_the_canonical_set_type_corresponding_to t_2
  )
OR
  t_1 is_a_string_type,
  t_2 is_a_string_type,
  #c is_the_number_of_components_of t_1,
  #c is_the_number_of_components_of t_2
OR
  (
    t_1 is_the_real_type, t_2 is_a_integer_type
  OR
    t_2 is_the_real_type, t_1 is_a_integer_type
  )
.

(*/S 6.4.5: Compatible Types. *)

(*S  6.4.6: Assignment-Compatibility. *)

(*
Used in 6.5.3.2, 6.6.3.2, 6.8.2.2, and 6.8.3.9.
*)
RELATION DEF
  t_2 is_assignable_to t_1
:-
```

```
  t_1 is_a_type, t_2 is_a_type, t_1 is t_2,
  NOT (t_1 is_a_invalid_file_component_type)
OR
  t_1 is_comparable_with t_2,
  NOT (t_2 is_the_real_type, t_1 is_a_integer_type)
.

(*/S 6.4.6: Assignment-Compatibility. *)

(*S  6.4.7: Example of a Type-Definition-Part. *)

(*/S 6.4.7: Example of a Type-Definition-Part. *)

(*/S 6.4: TYPE-DEFINITIONS. *)

(*S  6.5: DECLARATIONS AND DENOTATIONS OF VARIABLES. *)

(*S  6.5.1: Variable-Declarations. *)

CONSTRUCT VariableDeclaration IS
  <:IdentifierList> ":" <:TypeDenoter> ";"

(*
An Identifier in the IdentifierList
 of a VariableDeclaration of the VariableDeclarationList of a Block
 is a declaration-point
 as a variable-identifier
 over the region that is the Block,
and
 denotes a distinct variable
 possessing the type
 denoted by the TypeDenoter of the VariableDeclaration.
!: Overlap
*)
RELATION SUBDEF
  id is_a_declaration_point,
  id declares_a_variable_identifier
:-
  id is_a_Identifier_in idl,
  idl is_the_IdentifierList_of vd,
  vd is_a_VariableDeclaration
.
RELATION SUBDEF
  id is_effective_over b
:-
  id is_a_Identifier_in idl,
  idl is_the_IdentifierList_of vd,
  vd is_a_VariableDeclaration_in vdl,
  vdl is_the_VariableDeclarationList_of b,
  b is_a_Block
.
RELATION SUBDEF
  t is_the_type_of id
```

```
:-
  id is_a_Identifier_in idl,
  idl is_the_IdentifierList_of vd,
  vd is_a_VariableDeclaration,
  td is_the_TypeDenoter_of vd,
  t is_the_type_denoted_by td
.

ALTERNATE OPEN VariableAccess IS
  Identifier | ComponentVariable | HattedVariable

(*
Normally, a VariableAccess must denote a variable.
There are some exceptions in which an Identifier
 occurring in the context of a VariableAccess
 should not, or need not, denote a variable.
*)
RELATION DECL va denotes_a_variable .
RELATION DECL x should_denote_a_variable .
RELATION DEF
  VIOLATION x should_denote_a_variable_but_doesnt
:-
  x should_denote_a_variable,
  NOT (x denotes_a_variable)
.

RELATION SUBDEF
  va denotes_a_variable
:-
  va is_a_variable_identifier
OR
  va is_a_field_identifier
OR
  va is_a_ComponentVariable
OR
  va is_a_HattedVariable
.

(*/S 6.5.1: Variable-Declarations. *)

(*S  6.5.2: Entire-Variables. *)

RELATION DECL id declares_a_variable_identifier .

RELATION DEF
  id is_a_variable_identifier
:-
  d is_the_defining_occurrence_of id,
  d declares_a_variable_identifier
.

(*/S 6.5.2: Entire-Variables. *)
```

```
(*S  6.5.3: Component-Variables. *)

(*S  6.5.3.1: General. *)

ALTERNATE CLOSED ComponentVariable IS IndexedVariable | SelectedVariable

(*/S 6.5.3.1: General. *)

(*S  6.5.3.2: Indexed-Variables. *)

CONSTRUCT IndexedVariable IS
  <ArrayVariable:VariableAccess> "[" <IndexList:ExpressionList> "]"

LIST NONEMPTY ExpressionList OF Expression SEPARATED_BY _ ","

(*
The ArrayVariable of an IndexedVariable
 must denote a variable possessing an array-type.
*)
RELATION SUBDEF
  va should_denote_a_variable
:-
  iv is_a_IndexedVariable,
  va is_the_ArrayVariable_of iv
.
RELATION SUBDEF
  va has_a_inappropriate_type
:-
  iv is_a_IndexedVariable,
  va is_the_ArrayVariable_of iv,
  t is_the_type_of va,
  NOT (t is_a_array_type)
.

(*
The IndexedVariable
  v [e1,e2,...,en]
is equivalent to
  v [e1] [e2] ... [en]
*)
RELATION SUBDEF
  ivex IS_THE_EXPANSION_OF iv
:-
  iv is_a_IndexedVariable,
  v is_the_ArrayVariable_of iv,
  il is_the_IndexList_of iv,
  #len is_the_length_of il,
  #len IS_GREATER_THAN 1,
  e_1 is_the 1 th_Expression_in il,
  ivex :=
    make_IndexedVariable
      ( make_IndexedVariable ( v, make_ExpressionList(e_1) ),
        tail(il)
      )
.

(*
The type of the only Expression in the IndexList of an IndexedVariable
 must be assignable to
 the index-type of the array-type of
  the ArrayVariable of the IndexedVariable.
*)
RELATION SUBDEF
  expr has_a_inappropriate_type
:-
  iv is_a_IndexedVariable,
  il is_the_IndexList_of iv,
  expr is_the 1 th_Expression_in il,
  et is_the_type_of expr,
  va is_the_ArrayVariable_of iv,
  at is_the_type_of va,
  it is_the_index_type_of at,
  NOT (et is_assignable_to it)
.

(*
The type of an IndexedVariable is
 the component-type of
 the type of the ArrayVariable of
 the IndexedVariable.
*)
RELATION SUBDEF
  t is_the_type_of iv
:-
  iv is_a_IndexedVariable,
  t is_the_array_component_type_of at,
  at is_the_type_of va,
  va is_the_ArrayVariable_of iv
.

(*/S 6.5.3.2: Indexed-Variables. *)

(*S  6.5.3.3: Field-Designators. *)

CONSTRUCT SelectedVariable IS
  <RecordVariable:VariableAccess> _ "." _ <FieldSpecifier:Identifier>

(*
The RecordVariable of a SelectedVariable
  must denote a variable possessing a record-type.
!: Overlap
*)
RELATION SUBDEF
  va should_denote_a_variable
:-
  sv is_a_SelectedVariable,
  va is_the_RecordVariable_of sv
.
RELATION SUBDEF
  va has_a_inappropriate_type
:-
  sv is_a_SelectedVariable,
  va is_the_RecordVariable_of sv,
  t is_the_type_of va,
  NOT (t is_a_record_type)
.

(*
An Identifier that is effective over
 the record-type possessed by the RecordVariable of a SelectedVariable
 is also effective over the region that is
 the FieldSpecifier of the SelectedVariable.
*)
RELATION SUBDEF
  d is_effective_over id
:-
  sv is_a_SelectedVariable,
  id is_the_FieldSpecifier_of sv,
  va is_the_RecordVariable_of sv,
  rt is_the_type_of va,
  rt is_a_record_type,
  d is_effective_over rt
.

(*
The defining-occurrence of the FieldSpecifier of a SelectedVariable
 must be a field-identifier for the record-type
 that is the type of the RecordVariable of the SelectedVariable.
*)
RELATION SUBDEF
  id is_not_allowed_in_this_context
:-
  sv is_a_SelectedVariable,
  va is_the_RecordVariable_of sv,
  rt is_the_type_of va,
  id is_the_FieldSpecifier_of sv,
  d is_the_defining_occurrence_of id,
  NOT (d is_effective_over rt)
.

(*
The type of a SelectedVariable is
 the type of the FieldSpecifier of the SelectedVariable.
*)
RELATION SUBDEF
  t is_the_type_of sv
:-
  sv is_a_SelectedVariable,
  fs is_the_FieldSpecifier_of sv,
  t is_the_type_of fs
.

(*/S 6.5.3.3: Field-Designators. *)

(*/S 6.5.3: Component-Variables. *)

CONSTRUCT HattedVariable IS <HeadVariable:VariableAccess> _ "@"

(*
The HeadVariable of a HattedVariable
 must be a pointer-variable or a file-variable.
*)
RELATION SUBDEF
  va should_denote_a_variable
:-
  hv is_a_HattedVariable,
  va is_the_HeadVariable_of hv
.
RELATION SUBDEF
  va has_a_inappropriate_type
:-
  hv is_a_HattedVariable,
  va is_the_HeadVariable_of hv,
  NOT (va is_a_pointer_variable OR va is_a_file_variable)
.

(*S  6.5.4: Identified-Variables. *)

(*
A pointer-variable is a VariableAccess whose type is a pointer-type.
*)
RELATION DEF
  va is_a_pointer_variable
:-
  va denotes_a_variable,
  pt is_the_type_of va,
  pt is_a_pointer_type
.

(*
An identified-variable is a HattedVariable
 whose HeadVariable is a pointer-variable.
*)
RELATION DEF
  va is_the_pointer_variable_of hv: FUNCTIONAL ON {va},{hv}
:-
  hv is_a_HattedVariable,
  va is_the_HeadVariable_of hv,
  va is_a_pointer_variable
.
RELATION DEF
  hv is_a_identified_variable
```

```
:-
  va is_the_pointer_variable_of hv
.

(*
The type of an identified-variable shall be
 [the type denoted by]
 the DomainTypeIdentifier of
 the pointer-type possessed by
 the pointer-variable of
 the identified-variable.
*)
RELATION SUBDEF
  t is_the_type_of iv
:-
  iv is_a_identified_variable,
  va is_the_pointer_variable_of iv,
  pt is_the_type_of va,
  pt is_a_pointer_type,
  dt is_the_DomainTypeIdentifier_of pt,
  t is_the_type_denoted_by dt
.

(*/S 6.5.4: Identified-Variables. *)

(*S  6.5.5: Buffer-Variables. *)

(*
A file-variable is a VariableAccess whose type is a file-type.
*)
RELATION DEF
  va is_a_file_variable
:-
  va denotes_a_variable,
  ft is_the_type_of va,
  ft is_a_file_type
.

(*
A buffer-variable is a HattedVariable
 whose HeadVariable is a file-variable.
*)
RELATION DEF
  va is_the_file_variable_of hv: FUNCTIONAL ON {va},{hv}
:-
  va is_the_HeadVariable_of hv,
  hv is_a_HattedVariable,
  va is_a_file_variable
.
RELATION DEF
  hv is_a_buffer_variable
:-
  va is_the_file_variable_of hv
```

```
.

(*
The type of a buffer-variable is
 the component-type of
 the file-type possessed by
 the file-variable of the buffer-variable.
*)
RELATION SUBDEF
  t is_the_type_of bv
:-
  bv is_a_buffer_variable,
  va is_the_file_variable_of bv,
  ft is_the_type_of va,
  ft is_a_file_type,
  t is_the_file_component_type_of ft
.

(*/S 6.5.5: Buffer-Variables. *)

(*/S 6.5: DECLARATIONS AND DENOTATIONS OF VARIABLES. *)

(*S  6.6: PROCEDURE AND FUNCTION DECLARATIONS. *)

(*S  6.6.1: Procedure-Declarations. *)

RELATION DECL id declares_a_procedure_identifier .
RELATION DECL VIOLATION x should_be_a_procedure_identifier_but_isnt .

RELATION DEF
  id is_a_procedure_identifier
:-
  d is_the_defining_occurrence_of id,
  d declares_a_procedure_identifier
.

CONSTRUCT ProcedureDeclaration IS
  "procedure" <Name:Identifier> [ "(" <:FormalParameterList> ")" ] ";"
  <Body:RoutineBody> ";"

RELATION SUBDEF
  id declares_a_procedure_identifier
:-
  id declares_a_routine_identifier,
  id is_the_Name_of pd, pd is_a_ProcedureDeclaration
.

(*/S 6.6.1: Procedure-Declarations. *)

(*S  6.6.2: Function-Declarations. *)

RELATION DECL id declares_a_function_identifier .
RELATION DECL VIOLATION x should_be_a_function_identifier_but_isnt .
RELATION DEF
  id is_a_function_identifier
:-
  d is_the_defining_occurrence_of id,
  d declares_a_function_identifier
.

ALTERNATE CLOSED FunctionDeclaration IS FunctionDef | FunctionRes

CONSTRUCT FunctionDef IS
  "function" <Name:Identifier> [ "(" <:FormalParameterList> ")" ] ":"
  <ResultTypeIdentifier:TypeIdentifier> ";" <Body:RoutineBody> ";"

CONSTRUCT FunctionRes IS
  "function" <Name:Identifier> ";" <Body:Block> ";"

RELATION SUBDEF
  id declares_a_function_identifier
:-
  id declares_a_routine_identifier,
  id is_the_Name_of fd, fd is_a_FunctionDeclaration
.

(*
The ResultTypeIdentifier of a FunctionDef
 must denote an ordinal-type, the real-type, or a pointer-type.
*)
RELATION SUBDEF
  id denotes_a_inappropriate_type
:-
  id is_the_ResultTypeIdentifier_of fd, fd is_a_FunctionDef,
  t is_the_type_denoted_by id,
  NOT (t is_a_ordinal_type OR t is_the_real_type OR t is_a_pointer_type)
.

(*
The Block associated with a function-identifier must contain
 at least one AssignmentStatement
 such that the VariableAccess of the AssignmentStatement
 is an applied occurrence of the function-identifier.
*)
RELATION DEF
  VIOLATION
    b is_a_function_block_with_no_assignment_to_the_function_identifier
:-
  b is_the_Block_associated_with fid,
  fid declares_a_function_identifier,
  NOT
  (
    b contains asmt,
    asmt is_a_AssignmentStatement,
    id is_the_VariableAccess_of asmt,
    id is_a_Identifier,
    id is_a_applied_occurrence_of fid
  )
.

(*/S 6.6.2: Function-Declarations. *)

ALTERNATE OPEN RoutineBody IS Directive | Block

(*
A ProcedureDeclaration d_2 is the resolution of ProcedureDeclaration d_1
 if d_2 and d_1 are in the same RoutineDeclarationList,
 their Names have the same spelling,
 d_1's Body is the Directive FORWARD,
 and d_2's Body is a Block.
Ditto (mutatis mutandis) for FunctionDeclarations.
*)
RELATION DEF
  d_2 is_the_resolution_of d_1: FUNCTIONAL ON {d_2},{d_1}
:-
  rdl is_a_RoutineDeclarationList,
  d_1 is_a_RoutineDeclaration_in rdl,
  d_2 is_a_RoutineDeclaration_in rdl,
  ( d_1 is_a_ProcedureDeclaration,
    d_2 is_a_ProcedureDeclaration
  OR
    d_1 is_a_FunctionDef,
    d_2 is_a_FunctionRes ),
  n_1 is_the_Name_of d_1,
  n_2 is_the_Name_of d_2,
  $s is_the_spelling_of n_1,
  $s is_the_spelling_of n_2,
  b_1 is_the_Body_of d_1,
  b_1 is_a_Directive, "forward" is_the_spelling_of b_1,
  b_2 is_the_Body_of d_2,
  b_2 is_a_Block
.
RELATION DEF
  d_2 is_a_resolution
:-
  d_2 is_the_resolution_of d_1
.

(*
Every RoutineDeclaration whose Body is the Directive FORWARD
 must have exactly one resolution.
*)
RELATION DEF
  VIOLATION rd is_a_unresolved_forward_declaration
:-
  rd is_a_RoutineDeclaration,
  body is_the_Body_of rd,
  body is_a_Directive,
  "forward" is_the_spelling_of body,
  NOT (r is_the_resolution_of rd)
.

(*
The Name of a RoutineDeclaration
 in the RoutineDeclarationList
 of a Block
 is a declaration-point
 over the region that is the Block,
 provided that the RoutineDeclaration is not a resolution.
*)
RELATION DECL id declares_a_routine_identifier .

RELATION SUBDEF
  id is_a_declaration_point,
  id declares_a_routine_identifier
:-
  id is_the_Name_of rd,
  rd is_a_RoutineDeclaration,
  NOT (rd is_a_resolution)
.
RELATION SUBDEF
  id is_effective_over b
:-
  id is_the_Name_of rd,
  rd is_a_RoutineDeclaration_in rdl,
  rdl is_the_RoutineDeclarationList_of b,
  b is_a_Block,
  NOT (rd is_a_resolution)
.

RELATION DEF
  b is_the_Block_associated_with id: FUNCTIONAL ON {b},{id}
:-
  id is_the_Name_of rd,
  rd is_a_RoutineDeclaration,
  IF (res is_the_resolution_of rd) THEN
    b is_the_Body_of res
  ELSE
    b is_the_Body_of rd
  END,
  b is_a_Block
.

(*S  6.6.3: Parameters. *)

(*S  6.6.3.1: General. *)

LIST NONEMPTY FormalParameterList OF FormalParameterSection SEPARATED_BY ";"

ALTERNATE CLOSED FormalParameterSection IS
  ValueParameterSection | VariableParameterSection |
  ProceduralParameterSection | FunctionalParameterSection

(*
A FormalParameterList comprises a sequence of formal parameters.
A FormalParameterSection declares one or more formal parameters.
*)

RELATION DEF
  fp is_a_formal_parameter_in fpl
:-
  fp is_a_formal_parameter_from fps,
  fps is_a_FormalParameterSection_in fpl
.
RELATION DEF
  fp is_the #i th_formal_parameter_in fpl: FUNCTIONAL ON {fp},{#i,fpl}
:-
  fp is_a_formal_parameter_in fpl,
  IF (fp_0 is_the_formal_parameter_preceding fp) THEN
    fp_0 is_the #i_0 th_formal_parameter_in fpl,
    #i IS #i_0 PLUS 1
  ELSE
    #i EQUALS 1
  END
.
RELATION DEF
  #n is_the_number_of_formal_parameters_in fpl: FUNCTIONAL ON {fpl}
:-
  fpl is_a_FormalParameterList,
  fp is_the #n th_formal_parameter_in fpl,
  NOT (fp is_the_formal_parameter_preceding fp_2)
.

RELATION DEF
  fp_1 is_the_formal_parameter_preceding fp_2: FUNCTIONAL ON {fp_1},{fp_2}
:-
  fps is_a_FormalParameterSection,
  fp_1 is_the #k_1 th_formal_parameter_from fps,
  fp_2 is_the #k_2 th_formal_parameter_from fps,
  #k_1 IS #k_2 MINUS 1
  OR
  fpl is_a_FormalParameterList,
  fps_1 is_the #s_1 th_FormalParameterSection_in fpl,
  fps_2 is_the #s_2 th_FormalParameterSection_in fpl,
  #s_1 IS #s_2 MINUS 1,
  #n_1 is_the_number_of_formal_parameters_from fps_1,
  fp_1 is_the #n_1 th_formal_parameter_from fps_1,
  fp_2 is_the   1 th_formal_parameter_from fps_2
.

RELATION DEF
  fp is_a_formal_parameter_from fps: FUNCTIONAL ON {fp}
:-
  fp is_a_Identifier_in idl,
  idl is_the_IdentifierList_of fps,
  (fps is_a_ValueParameterSection OR fps is_a_VariableParameterSection)
OR
  fp is_the_Name_of fps,
  (fps is_a_ProceduralParameterSection OR
   fps is_a_FunctionalParameterSection)
.
RELATION DEF
  fp is_the #k th_formal_parameter_from fps: FUNCTIONAL ON {fp}, {#k,fps}
:-
  fp is_the #k th_Identifier_in idl,
  idl is_the_IdentifierList_of fps,
  (fps is_a_ValueParameterSection OR fps is_a_VariableParameterSection)
OR
  fp is_the_Name_of fps,
  (fps is_a_ProceduralParameterSection OR
   fps is_a_FunctionalParameterSection),
  #k EQUALS 1
.
RELATION DEF
  #n is_the_number_of_formal_parameters_from fps: FUNCTIONAL ON {fps}
:-
  #n is_the_length_of idl,
  idl is_the_IdentifierList_of fps,
  (fps is_a_ValueParameterSection OR fps is_a_VariableParameterSection)
OR
  #n EQUALS 1,
  (fps is_a_ProceduralParameterSection OR
   fps is_a_FunctionalParameterSection)
.

(*
A formal parameter from a FormalParameterSection
 in a FormalParameterList
 is a declaration-point
 effective over the region that is the FormalParameterList,
 and
 effective over the region that is the Block (if any)
 associated with the FormalParameterList.
*)

RELATION SUBDEF
  id is_a_declaration_point
:-
  id is_a_formal_parameter_from fps,
  fps is_a_FormalParameterSection
.
RELATION SUBDEF
  id is_effective_over fpl
:-
  id is_a_formal_parameter_from fps,
  fps is_a_FormalParameterSection_in fpl,
  fpl is_a_FormalParameterList
```

```
.
RELATION SUBDEF
  dp is_effective_over b
:-
  dp is_effective_over fpl,
  fpl is_the_FormalParameterList_of x,
  id is_the_Name_of x,
  b is_the_Block_associated_with id
.

LIST NONEMPTY ActualParameterList OF ActualParameter SEPARATED_BY _ ","

ALTERNATE OPEN ActualParameter IS Expression | WriteParameter

RELATION DEF
  ap is_a_ActualParameter
:-
  ap is_a_ActualParameter_in apl
.

(*
A WriteParameter may only occur in the ActualParameterList
 of a ProcedureStatement whose Callee denotes WRITE or WRITELN.
*)
RELATION SUBDEF
  ap is_not_allowed_in_this_context
:-
  ap is_a_WriteParameter,
  ap is_a_ActualParameter_in apl,
  apl is_the_ActualParameterList_of call,
  NOT
  ( call is_a_ProcedureStatement,
    r is_the_routine_called_by call,
    ( r is_the_required_routine_named "write" OR
      r is_the_required_routine_named "writeln" )
  )
.

(*
ALTERNATE OPEN RoutineHeader IS
  ProcedureDeclaration | FunctionDef |
  ProceduralParameterSection | FunctionalParameterSection
*)
RELATION DEF
  r is_the_routine_called_by call
:-
  (
    call is_a_ProcedureStatement, id is_the_Callee_of call
  OR
    call is_a_FunctionCall, id is_the_Callee_of call
  OR
    call is_a_parameterless_function_call, id is call
  ),
  d is_the_defining_occurrence_of id,
  d is_the_Name_of r,
  (r is_a_ProcedureDeclaration OR
   r is_a_FunctionDef OR
   r is_a_ProceduralParameterSection OR
   r is_a_FunctionalParameterSection)
.

(*
The number of actual Parameters
 must be equal to the number of formal Parameters.
(This is complicated by empty ParameterLists and
  parameterless-function-calls.)
*)
RELATION DECL VIOLATION call has_the_wrong_number_of_actual_parameters .
RELATION SUBDEF
  call has_the_wrong_number_of_actual_parameters
:-
  (
    call is_a_parameterless_function_call, #aps EQUALS 0
  OR
    (call is_a_ProcedureStatement OR call is_a_FunctionCall),
    apl is_the_ActualParameterList_of call,
    (
      apl is_empty, #aps EQUALS 0
    OR
      apl is_a_ActualParameterList, #aps is_the_length_of apl
    )
  ),
  r is_the_routine_called_by call,
  fpl is_the_FormalParameterList_of r,
  NOT
    (
      fpl has_a_variable_number_of_formal_parameters
    OR
      fpl is_empty, #aps EQUALS 0
    OR
      fpl is_a_FormalParameterList,
      #aps is_the_number_of_formal_parameters_in fpl
    )
.

(*
Only required routines can have variable-length parameter lists,
 and only some of them do.
The ones that do are declared in the predefined-block
 without a FormalParameterList.
*)
RELATION DEF
  fpl has_a_variable_number_of_formal_parameters
:-
  fpl is_the_FormalParameterList_of rd, rd is_a_required_RoutineDeclaration,
  fpl is_empty
```

```
.
RELATION DEF
  rd is_a_required_RoutineDeclaration
:-
  rd is_a_RoutineDeclaration_in rdl,
  rdl is_the_RoutineDeclarationList_of b,
  b is_the_required_Block
.

(*
The correspondence shall be established by the positions of the
 Parameters.
*)
RELATION DEF
  fp is_the_formal_for ap: FUNCTIONAL ON {ap}
:-
  r is_the_routine_called_by call,
  fpl is_the_FormalParameterList_of r,
  apl is_the_ActualParameterList_of call,
  ap is_the #i th_ActualParameter_in apl,
  fp is_the #i th_formal_parameter_in fpl
.

(*/S 6.6.3.1: General. *)

(*S  6.6.3.2: Value Parameters. *)

CONSTRUCT ValueParameterSection IS
  <:IdentifierList> ":" <ParameterTypeIdentifier:TypeIdentifier>

(*
The ParameterTypeIdentifier of a ValueParameterSection
 must denote a type
 that is permitted as the component-type of a file-type.
*)
RELATION SUBDEF
  id denotes_a_inappropriate_type
:-
  id is_the_ParameterTypeIdentifier_of vps, vps is_a_ValueParameterSection,
  t is_the_type_denoted_by id,
  t is_a_invalid_file_component_type
.

(*
An Identifier in the IdentifierList of a ValueParameterSection
 is a formal parameter from the ValueParameterSection
 and is a variable-identifier
 whose type is the type denoted by
 the ParameterTypeIdentifier of the ValueParameterSection.
*)
RELATION SUBDEF
  id declares_a_variable_identifier
:-
```

```
  id is_a_formal_parameter_from vps,
  vps is_a_ValueParameterSection
.
RELATION SUBDEF
  t is_the_type_of id
:-
  vps is_a_ValueParameterSection,
  id is_a_formal_parameter_from vps,
  tid is_the_ParameterTypeIdentifier_of vps,
  t is_the_type_denoted_by tid
.

(*
An ActualParameter that corresponds to
 a formal parameter from a ValueParameterSection
 shall be designated an actual value parameter.
*)
RELATION DEF
  ap is_a_actual_value_parameter
:-
  ap is_a_ActualParameter,
  fp is_the_formal_for ap,
  fp is_a_formal_parameter_from vps,
  vps is_a_ValueParameterSection
.

(*
The type of an actual value parameter
 must be assignable to
 the type of the corresponding formal parameter.
*)
RELATION SUBDEF
  ap has_a_inappropriate_type
:-
  ap is_a_actual_value_parameter,
  tap is_the_type_of ap,
  fp is_the_formal_for ap,
  tfp is_the_type_of fp,
  NOT (tap is_assignable_to tfp)
.

(*/S 6.6.3.2: Value Parameters. *)

(*S  6.6.3.3: Variable Parameters. *)

CONSTRUCT VariableParameterSection IS
  "var" <:IdentifierList> ":" <ParameterTypeIdentifier:TypeIdentifier>

(*
An Identifier in the IdentifierList of a VariableParameterSection
 is a formal parameter from the VariableParameterSection
 and declares a variable-identifier
 whose type is the type denoted by
 the ParameterTypeIdentifier of the ValueParameterSection.
*)
RELATION SUBDEF
  id declares_a_variable_identifier
:-
  id is_a_formal_parameter_from fps,
  fps is_a_VariableParameterSection
.
RELATION SUBDEF
  t is_the_type_of id
:-
  vps is_a_VariableParameterSection,
  id is_a_formal_parameter_from vps,
  tid is_the_ParameterTypeIdentifier_of vps,
  t is_the_type_denoted_by tid
.

(*
An ActualParameter that corresponds to
 a formal parameter from a VariableParameterSection
 shall be designated an actual variable parameter.
*)
RELATION DEF
  ap is_a_actual_variable_parameter
:-
  ap is_a_ActualParameter,
  fp is_the_formal_for ap,
  fp is_a_formal_parameter_from vps,
  vps is_a_VariableParameterSection
.

(*
An actual variable parameter:
 1) must denote a variable.
 2) must possess the type of the corresponding formal parameter.
 3) must not denote the TagField of a VariantPart.
 4) must not denote a component of a variable
     whose type is designated packed,
      unless it is the file-parameter to a required I/O routine.
*)
RELATION SUBDEF
  ap should_denote_a_variable
:-
  ap is_a_actual_variable_parameter
.
RELATION SUBDEF
  ap has_a_inappropriate_type
:-
  ap is_a_actual_variable_parameter,
  fp is_the_formal_for ap,
  t is_the_type_of fp,
  NOT (t is_the_type_of ap)
.
```

```
RELATION DEF
  VIOLATION ap is_a_invalid_actual_variable_parameter
:-
  ap is_a_actual_variable_parameter,
  (
    ap denotes_a_tag_field
  OR
    ap denotes_a_component_of_a_variable_of_type t,
    t is_designated_packed,
    NOT ( ap is_a_file_parameter )
  )
.

RELATION DEF
  ap is_a_file_parameter
:-
  ap is_the 1 th_ActualParameter_in apl,
  apl is_the_ActualParameterList_of call,
  r is_the_routine_called_by call,
  ( r is_the_required_routine_named "rewrite" OR
    r is_the_required_routine_named "put" OR
    r is_the_required_routine_named "reset" OR
    r is_the_required_routine_named "get"
  )
.

RELATION DEF
  va denotes_a_tag_field
:-
  (
    va is id, id is_a_Identifier
  OR
    va is_a_SelectedVariable, id is_the_FieldSpecifier_of va
  ),
  d is_the_defining_occurrence_of id,
  d is_the_TagField_of vp
.

RELATION DEF
  va denotes_a_component_of_a_variable_of_type t: FUNCTIONAL ON {va}
:-
  (
    va is id, id is_a_Identifier
  OR
    va is_a_SelectedVariable, id is_the_FieldSpecifier_of va
  ),
  d is_the_defining_occurrence_of id,
  d is_effective_over t,
  t is_a_record_type
OR
  va is_a_IndexedVariable,
  av is_the_ArrayVariable_of va,
  t is_the_type_of av,
  t is_a_array_type
.

(*/S 6.6.3.3: Variable Parameters. *)

(*S  6.6.3.4: Procedural Parameters. *)

CONSTRUCT ProceduralParameterSection IS
  "procedure" <Name:Identifier> [ "(" <:FormalParameterList> ")" ]

(*
The Name of a ProceduralParameterSection
 is a formal parameter from the ProceduralParameterSection
 and declares a procedure-identifier
 [with associated FormalParameterList?]
*)
RELATION SUBDEF
  id declares_a_procedure_identifier
:-
  id is_the_Name_of fps,
  fps is_a_ProceduralParameterSection
.

(*
An ActualParameter that corresponds to
 a formal parameter from a ProceduralParameterSection
 shall be designated an actual procedural parameter.
*)
RELATION DEF
  ap is_a_actual_procedural_parameter
:-
  ap is_a_ActualParameter,
  fp is_the_formal_for ap,
  fp is_a_formal_parameter_from fps,
  fps is_a_ProceduralParameterSection
.

(*
An actual procedural parameter
 must be a procedure-identifier
 whose defining occurrence is contained by the Block of the Program.
*)
RELATION SUBDEF
  ap should_be_a_procedure_identifier_but_isnt
:-
  ap is_a_actual_procedural_parameter,
  NOT (ap is_a_procedure_identifier)
.
RELATION SUBDEF
  ap is_not_allowed_in_this_context
:-
  ap is_a_actual_procedural_parameter,
  dap is_the_defining_occurrence_of ap,
  dap is_effective_over r,
  r is_the_required_Block
.

(*
The procedure denoted by an actual procedural parameter
 and the formal-parameter
 must have congruous FormalParameterLists.
*)
RELATION SUBDEF
  ap has_a_incongruous_FormalParameterList
:-
  ap is_a_actual_procedural_parameter,
  dp is_the_defining_occurrence_of ap,
  fp is_the_formal_for ap,
  dp is_the_Name_of arh,
  fp is_the_Name_of frh,
  al is_the_FormalParameterList_of arh,
  fl is_the_FormalParameterList_of frh,
  NOT (fl is_congruous_with al)
.

(*/S 6.6.3.4: Procedural Parameters. *)

(*S  6.6.3.5: Functional Parameters. *)

CONSTRUCT FunctionalParameterSection IS
  "function" <Name:Identifier> [ "(" <:FormalParameterList> ")" ] ":"
  <ResultTypeIdentifier:Identifier>

(*
The Name of a FunctionalParameterSection
 is a formal parameter from the FunctionalParameterSection
 and declares a function-identifier.
*)
RELATION SUBDEF
  id declares_a_function_identifier
:-
  id is_the_Name_of fps,
  fps is_a_FunctionalParameterSection
.

(*
An ActualParameter that corresponds to
 a formal parameter from a ProceduralParameterSection
 shall be designated an actual functional parameter.
*)
RELATION DEF
  ap is_a_actual_functional_parameter
:-
  ap is_a_ActualParameter,
  fp is_the_formal_for ap,
  fp is_a_formal_parameter_from fps,
  fps is_a_FunctionalParameterSection
.

(*
An actual functional parameter
 must be a function-identifier
 whose defining occurrence is contained by the Block of the Program.
*)
RELATION SUBDEF
  ap should_be_a_function_identifier_but_isnt
:-
  ap is_a_actual_functional_parameter,
  NOT (ap is_a_function_identifier)
.
RELATION SUBDEF
  ap is_not_allowed_in_this_context
:-
  ap is_a_actual_functional_parameter,
  dap is_the_defining_occurrence_of ap,
  dap is_effective_over r,
  r is_the_required_Block
.

(*
The function denoted by an actual functional parameter
 and the formal-parameter
 must have the same result-type
 and congruous FormalParameterLists.
!: The relation's name should be generalized
   to indicate the possibility of mismatched result-types.
*)
RELATION SUBDEF
  ap has_a_incongruous_FormalParameterList
:-
  ap is_a_actual_functional_parameter,
  dp is_the_defining_occurrence_of ap,
  fp is_the_formal_for ap,
  dp is_the_Name_of arh,
  fp is_the_Name_of frh,
  art is_the_type_denoted_by artd, artd is_the_ResultTypeIdentifier_of arh,
  frt is_the_type_denoted_by frtd, frtd is_the_ResultTypeIdentifier_of frh,
  al is_the_FormalParameterList_of arh,
  fl is_the_FormalParameterList_of frh,
  NOT (art is frt, fl is_congruous_with al)
.

(*/S 6.6.3.5: Functional Parameters. *)

(*S  6.6.3.6: Parameter List Congruity. *)

RELATION DECL VIOLATION ap has_a_incongruous_FormalParameterList .

RELATION DEF
```

```
  fpl_1 is_congruous_with fpl_2
:-
  fpl_1 is_empty,
  fpl_2 is_empty
OR
  fpl_1 is_a_FormalParameterList,
  fpl_2 is_a_FormalParameterList,
  #len is_the_length_of fpl_1,
  #len is_the_length_of fpl_2,
  NOT
  ( fps_1 is_the #i th_FormalParameterSection_in fpl_1,
    fps_2 is_the #i th_FormalParameterSection_in fpl_2,
    NOT (fps_1 matches fps_2)
  )
.

RELATION DEF
  fps_1 matches fps_2
:-
(
  fps_1 is_a_ValueParameterSection,
  fps_2 is_a_ValueParameterSection
OR
  fps_1 is_a_VariableParameterSection,
  fps_2 is_a_VariableParameterSection
),
  idl_1 is_the_IdentifierList_of fps_1,
  idl_2 is_the_IdentifierList_of fps_2,
  #len is_the_length_of idl_1,
  #len is_the_length_of idl_2,
  ptd_1 is_the_ParameterTypeIdentifier_of fps_1,
  ptd_2 is_the_ParameterTypeIdentifier_of fps_2,
  pt is_the_type_denoted_by ptd_1,
  pt is_the_type_denoted_by ptd_2
OR
  fps_1 is_a_ProceduralParameterSection,
  fps_2 is_a_ProceduralParameterSection,
  fpl_1 is_the_FormalParameterList_of fps_1,
  fpl_2 is_the_FormalParameterList_of fps_2,
  fpl_1 is_congruous_with fpl_2
OR
  fps_1 is_a_FunctionalParameterSection,
  fps_2 is_a_FunctionalParameterSection,
  fpl_1 is_the_FormalParameterList_of fps_1,
  fpl_2 is_the_FormalParameterList_of fps_2,
  fpl_1 is_congruous_with fpl_2,
  id_1 is_the_ResultTypeIdentifier_of fps_1,
  id_2 is_the_ResultTypeIdentifier_of fps_2,
  rt is_the_type_denoted_by id_1,
  rt is_the_type_denoted_by id_2
.

(*/S 6.6.3.6: Parameter List Congruity. *)
```

```
(*/S 6.6.3: Parameters. *)

(*S  6.6.4: Required Procedures and Functions. *)
(*/S 6.6.4: Required Procedures and Functions. *)

(*S  6.6.5: Required Procedures. *)

RELATION SUBDEF
  call has_the_wrong_number_of_actual_parameters
:-
  call is_a_ProcedureStatement,
  apl is_the_ActualParameterList_of call,
  r is_the_routine_called_by call,
  (
    ( r is_the_required_routine_named "read" OR
      r is_the_required_routine_named "write" ),
    (
      apl is_empty
    OR
      l is_the_length_of apl,
      ap is_the l th_ActualParameter_in apl,
      tap is_the_type_of ap,
      tap is_a_file_type
    )
  OR
    r is_the_required_routine_named "page",
    #len is_the_length_of apl, #len IS_GREATER_THAN 1
  OR
    ( r is_the_required_routine_named "new" OR
      r is_the_required_routine_named "dispose" ),
    apl is_empty
  )
.

(*
Several required procedures have a particular type-constraint
 on the first (sometimes only) actual parameter.
*)
RELATION SUBDEF
  ap has_a_inappropriate_type
:-
  call is_a_ProcedureStatement,
  apl is_the_ActualParameterList_of call,
  ap is_the 1 th_ActualParameter_in apl,
  tap is_the_type_of ap,
  r is_the_routine_called_by call,
  (
    (
      r is_the_required_routine_named "rewrite"
    OR
      r is_the_required_routine_named "put"
    OR
```

```
      r is_the_required_routine_named "reset"
    OR
      r is_the_required_routine_named "get"
    ),
    NOT (tap is_a_file_type)
  OR
    r is_the_required_routine_named "page",
    NOT (tap is_the_textfile_type)
  OR
    (
      r is_the_required_routine_named "new"
    OR
      r is_the_required_routine_named "dispose"
    ),
    NOT (tap is_a_pointer_type)
  )
.

(*
NEW: the first parameter must be a variable.
*)
RELATION SUBDEF
  ap should_denote_a_variable
:-
  call is_a_ProcedureStatement,
  r is_the_routine_called_by call,
  r is_the_required_routine_named "new",
  apl is_the_ActualParameterList_of call,
  ap is_the 1 th_ActualParameter_in apl
.

(*
NEW and DISPOSE: are complicated by optional tag constants.
!: Overlap
*)
RELATION DEF
  vp is_the_VariantPart_raised_by ap: FUNCTIONAL ON {ap}
:-
  call is_a_ProcedureStatement,
  r is_the_routine_called_by call,
  (r is_the_required_routine_named "new" OR
   r is_the_required_routine_named "dispose"),
  apl is_the_ActualParameterList_of call,
  ap is_the #i th_ActualParameter_in apl,
  IF (#i EQUALS 1) THEN
    tap is_the_type_of ap,
    tap is_a_pointer_type,
    tid is_the_DomainTypeIdentifier_of tap,
    dt is_the_type_denoted_by tid,
    dt is_a_record_type,
    fl is_the_FieldList_of dt
  ELSE
    #i_0 IS #i MINUS 1,
    ap_0 is_the #i_0 th_ActualParameter_in apl,
    vp_0 is_the_VariantPart_raised_by ap_0,
    val is_the_value_denoted_by ap,
    fl is_the_FieldList_associated_with val in vp_0
  END,
  (
    fl is_a_VariantPart, vp is fl
  OR
    fl is_a_FixedPartAndVariantPart, vp is_the_VariantPart_of fl
  )
.
RELATION SUBDEF
  ap is_not_allowed_in_this_context
:-
  call is_a_ProcedureStatement,
  r is_the_routine_called_by call,
  (r is_the_required_routine_named "new" OR
   r is_the_required_routine_named "dispose"),
  apl is_the_ActualParameterList_of call,
  ap is_the #i th_ActualParameter_in apl,
  #i IS_GREATER_THAN 1,
  #i_0 IS #i MINUS 1,
  ap_0 is_the #i_0 th_ActualParameter_in apl,
  NOT
  ( vp_0 is_the_VariantPart_raised_by ap_0,
    val is_the_value_denoted_by ap,
    fl is_the_FieldList_associated_with val in vp_0
  )
.

(*
READ, READLN: require variable parameters.
*)
RELATION SUBDEF
  ap should_denote_a_variable
:-
  call is_a_ProcedureStatement,
  r is_the_routine_called_by call,
  (r is_the_required_routine_named "read" OR
   r is_the_required_routine_named "readln"),
  apl is_the_ActualParameterList_of call,
  ap is_a_ActualParameter_in apl
.

(*
WRITELN, READLN: apply only to textfiles.
(This can only be violated by an explicit file-variable.)
*)
RELATION SUBDEF
  ap_1 has_a_inappropriate_type
:-
  call is_a_ProcedureStatement,
  r is_the_routine_called_by call,
  (r is_the_required_routine_named "readln" OR
   r is_the_required_routine_named "writeln"),
  apl is_the_ActualParameterList_of call,
  ap_1 is_the 1 th_ActualParameter_in apl,
  tap_1 is_the_type_of ap_1,
  tap_1 is_a_file_type,
  NOT (tap_1 is_the_textfile_type)
.

(*
READ, READLN, WRITE, WRITELN:
 all have an optional file-variable as the first parameter.
If omitted, the file-variable is taken to be INPUT or OUTPUT,
 as appropriate.
The other parameters have other type-constraints,
 significantly depending on whether or not
 the file-variable's type is TEXT.
*)
RELATION SUBDEF
  ap has_a_inappropriate_type
:-
  call is_a_ProcedureStatement,
  r is_the_routine_called_by call,
  (r is_the_required_routine_named "read" OR
   r is_the_required_routine_named "write" OR
   r is_the_required_routine_named "readln" OR
   r is_the_required_routine_named "writeln"),
  apl is_the_ActualParameterList_of call,
  ap_1 is_the 1 th_ActualParameter_in apl,
  tap_1 is_the_type_of ap_1,
  IF (tap_1 is_a_file_type) THEN
    ft is tap_1, NOT (ap is ap_1)
  ELSE
    ft is_the_textfile_type
  END,
  ap is_a_ActualParameter_in apl,
  tap is_the_type_of ap,
  IF (ft is_the_textfile_type) THEN
    (
      (r is_the_required_routine_named "read" OR
       r is_the_required_routine_named "readln"),
      NOT
       (tap is_a_char_type OR
        tap is_a_integer_type OR
        tap is_the_real_type)
    OR
      (r is_the_required_routine_named "write" OR
       r is_the_required_routine_named "writeln"),
      NOT
       (tap is_a_integer_type OR
        tap is_the_real_type OR
        tap is_a_char_type OR
        tap is_a_boolean_type OR
        tap is_a_string_type)
     )
   ELSE
     ct is_the_file_component_type_of ft,
     (
       (r is_the_required_routine_named "read" OR
        r is_the_required_routine_named "readln"),
       NOT (ct is_assignable_to tap)
     OR
       (r is_the_required_routine_named "write" OR
        r is_the_required_routine_named "writeln"),
       NOT (tap is_assignable_to ct)
     )
   END
.

(*
PACK and UNPACK have a number of constraints.
*)
RELATION SUBDEF
  ap has_a_inappropriate_type
:-
  call is_a_ProcedureStatement,
  apl is_the_ActualParameterList_of call,
  r is_the_routine_called_by call,
  (
    r is_the_required_routine_named "pack",
    a is_the 1 th_ActualParameter_in apl,
    i is_the 2 th_ActualParameter_in apl,
    z is_the 3 th_ActualParameter_in apl
  OR
    r is_the_required_routine_named "unpack",
    z is_the 1 th_ActualParameter_in apl,
    a is_the 2 th_ActualParameter_in apl,
    i is_the 3 th_ActualParameter_in apl
  ),
  ta is_the_type_of a,
  ti is_the_type_of i,
  tz is_the_type_of z,
  (
    NOT (ta is_a_array_type, NOT (ta is_designated_packed)), ap is a
  OR
    NOT (ti is_a_ordinal_type), ap is i
  OR
    NOT (tz is_a_array_type, tz is_designated_packed), ap is z
  OR
    ta is_a_array_type,
    s_1 is_the_index_type_of ta,
    ti is_a_ordinal_type,
    NOT (ti is_assignable_to s_1),
    ap is i
  OR
    ta is_a_array_type, tac is_the_array_component_type_of ta,
    tz is_a_array_type, tzc is_the_array_component_type_of tz,
    NOT (tac is tzc),
    (ap is a OR ap is z)
  )
.

(*/S 6.6.5: Required Procedures. *)

(*S  6.6.6: Required Functions. *)

(*
Functions SIN, COS, EXP, LN, SQRT, ARCTAN, TRUNC, ROUND, CHR, ODD
 are fully handled by the required-Block.
Functions ABS, SQR, ORD, SUCC, PRED, EOF, EOLN
 have quirks that can't be handled that way.
!: Overlap
*)

RELATION SUBDEF
  call has_the_wrong_number_of_actual_parameters
:-
  r is_the_routine_called_by call,
  (
    r is_the_required_routine_named "eof"
  OR
    r is_the_required_routine_named "eoln"
  ),
  NOT
    (
      call is_a_parameterless_function_call
    OR
      call is_a_FunctionCall,
      apl is_the_ActualParameterList_of call,
      1 is_the_length_of apl
    )
.

RELATION SUBDEF
  ap has_a_inappropriate_type
:-
  call is_a_FunctionCall,
  apl is_the_ActualParameterList_of call,
  ap is_the 1 th_ActualParameter_in apl,
  tap is_the_type_of ap,
  r is_the_routine_called_by call,
  (
    r is_the_required_routine_named "abs",
    NOT (tap is_a_integer_type OR tap is_the_real_type)
  OR
    r is_the_required_routine_named "sqr",
    NOT (tap is_a_integer_type OR tap is_the_real_type)
  OR
    r is_the_required_routine_named "ord",
    NOT (tap is_a_ordinal_type)
  OR
    r is_the_required_routine_named "succ",
    NOT (tap is_a_ordinal_type)
  OR
    r is_the_required_routine_named "pred",
    NOT (tap is_a_ordinal_type)
  OR
    r is_the_required_routine_named "eof",
    NOT (tap is_a_file_type)
  OR
    r is_the_required_routine_named "eoln",
    NOT (tap is_a_file_type)
  )
.

RELATION SUBDEF
  tap is_the_type_of call
:-
  call is_a_FunctionCall,
  r is_the_routine_called_by call,
  (
    r is_the_required_routine_named "abs"
  OR
    r is_the_required_routine_named "sqr"
  OR
    r is_the_required_routine_named "succ"
  OR
    r is_the_required_routine_named "pred"
  ),
  apl is_the_ActualParameterList_of call,
  ap is_the 1 th_ActualParameter_in apl,
  tap is_the_type_of ap
.

(*/S 6.6.6: Required Functions. *)

(*/S 6.6: PROCEDURE AND FUNCTION DECLARATIONS. *)

(*S  6.7: EXPRESSIONS. *)

(*S  6.7.1: General. *)

(*
An Expression shall yield a value.
*)
RELATION DEF
  x is_a_genuine_expression
:-
  x is_in_a_Expression_context,
  NOT
  ( x is_a_actual_variable_parameter OR
    x is_a_actual_procedural_parameter OR
    x is_a_actual_functional_parameter OR
    x is_the_FracDigits_of wp, wp is_a_WriteParameter, x is_empty
  )
.
RELATION DECL VIOLATION x has_children_with_conflicting_types .

ALTERNATE OPEN Expression IS RelationalExpression | SimpleExpression

CONSTRUCT RelationalExpression IS
  <LeftOperand:SimpleExpression> <Op:RelOp>
  <RightOperand:SimpleExpression>

ALTERNATE OPEN SimpleExpression IS AddingExpression | SignedTerm | Term

CONSTRUCT AddingExpression IS
  <LeftOperand:SimpleExpression> <Op:AddOp> <RightOperand:Term>

CONSTRUCT SignedTerm IS <:Sign> <:Term>

ALTERNATE OPEN Term IS MultiplyingExpression | Factor

CONSTRUCT MultiplyingExpression IS
  <LeftOperand:Term> <Op:MultOp> <RightOperand:Factor>

ALTERNATE OPEN Factor IS
  VariableAccess | FunctionCall | SetConstructor |
  BrackettedExpression | NegationExpression |
  UnsignedIntegerLiteral | UnsignedRealLiteral |
  CharacterString | Nil

CONSTRUCT BrackettedExpression IS "(" <:Expression> ")"
(*
The type of a BrackettedExpression is that of its Expression.
*)
RELATION SUBDEF
  t is_the_type_of brack
:-
  brack is_a_BrackettedExpression,
  expr is_the_Expression_of brack,
  t is_the_type_of expr
.

CONSTRUCT NegationExpression IS "not" <:Factor>

CONSTRUCT Nil IS "nil"
(*
The type of Nil is the null-type.
(The Standard (Section 6.4.4) says that
"Nil does not have a single type,
 but assumes a suitable pointer-type
 to satisfy the assignment-compatibility rules,
 or the compatibility rules for operators,
 if possible."
However, this definition is not very useful for our purposes.)
*)
RELATION SUBDEF
  t is_the_type_of expr
:-
  expr is_a_Nil,
  t is_the_null_type
.
RELATION DEF
  MAKER t is_the_null_type: FUNCTIONAL ON {}
:-
  t := make_Identifier("TheNullType")
.

(*
Set Constructors:
*)

CONSTRUCT SetConstructor IS "[" <:MemberDesignatorList> "]"

LIST MemberDesignatorList OF MemberDesignator SEPARATED_BY _ ","

ALTERNATE CLOSED MemberDesignator IS SingletonDesignator | RangeDesignator

CONSTRUCT SingletonDesignator IS <:Expression>

CONSTRUCT RangeDesignator IS
  <LowerLimit:Expression> ".." <UpperLimit:Expression>

(*
The type of a SetConstructor with an empty MemberDesignatorList is
 the empty-set-type.
The type of a SetConstructor with a non-empty MemberDesignatorList is
  the type of the first MemberDesignator in the MemberDesignatorList.
*)
RELATION SUBDEF
  t is_the_type_of sc
:-
  sc is_a_SetConstructor,
  mdl is_the_MemberDesignatorList_of sc,
  IF (0 is_the_length_of mdl) THEN
    t is_the_empty_set_type
  ELSE
    md is_the 1 th_MemberDesignator_in mdl,
    t is_the_type_of md
  END
.
RELATION DEF
  MAKER t is_the_empty_set_type: FUNCTIONAL ON {}
:-
  t := make_Identifier ( "TheEmptySetType" )
.
```

```
(*
The type of a SingletonDesignator is
  the appropriate canonical-set-type of
  the range-type of
  the type of
  the Expression of the SingletonDesignator.
*)
RELATION SUBDEF
  t is_the_type_of sd
:-
  sd is_a_SingletonDesignator,
  e is_the_Expression_of sd,
  te is_the_type_of e,
  rt is_the_range_type_of te,
  IF (sd is_in_a_context_requiring_packed) THEN
    t is_the_packed_canonical_set_type_associated_with rt
  ELSE
    t is_the_unpacked_canonical_set_type_associated_with rt
  END
.

(*
The types of the LowerLimit and UpperLimit of a RangeDesignator
  must have the same range-type,
and the type of the RangeDesignator is
  the appropriate canonical-set-type of this range-type.
*)
RELATION SUBDEF
  rd has_children_with_conflicting_types
:-
  rd is_a_RangeDesignator,
  lt is_the_type_of ll, ll is_the_LowerLimit_of rd,
  ut is_the_type_of ul, ul is_the_UpperLimit_of rd,
  rt is_the_range_type_of lt,
  NOT (rt is_the_range_type_of ut)
.
RELATION SUBDEF
  t is_the_type_of rd
:-
  rd is_a_RangeDesignator,
  lt is_the_type_of ll, ll is_the_LowerLimit_of rd,
  rt is_the_range_type_of lt,
  IF (rd is_in_a_context_requiring_packed) THEN
    t is_the_packed_canonical_set_type_associated_with rt
  ELSE
    t is_the_unpacked_canonical_set_type_associated_with rt
  END
.

(*
The type of each MemberDesignator in a MemberDesignatorList must be
  the same.
*)
```

```
RELATION SUBDEF
  mdl has_children_with_conflicting_types
:-
  mdl is_a_MemberDesignatorList,
  md_1 is_the 1 th_MemberDesignator_in mdl,
  t_1 is_the_type_of md_1,
  md is_a_MemberDesignator_in mdl,
  NOT (t_1 is_the_type_of md)
.

(*
The "appropriate" canonical set-type is determined by context.
*)
RELATION DEF
  x is_in_a_context_requiring_packed
:-
  asmt is_a_AssignmentStatement,
  x is_the_Expression_of asmt,
  va is_the_VariableAccess_of asmt,
  tv is_the_type_of va,
  tv is_a_set_type,
  tv is_designated_packed
OR
  x is_a_actual_value_parameter,
  fp is_the_formal_for x,
  tfp is_the_type_of fp,
  tfp is_a_set_type,
  tfp is_designated_packed
OR
  expr is_a_dyadic_expression,
  op is_the_Op_of expr, NOT (op is_a_InOp),
  (
    y is_the_LeftOperand_of expr, x is_the_RightOperand_of expr
  OR
    x is_the_LeftOperand_of expr, y is_the_RightOperand_of expr
  ),
  ty is_the_type_of y,
  ty is_a_set_type,
  ty is_designated_packed
OR
  expr is_a_dyadic_expression,
  expr is_in_a_context_requiring_packed,
  (x is_the_LeftOperand_of expr OR x is_the_RightOperand_of expr)
OR
  sc is_a_SetConstructor,
  sc is_in_a_context_requiring_packed,
  mdl is_the_MemberDesignatorList_of sc,
  x is_a_MemberDesignator_in mdl
.

(*/S 6.7.1: General. *)

(*S 6.7.2: Operators. *)
```

```
ALTERNATE CLOSED RelOp IS
  EqOp | UneqOp | LtOp | GtOp | LeqOp | GeqOp | InOp

CONSTRUCT EqOp IS "="
CONSTRUCT UneqOp IS "<>"
CONSTRUCT LtOp IS "<"
CONSTRUCT GtOp IS ">"
CONSTRUCT LeqOp IS "<="
CONSTRUCT GeqOp IS ">="
CONSTRUCT InOp IS "in"

ALTERNATE OPEN AddOp IS PlusOp | MinusOp | OrOp

CONSTRUCT PlusOp IS "+"
CONSTRUCT MinusOp IS "-"
CONSTRUCT OrOp IS "or"

ALTERNATE CLOSED MultOp IS StarOp | SlashOp | DivOp | ModOp | AndOp

CONSTRUCT StarOp IS "*"
CONSTRUCT SlashOp IS "/"
CONSTRUCT DivOp IS "div"
CONSTRUCT ModOp IS "mod"
CONSTRUCT AndOp IS "and"

(*
!: could be an auxiliary cf rule:
ALTERNATE CLOSED DyadicEpression IS
  RelationalExpression | AddingExpression | MultiplyingExpression
*)
RELATION DEF
  expr is_a_dyadic_expression
:-
  expr is_a_RelationalExpression
OR
  expr is_a_AddingExpression
OR
  expr is_a_MultiplyingExpression
.

RELATION DEF
  t is_a_relating_type
:-
  t is_a_ordinal_type OR t is_the_real_type OR t is_a_string_type
.

(*
Dyadic operations:
The types of operands and results for dyadic arithmetic operations
 ('+', '-', '*', '/', 'div', 'mod')
 must be as shown in Table 2.
operands and results for dyadic boolean operations
```

```
 ('or', 'and')
 must be of boolean type.
The types of operands and results for set operations
 ('+', '-', '*')
 must be as shown in Table 4.
The types of operands and results for relational operations
 ('=', '<>', '<', '>', '>=', '<=', 'in')
 must be as shown in Table 5.
*)
RELATION SUBDEF
  x has_a_inappropriate_type
:-
  expr is_a_dyadic_expression,
  op is_the_Op_of expr,
  (x is_the_LeftOperand_of expr OR x is_the_RightOperand_of expr),
  t is_the_type_of x,
  (
    (op is_a_PlusOp OR op is_a_MinusOp OR op is_a_StarOp),
    NOT (t is_a_integer_type OR t is_the_real_type OR t is_a_set_type)
  OR
    op is_a_SlashOp,
    NOT (t is_a_integer_type OR t is_the_real_type)
  OR
    (op is_a_DivOp OR op is_a_ModOp),
    NOT (t is_a_integer_type)
  OR
    (op is_a_OrOp OR op is_a_AndOp),
    NOT (t is_a_boolean_type)
  OR
    (op is_a_EqOp OR op is_a_UneqOp),
    NOT (t is_a_relating_type OR t is_a_set_type OR t is_a_pointer_type)
  OR
    (op is_a_LeqOp OR op is_a_GeqOp),
    NOT (t is_a_relating_type OR t is_a_set_type)
  OR
    (op is_a_LtOp OR op is_a_GtOp),
    NOT (t is_a_relating_type)
  )
OR
  expr is_a_dyadic_expression,
  op is_the_Op_of expr,
  op is_a_InOp,
  (
    x is_the_LeftOperand_of expr,
    t is_the_type_of x,
    NOT (t is_a_ordinal_type)
  OR
    x is_the_RightOperand_of expr,
    t is_the_type_of x,
    NOT (t is_a_set_type)
  )
.
RELATION SUBDEF
```

```
  expr has_children_with_conflicting_types
:-
  expr is_a_dyadic_expression,
  op is_the_Op_of expr,
  tl is_the_type_of l, l is_the_LeftOperand_of  expr,
  tr is_the_type_of r, r is_the_RightOperand_of expr,
(
  (op is_a_PlusOp OR op is_a_MinusOp OR
   op is_a_StarOp OR op is_a_SlashOp OR
   op is_a_EqOp   OR op is_a_UneqOp  OR
   op is_a_LeqOp  OR op is_a_GeqOp   OR
   op is_a_LtOp   OR op is_a_GtOp    ),
  NOT (tl is_comparable_with tr)
OR
  op is_a_InOp,
  NOT (tr is_the_empty_set_type),
  NOT
  (tr is_a_set_type,
   bt is_the_base_type_of tr,
   tl is_comparable_with bt)
)
.

RELATION SUBDEF
  t is_the_type_of expr
:-
  expr is_a_dyadic_expression,
  op is_the_Op_of expr,
(
  (op is_a_PlusOp OR op is_a_MinusOp OR op is_a_StarOp),
  tl is_the_type_of l, l is_the_LeftOperand_of  expr,
  tr is_the_type_of r, r is_the_RightOperand_of expr,
  (
    tl is_a_integer_type, tr is_a_integer_type, t is_the_integer_type
  OR
    (tl is_the_real_type OR tr is_the_real_type), t is_the_real_type
  OR
    tl is_a_set_type, tr is_a_set_type,
    (
      tl is_the_empty_set_type,
      tr is_the_empty_set_type,
      t  is_the_empty_set_type
    OR
      tl is_the_empty_set_type,
      NOT (tr is_the_empty_set_type),
      t is tr
    OR
      NOT (tl is_the_empty_set_type),
      tr is_the_empty_set_type,
      t is tl
    OR
      NOT (tl is_the_empty_set_type),
      NOT (tr is_the_empty_set_type),
      t is_the_canonical_set_type_corresponding_to tl
    )
  )
OR
  op is_a_SlashOp,
  t is_the_real_type
OR
  (op is_a_DivOp OR op is_a_ModOp),
  t is_the_integer_type
OR
  (op is_a_OrOp OR op is_a_AndOp OR op is_a_RelOp),
  t is_the_boolean_type
)
.

(*
Monadic operations:
The types of operands and results for monadic arithmetic operations
 ('+', '-')
 must be as shown in Table 3.
The types of operands and results for monadic boolean operations
 ('not')
 must be boolean.
*)
RELATION SUBDEF
  operand has_a_inappropriate_type
:-
  expr is_a_SignedTerm,
  operand is_the_Term_of expr,
  t is_the_type_of operand,
  NOT (t is_a_integer_type OR t is_the_real_type)
OR
  expr is_a_NegationExpression,
  operand is_the_Factor_of expr,
  t is_the_type_of operand,
  NOT (t is_a_boolean_type)
.
RELATION SUBDEF
  t is_the_type_of expr
:-
  expr is_a_SignedTerm,
  operand is_the_Term_of expr,
  top is_the_type_of operand,
  (
    top is_a_integer_type, t is_the_integer_type
  OR
    top is_the_real_type, t is_the_real_type
  )
OR
  expr is_a_NegationExpression,
  t is_the_boolean_type
.
```

```
(*
The required constant-identifier MAXINT
 shall denote an implementation-defined value of the integer-type.
*)
RELATION DEF
  v is_the_integer_value_maxint
:-
  v is_the_integer_value_for_ordinal 32767
.

(*/S 6.7.2: Operators. *)

(*S  6.7.3: Function-designators. *)

CONSTRUCT FunctionCall IS
  <Callee:Identifier> "(" <:ActualParameterList> ")"

(*
The Callee of a FunctionCall
 must denote a function.
*)
RELATION SUBDEF
  id should_be_a_function_identifier_but_isnt
:-
  id is_the_Callee_of fc, fc is_a_FunctionCall,
  NOT (id is_a_function_identifier)
.

(*
A function-identifier occurring in a FunctionCall context
 (excluding an actual-functional-parameter)
 is a parameterless-function-call.
*)
RELATION DEF
  id is_a_parameterless_function_call
:-
  id is_a_function_identifier,
  id is_in_a_Factor_context,
  NOT (id is_a_actual_functional_parameter)
.

(*
The type of a FunctionCall or parameterless-function-call is
 the type denoted by
 the ResultTypeIdentifier of
 the function called.
*)
RELATION SUBDEF
  t is_the_type_of fc
:-
  (fc is_a_FunctionCall OR fc is_a_parameterless_function_call),
  f is_the_routine_called_by fc,
  rt is_the_ResultTypeIdentifier_of f,
  t is_the_type_denoted_by rt
.

(*/S 6.7.3: Function-designators. *)

(*/S 6.7: EXPRESSIONS. *)

(*S  6.8: STATEMENTS. *)

(*S  6.8.1: General. *)

ALTERNATE OPEN Statement IS LabelledStatement | UnlabelledStatement

CONSTRUCT LabelledStatement IS <:Label> ":" <:UnlabelledStatement>

ALTERNATE CLOSED UnlabelledStatement IS
  SimpleStatement | StructuredStatement

(*
The type of the Condition of any Statement (that has one)
 must be BOOLEAN.
*)
RELATION SUBDEF
  e has_a_inappropriate_type
:-
  e is_the_Condition_of stmt,
  stmt is_a_UnlabelledStatement,
  t is_the_type_of e,
  NOT (t is_a_boolean_type)
.

(*/S 6.8.1: General. *)

(*S  6.8.2: Simple Statements. *)

(*S  6.8.2.1: General. *)

ALTERNATE CLOSED SimpleStatement IS
  EmptyStatement | AssignmentStatement | ProcedureStatement | GotoStatement

CONSTRUCT EmptyStatement IS

(*/S 6.8.2.1: General. *)

(*S  6.8.2.2: Assignment-Statement. *)

CONSTRUCT AssignmentStatement IS <:VariableAccess> ":=" <:Expression>

(*
The VariableAccess of an AssignmentStatement
 must denote a variable or a function.
*)
RELATION SUBDEF
```

```
  va is_not_allowed_in_this_context
:-
  va is_the_VariableAccess_of asmt,
  asmt is_a_AssignmentStatement,
  NOT (va denotes_a_variable OR va is_a_function_identifier)
.

(*
If the VariableAccess of an AssignmentStatement
 is a function-identifier,
 the AssignmentStatement must be within the Block associated with the
 defining-occurrence of the function-identifier.
*)
RELATION SUBDEF
  id is_not_allowed_in_this_context
:-
  id is_the_VariableAccess_of asmt,
  asmt is_a_AssignmentStatement,
  id is_a_function_identifier,
  d is_the_defining_occurrence_of id,
  b is_the_Block_associated_with d,
  NOT (b contains asmt)
.

(*
The type of the Expression of an AssignmentStatement
 must be assignable to the type of the VariableAccess.
*)
RELATION SUBDEF
  expr has_a_inappropriate_type
:-
  asmt is_a_AssignmentStatement,
  te is_the_type_of expr, expr is_the_Expression_of asmt,
  tv is_the_type_of va,   va is_the_VariableAccess_of asmt,
  NOT (te is_assignable_to tv)
.

(*/S 6.8.2.2: Assignment-Statement. *)

(*S  6.8.2.3: Procedure-Statements. *)

CONSTRUCT ProcedureStatement IS
  <Callee:Identifier> [ "(" <:ActualParameterList> ")" ]

(*
The Callee of a ProcedureStatement must denote a procedure.
*)
RELATION SUBDEF
  id should_be_a_procedure_identifier_but_isnt
:-
  id is_the_Callee_of ps,
  ps is_a_ProcedureStatement,
  NOT (id is_a_procedure_identifier)
```

```
.

(*/S 6.8.2.3: Procedure-Statements. *)

(*S  6.8.2.4: Goto-Statements. *)

CONSTRUCT GotoStatement IS "goto" <:Label>

(*
The target of a GotoStatement
 is the LabelledStatement whose Label
  is the site of the defining occurrence of
  the Label of the GotoStatement.
*)
RELATION DEF
  s is_the_target_of g: FUNCTIONAL ON {g}
:-
  s is_a_LabelledStatement,
  slab is_the_Label_of s,
  slab is_the_site_for d,
  d is_the_defining_occurrence_of glab,
  glab is_the_Label_of g,
  g is_a_GotoStatement
.

(*
6.8.1:
The Label of a LabelledStatement s
may have an applied occurrence
in a GotoStatement g
if and only if any of the following three Conditions is satisfied.
  (a) s contains g.
  (b) s is a Statement of a StatementSequence containing g.
  (c) s is a Statement of the StatementSequence of a Block containing g.
*)
RELATION SUBDEF
  g is_not_allowed_in_this_context
:-
  g is_a_GotoStatement,
  s is_the_target_of g,
  NOT
  (
    s contains g
  OR
    s is_a_Statement_in ss,
    ss contains g
  OR
    s is_a_Statement_in ss,
    ss is_the_StatementSequence_of b,
    b is_a_Block,
    b contains g
  )
.
```

```
(*/S 6.8.2.4: Goto-Statements. *)

(*/S 6.8.2: Simple Statements. *)

(*S  6.8.3: Structured-Statements. *)

(*S  6.8.3.1: General. *)

ALTERNATE CLOSED StructuredStatement IS
  CompoundStatement | ConditionalStatement | RepetitiveStatement |
  WithStatement

LIST NONEMPTY StatementSequence OF Statement SEPARATED_BY ";"

(*/S 6.8.3.1: General. *)

(*S  6.8.3.2: Compound-Statements. *)

CONSTRUCT CompoundStatement IS "begin" <:StatementSequence> "end"

(*/S 6.8.3.2: Compound-Statements. *)

(*S  6.8.3.3: Conditional-Statements. *)

ALTERNATE CLOSED ConditionalStatement IS IfStatement | CaseStatement

(*/S 6.8.3.3: Conditional-Statements. *)

(*S  6.8.3.4: If-Statements. *)

CONSTRUCT IfStatement IS
  "if" <Condition:Expression> "then" <Consequent:Statement>
  [ "else" <Alternate:Statement> ]

(*/S 6.8.3.4: If-Statements. *)

(*S  6.8.3.5: Case-Statements. *)

CONSTRUCT CaseStatement IS
  "case" <CaseIndex:Expression> "of" <:CaseArmList> [";"] "end"

(*
The type of the CaseIndex of a CaseStatement
 must be an ordinal-type.
*)
RELATION SUBDEF
  e has_a_inappropriate_type
:-
  cs is_a_CaseStatement,
  e is_the_CaseIndex_of cs,
  et is_the_type_of e,
  NOT (et is_a_ordinal_type)
```

```
.

LIST NONEMPTY CaseArmList OF CaseArm SEPARATED_BY ";"

CONSTRUCT CaseArm IS <:ConstantList> ":" <:Statement>

(*
A Constant in the ConstantList of a CaseArm
 in the CaseArmList of a CaseStatement
 is a case-constant of the CaseStatement.
(Useful for next two defs.)
*)
RELATION DEF
  c is_a_case_constant_of cs: FUNCTIONAL ON {c}
:-
  c is_a_Constant_in cl,
  cl is_the_ConstantList_of arm,
  arm is_a_CaseArm_in cal,
  cal is_the_CaseArmList_of cs,
  cs is_a_CaseStatement
.

(*
The type of each case-constant of a CaseStatement
 must be the range-type of
 the type of the CaseIndex of the CaseStatement.
!: Overlaps with two rules ago.
*)
RELATION SUBDEF
  c has_a_inappropriate_type
:-
  cs is_a_CaseStatement,
  e is_the_CaseIndex_of cs,
  et is_the_type_of e,
  c is_a_case_constant_of cs,
  ct is_the_type_of c,
  NOT (ct is_the_range_type_of et)
.

(*
The value of a case-constant of a CaseStatement
 must be different from all other such values
 (for the CaseStatement).
*)
RELATION DEF
  VIOLATION c_2 denotes_a_duplicated_case_value
:-
  cs is_a_CaseStatement,
  c_1 is_a_case_constant_of cs,
  c_2 is_a_case_constant_of cs,
  NOT (c_1 is c_2),
  v is_the_value_denoted_by c_1,
  v is_the_value_denoted_by c_2
```

```
.

(*/S 6.8.3.5: Case-Statements. *)

(*S  6.8.3.6: Repetitive-Statements. *)

ALTERNATE CLOSED RepetitiveStatement IS
  RepeatStatement | WhileStatement | ForStatement

(*/S 6.8.3.6: Repetitive-Statements. *)

(*S  6.8.3.7: Repeat-Statements. *)

CONSTRUCT RepeatStatement IS
  "repeat" <:StatementSequence> "until" <Condition:Expression>

(*/S 6.8.3.7: Repeat-Statements. *)

(*S  6.8.3.8: While-Statements. *)

CONSTRUCT WhileStatement IS
  "while" <Condition:Expression> "do" <:Statement>

(*/S 6.8.3.8: While-Statements. *)

(*S  6.8.3.9: For-Statements. *)

CONSTRUCT ForStatement IS
  "for" <ControlVariable:Identifier> ":=" <InitialValue:Expression>
  <:DirIndicator> <FinalValue:Expression> "do" <:Statement>

ALTERNATE CLOSED DirIndicator IS UpTo | DownTo

CONSTRUCT UpTo IS "to"

CONSTRUCT DownTo IS "downto"

(*
The ControlVariable of a ForStatement must be an entire-variable
 whose Identifier is declared in the VariableDeclarationList
 of the Block closest-containing the ForStatement.
*)
RELATION SUBDEF
  id should_denote_a_variable
:-
  for is_a_ForStatement,
  id is_the_ControlVariable_of for
.
RELATION SUBDEF
  id is_not_allowed_in_this_context
:-
  for is_a_ForStatement,
  id is_the_ControlVariable_of for,
  d is_the_defining_occurrence_of id,
  NOT
  (
    d is_a_Identifier_in idl,
    idl is_the_IdentifierList_of vd,
    vd is_a_VariableDeclaration_in vdl,
    vdl is_the_VariableDeclarationList_of b,
    b is_the_Block_closest_containing for
  )
.

(*
The ControlVariable of a ForStatement
 must possess an ordinal-type,
 and the InitialValue and FinalValue
 must be of a type assignable to this type.
*)
RELATION SUBDEF
  id has_a_inappropriate_type
:-
  for is_a_ForStatement,
  id is_the_ControlVariable_of for,
  tid is_the_type_of id,
  NOT (tid is_a_ordinal_type)
.
RELATION SUBDEF
  expr has_a_inappropriate_type
:-
  for is_a_ForStatement,
  id is_the_ControlVariable_of for,
  tid is_the_type_of id,
  (expr is_the_InitialValue_of for OR expr is_the_FinalValue_of for),
  te is_the_type_of expr,
  NOT (te is_assignable_to tid)
.

(*
The variable denoted by the control-variable of a for-Statement
 must not be threatened within
  the Statement of the ForStatement
 or
  the RoutineDeclarationList of the Block
  that closest-contains the ForStatement.
*)
RELATION DEF
  VIOLATION id is_a_illegal_threat
:-
  cv is_the_ControlVariable_of for,
  for is_a_ForStatement,
  d is_the_defining_occurrence_of cv,
  d is_the_defining_occurrence_of id,
  id is_a_threatened_variable,
(
  s is_the_Statement_of for,
  s contains id
OR
  rdl contains id,
  rdl is_the_RoutineDeclarationList_of b,
  b is_the_Block_closest_containing for
)
.

(*
A variable-identifier v is threatened
 if one or more of the following Statements is true.
1) v is the VariableAccess of an AssignmentStatement.
2) v is an ActualParameter corresponding to a variable-parameter.
3) v is a ActualParameter in the ActualParameterList
   of a ProcedureStatement whose Callee denotes READ or READLN.
4) v is the ControlVariable of a ForStatement.

(The Standard defines the relation
 "Statement threatens variable",
 but the Statement doing the threatening is not really important,
 in fact it introduces redundancy:
a single threat to a variable (as an "actual variable parameter")
 can give rise to many Statements threatening the variable,
 because they all *contain* the threat.)
*)
RELATION DEF
  id is_a_threatened_variable
:-
  id is_a_variable_identifier,
(
  id is_the_VariableAccess_of asmt,
  asmt is_a_AssignmentStatement
OR
  id is_a_ActualParameter,
  fp is_the_formal_for id,
  fp is_a_formal_parameter_from vps,
  vps is_a_VariableParameterSection
OR
  id is_a_ActualParameter_in apl,
  apl is_the_ActualParameterList_of ps,
  ps is_a_ProcedureStatement,
  r is_the_routine_called_by ps,
  (r is_the_required_routine_named "read" OR
   r is_the_required_routine_named "readln")
OR
  id is_the_ControlVariable_of for,
  for is_a_ForStatement
)
.

(*/S 6.8.3.9: For-Statements. *)
```

```
(*S  6.8.3.10: With-Statements. *)

CONSTRUCT WithStatement IS
  "with" <:VariableAccessList> "do" <:Statement>

LIST NONEMPTY VariableAccessList OF VariableAccess SEPARATED_BY _ ","

(*
The statement
  with v1, v2, ..., vn do
    s
is equivalent to
  with v1 do
    with v2 do
      ...
        with vn do
          s
*)
RELATION SUBDEF
  wex IS_THE_EXPANSION_OF w
:-
  w is_a_WithStatement,
  body is_the_Statement_of w,
  val is_the_VariableAccessList_of w,
  #len is_the_length_of val,
  #len IS_GREATER_THAN 1,
  va is_the 1 th_VariableAccess_in val,
  wex :=
    make_WithStatement
      ( make_VariableAccessList ( va ),
        make_WithStatement ( tail(val), body )
      )
.

(*
A VariableAccess in the VariableAccessList of a WithStatement
  must denote a variable possessing a record-type.
*)
RELATION SUBDEF
  va should_denote_a_variable
:-
  va is_a_VariableAccess_in val,
  val is_the_VariableAccessList_of with,
  with is_a_WithStatement
.
RELATION SUBDEF
  va has_a_inappropriate_type
:-
  va is_a_VariableAccess_in val,
  val is_the_VariableAccessList_of with,
  with is_a_WithStatement,
  t is_the_type_of va,
  NOT (t is_a_record_type)
```

```
.

(*
An Identifier that is effective over
 the record-type possessed by
 the VariableAccess of a WithStatement
 is also effective over the region that is
 the Statement of the WithStatement.
*)
RELATION SUBDEF
  d is_effective_over body
:-
  with is_a_WithStatement,
  body is_the_Statement_of with,
  val is_the_VariableAccessList_of with,
  va is_a_VariableAccess_in val,
  rt is_the_type_of va,
  rt is_a_record_type,
  d is_effective_over rt
.

(*/S 6.8.3.10: With-Statements. *)

(*/S 6.8.3: Structured-Statements. *)

(*/S 6.8: STATEMENTS. *)

(*S  6.9: INPUT AND OUTPUT. *)

(*S  6.9.1: The Procedure READ. *)

(*/S 6.9.1: The Procedure READ. *)

(*S  6.9.2: The Procedure READLN. *)

(*/S 6.9.2: The Procedure READLN. *)

(*S  6.9.3: The Procedure WRITE. *)

CONSTRUCT WriteParameter IS
  <:Expression> _ ":" _ <TotalWidth:Expression>
  [ _ ":" _ <FracDigits:Expression> ]

(*
The type of the TotalWidth and FracDigits of a WriteParameter
 must be an integer-type.
*)
RELATION SUBDEF
  expr has_a_inappropriate_type
:-
  wp is_a_WriteParameter,
  (expr is_the_TotalWidth_of wp OR expr is_the_FracDigits_of wp),
  t is_the_type_of expr,
  NOT (t is_a_integer_type)
.
(*
The FracDigits of a WriteParameter can be non-empty
 only if the type of the Expression of the WriteParameter is the real-type.
*)
RELATION SUBDEF
  frac is_not_allowed_in_this_context
:-
  wp is_a_WriteParameter,
  frac is_the_FracDigits_of wp,
  NOT (frac is_empty),
  expr is_the_Expression_of wp,
  t is_the_type_of expr,
  NOT (t is_the_real_type)
.

(*
The type of a WriteParameter
 is the type of the Expression of the WriteParameter.
(To make things easier elsewhere.)
*)
RELATION SUBDEF
  t is_the_type_of wp
:-
  wp is_a_WriteParameter,
  expr is_the_Expression_of wp,
  t is_the_type_of expr
.

(*/S 6.9.3: The Procedure WRITE. *)

(*S  6.9.4: The Procedure WRITELN. *)

(*/S 6.9.4: The Procedure WRITELN. *)

(*S  6.9.5: The Procedure PAGE. *)

(*/S 6.9.5: The Procedure PAGE. *)

(*/S 6.9: INPUT AND OUTPUT. *)

(*S 6.10: PROGRAMS. *)

CONSTRUCT Program IS
  "program" <Name:Identifier> [ "(" <Parameters:IdentifierList> ")" ]
  ";" <:Block> "."

(*
The Identifiers in the Parameters of a Program
 must have distinct spellings.
*)
RELATION SUBDEF
```

```
  id_2 is_a_conflicting_declaration_point
:-
  ids is_the_Parameters_of p,
  p is_a_Program,
  id_1 is_a_Identifier_in ids,
  id_2 is_a_Identifier_in ids,
  NOT (id_1 is id_2),
  $s is_the_spelling_of id_1,
  $s is_the_spelling_of id_2
.

(*
For each Identifier in the Parameters of a Program,
 there must be a declaration-point
 over a region that is the Block of the Program
 for a variable-identifier with the same spelling.
*)
RELATION SUBDEF
  id is_undefined
:-
  id is_a_Identifier_in il,
  il is_the_Parameters_of p,
  p is_a_Program,
  b is_the_Block_of p,
  NOT
    (d is_effective_over b,
     d declares_a_variable_identifier,
     $s is_the_spelling_of d,
     $s is_the_spelling_of id)
.

(*
The occurrence of
 the required Identifier INPUT or the required Identifier OUTPUT
 in the Parameters of a Program
 is a declaration-point
 over the region that is the Block of the Program
 as a variable-identifier
 of the textfile-type.
!: Overlap
*)
RELATION SUBDEF
  id is_a_declaration_point,
  id declares_a_variable_identifier
:-
  ("input" is_the_spelling_of id OR "output" is_the_spelling_of id),
  id is_a_Identifier_in idl,
  idl is_the_Parameters_of p,
  p is_a_Program
.
RELATION SUBDEF
  id is_effective_over b
:-
```

```
  ("input" is_the_spelling_of id OR "output" is_the_spelling_of id),
  id is_a_Identifier_in idl,
  idl is_the_Parameters_of p,
  p is_a_Program,
  b is_the_Block_of p
.
RELATION SUBDEF
  t is_the_type_of id
:-
  ("input" is_the_spelling_of id OR "output" is_the_spelling_of id),
  id is_a_Identifier_in idl,
  idl is_the_Parameters_of p,
  p is_a_Program,
  t is_the_textfile_type
.

(*/S 6.10: PROGRAMS. *)
```

APPENDIX 3: TESTING THE PASCAL SMS

The correctness of the SMS generated for Pascal, and thus, indirectly, the correctness of the NURN specification for Pascal, was tested using Version 5.0 of the Pascal Validation Suite (PVS), a large set of Pascal programs specifically written to validate Pascal processors (such as compilers, interpreters, etc.) [Wich83]. The programs of the PVS are divided into eight classes, exploring the performance of a Pascal processor with respect to various aspects of the Standard. Two of these classes (DEVIANCE and CONFORMANCE) are relevant for testing a context-dependent SMS for Pascal.

In the DEVIANCE class, each program contains a single violation of the Pascal standard, which a Pascal processor should detect and complain about. The Pascal SMS includes the investigative routine `U_IS_A_VIOLATION`, which can detect all such violations in a Pascal program, and thus can be used as the basis for a Pascal syntax-checker. The following Modula-2 program calls `ParseInput` to parse the input stream as a Pascal `Program`, calls `InstallSubject` and `FindAllExtensions` to calculate the extensions of all relations, and finally calls `U_IS_A_VIOLATION` to get any instances of violation relations. If there are none, then the input is syntactically correct.

```
MODULE PascalSyntaxCheck ;
FROM IO IMPORT Writeln ;
FROM GRAFS_Generic IMPORT Node ;
FROM NURN_Generic IMPORT
  InstallSubject, FindAllExtensions, U_IS_A_VIOLATION, EntityList ;
FROM Pasc IMPORT
  NodeClass, ParseInput ;

VAR prog : Node ;
BEGIN
  IF ParseInput ( Program, prog ) THEN
    Writeln ( 'CF parse succeeded.' ) ;
    InstallSubject ( prog ) ;
    FindAllExtensions () ;
    IF EntityList.IsEmpty ( U_IS_A_VIOLATION () ) THEN
      Writeln ( 'No CD violations detected.' )
    ELSE
      Writeln ( 'CD violation(s) detected.' )
    END
  ELSE
    Writeln ( 'CF parse failed.' )
  END
END PascalSyntaxCheck .
```

When run on a DEVIANCE program, this program should either print "`CF parse failed`" or "`CD violation(s) detected`". This information alone would not be helpful to a programmer wanting to correct a mistake, but 1) if the context-free parse fails, the `ParseInput` routine produces a listing which indicates the point at which failure occurred; and 2) the syntax-checker can be modified fairly easily to produce a listing indicating the location and nature of any context-dependent violations.

In the CONFORMANCE class, each program is valid Pascal, and should be accepted by a Pascal processor, but produces different behaviour depending on whether some particular feature of Pascal is correctly implemented by the processor. When the above syntax-checker is run on a CONFORMANCE program, it should print "`No CD violations detected`". Note that if this occurs, it supports the idea that the syntax-checker correctly handles the particular feature being tested, but without much certainty. In general, some ability to execute Pascal programs is necessary to indicate whether the decisions made by the syntax-checker (the relation-instances it establishes) are actually correct, but context-dependent SMSs in general (and the Pascal SMS generated by Ginger in particular) do not have any execution capability[1]. To paraphrase the traditional maxim of testing, CONFORMANCE programs can be used to show the *presence* of errors in a syntax-checker (if it erroneously detects a violation in a valid program), but not their *absence.*

The PVS contains 257 DEVIANCE programs and 212 CONFORMANCE programs. These were all submitted to a Pascal syntax-checker similar to the one presented above. Eventually (after much grammar-revising), it rejected all DEVIANCE programs and accepted all CONFORMANCE programs. Thus, to the extent that the PVS allows validation of mere syntax-checkers, my NURN grammar for Pascal has passed all tests.

[1] Writing a Pascal interpreter (which could actually execute the PVS programs) using the routines of a Pascal SMS would be a good demonstration of the usefulness of the SMS.

REFERENCES

[ANSI83] American National Standards Institute, *American Standard Definition of the Computer Programming Language Pascal*, ANSI/IEEE 770 X3.97-1983, ANSI, New York, 1983.

[ASU86] Aho, A.V., Sethi, R., and Ullman, J.D., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Series in Computer Science, Addison-Wesley, Reading, MA, 1986.

[BahSne86] Bahlke, R. and Snelting, G., "The PSG System: from formal language definitions to interactive programming environments", *ACM Transactions on Programming Languages and Systems*, Vol. 8, No. 4, October 1986, pp. 547-576.

[BjoJon82] Bjorner, D. and Jones, C.B., *Formal Specification and Software Development*, Prentice-Hall International Series in Computer Science, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[Cam87] Cameron, R.D., *Pascal MPS Reference Manual*, School of Computing Science, Simon Fraser University, Burnaby, 1987.

[CamIto84] Cameron, R.D. and Ito, M.R., "Grammar-based definition of metaprogramming systems", *ACM Transactions on Programming Languages and Systems*, Vol. 6, No. 1, January 1984, pp. 20-54.

[Car*89] Cardelli, L., Donahue, J., Glassman, L. Jordan, M., Kalsow, B., and Nelson, G., *Modula-3 Report (revised)*, Technical Report 52, Digital Systems Research Center, Palo Alto, CA, 1989.

[DeRJul80] DeRemer, F. and Jullig, R., "Tree-affix dendrogrammars for languages and compilers", in Jones, N.D. (ed.), *Semantics-Directed Compiler Generation*, Lecture Notes in Computer Science #94, Springer-Verlag, 1980, pp. 300-319.

[DoD83] Department of Defence, *The Programming Language Ada Reference Manual*, ANSI/MIL-STD-1815A-1983, American National Standards Institute, New York, 1983.

[GMN84] Gallaire, Herve, Jack Minker, and Jean-Marie Nicolas, "Logic and Databases: A Deductive Approach", *ACM Computing Surveys*, Vol. 16, No. 2, June 1984, pp. 153-185.

[Knu68] Knuth, D.E., "Semantics of Context-Free Languages", *Mathematical Systems Theory*, Vol. 2, No. 2, 1968, pp. 127-145.

[Lal77] La Londe, W.R., "Regular right part grammars and their parsers", *Communications of the ACM*, Vol. 20, No. 10, October 1977, pp. 731-741.

[MadNor88] Madsen, O.L. and Norgaard, C., An Object-Oriented Metaprogramming System, in Shriver, B. (ed.), *Proceedings of the 21st Annual Hawaii International Conference: Software Track*, IEEE Computer Society, 1988, pp. 406-415.

[Mer87] Merks, E., *Compilation Using Multiple Source-to-Source Stages*, Master's Thesis, Simon Fraser University, Burnaby, 1987.

[Mer88] Merks, E., *AST: An Abstract Symbol Table for Modula-2*, unpublished document, 1988.

[Mer90] Merks, E., *An interactive environment for the programming language Acer*, Ph.D. work in progress, 1990.

[Nau60] Naur, P. (ed.), "Revised Report on the Algorithmic Language ALGOL 60", *Communications of the ACM*, Vol. 6, No. 1, January 1963, pp. 1–17.

[RepTei84] Reps, T., and Teitelbaum, T., "The Synthesizer Generator", *ACM SigPlan Notices*, Vol. 19, No. 5, May 1984, pp. 42–48.

[SteSha86] Sterling, L. and Shapiro, E., *The Art of Prolog*, MIT Press, Cambridge, MA, 1986.

[Ter87] Terry, B., *Grammar-Based File Structure*, Master's Thesis, Simon Fraser University, Burnaby, 1987.

[Wich83] Wichmann, B.A., and Ciechanowicz, Z.J., (eds.), *Pascal Compiler Validation*, John Wiley and Sons, Cichester, 1983.

[Wir85] Wirth, N., *Programming in Modula-2: Third, Corrected Edition*, Springer–Verlag, New York, 1985.

[WinHor89] Winston, P.H. and Horn, B.K.P., *LISP, Third Edition*, Addison–Wesley, Reading, MA, 1989.