CANADIAN THESES

THÈSES CANADIENNES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

THIS DISSERTATION HAS BEEN MICROFILMED EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ MICROFILMÉE TELLE QUE NOUS L'AVONS REÇUE

COMPILATION USING MULTIPLE SOURCE-TO-SOURCE STAGES

by

Eduardus Antonius Theodorus Merks

B.Sc., Simon Fraser University, 1986

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the School

of

Computing Science

SIMON FRASER UNIVERSITY

April 1987

ISBN 0-315-36377-0

APPROVAL

Name: Eduardus Antonius Theodorus Merks

Degree: Master of Science

Title of thesis: Compilation Using Multiple Source-to-Source Stages

Examining Committee:

Chairman: Dr. R. Hobson

Dr. R. Cameron
Senior Supervisor

Dr. L. Hafer
Committee Member

Dr. J. Weinkam
Committee Member

Dr. J. Delgrande
External Examiner

Date Approved: April, 1987

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

Compilation Using Multiple Source-to-Source Stages

Author:

(signature)

Ed A. T. Merks

(name)

April 15/87

(date)

ABSTRACT

Compilers have traditionally been provided as monolithic entities. There is little or no control over the process of compilation since the behaviour of the compiler is completely determined by its fixed implementation. Much of a conventional compiler's inflexibility can be attributed to the technique of translating the high-level language to an inaccessible intermediate form. The proposed alternative is to use transformations which replace complex high-level constructions with equivalent lower-level constructions still in source form. The final transformed program is then directly translated to object code. Thus the source language is also the intermediate language.

A source-to-source compiler is composed of general purpose components which can be used in many other applications. Compiler flexibility is achieved through reconfiguration of the individual transformations. These transformations produce conceptually simple changes and can therefore be reliably combined to produce a customized compiler. A Modula-2 compiler has been constructed to explore increases in functionality, flexibility and reliability.

ACKNOWLEDGEMENTS

I would like to thank Rob Cameron for his guidance and support throughout the work on this thesis. I am indebted to Michael Dyck for his careful reading of this thesis and his invaluable suggestions. I am also grateful to my family and close friends for their encouragement.

TABLE OF CONTENTS

1. INTRODUCTION

1.1 The Need for Flexible Compilation

Modern compiler design techniques [1], which systematically organize the compilation process as a series of relatively independent phases, have greatly simplified the task of compiler implementation. The conventional compiler comprises a data structure called the *symbol table*, which records information about the names used in the source program, and modules which implement the following phases in the compilation process:

- The *lexical analyzer* extracts logical entities known as *tokens* from the character stream representation of the source program.
- The *parser* groups the tokens into syntactic constructs according to the source language's grammar.
- The *intermediate code generator* produces abstract instructions corresponding to the syntactic constructs recognized by the parser.
- The optional *code optimizer* alters the abstract instructions to improve performance characteristics.
- The *object code generator* produces the object code which implements the abstract instructions.

A disadvantage of conventional compilers is the minimal flexibility that is provided. Often only a few compiler options are available for controlling the nature of the object code produced. Optimization may be switched on or off but the actual behaviour of the optimizer is immutable and shrouded in obscurity. Simple language extensions can not be easily integrated into the existing structure. The kinds of abstract instructions produced for a given source language construct are not subject to modification. Alternative object code generators for language retargetting are a major problem. In essence, a compiler is a monolithic black box.

This thesis considers an alternative compiler organization which is designed to achieve certain goals. The first goal concerns the generality of the compiler's components. A compiler is a complex tool with much effort invested in its design and implementation. This effort is best invested into the design and implementation of general purpose components because the initial investment into the components is amortized by their use in many language-based tools. Program editors, analyzers, debuggers and

interpreters are just a few of the language-based tools that can make use of compiler components. Therefore, our first design goal is to construct the compiler out of general purpose components.

The next goal concerns compiler complexity. Despite advances in compiler design techniques, compilers are still extremely complex programs to write. This complexity also makes compiler reliability a problem and makes the translation process difficult to comprehend. Indeed, demonstrating the correctness of compilers is a hard problem of considerable interest [6]. It is desirable for the complexity of conventional compilers to be hidden in the general purpose components we have discussed. Therefore, the second design goal is to conceptually simplify the translation process.

The main goal of this thesis is to make significant user modification of the compiler feasible. The conceptual simplification of the translation process is a prerequisite to this goal since the user cannot modify what he does not understand. The ability to modify the compiler gives a user great freedom: he may extend or restrict the standard semantics of the language, he may customize program optimization and he may control code generation.

Compiler modification has been advocated as a mechanism to allow for the implementation of special purpose abstractions [26]. It is argued that most languages have abstraction mechanisms which are too restrictive. For example, most Algol-like languages have subroutines that take a fixed number of arguments. Such subroutines do not allow for abstractions that are most naturally expressed as subroutines taking a variable number of arguments. Although [26] argues the case for compiler modification, the practicality of such modification in the traditional compiler organization is not fully addressed. The problem of compiler modification is addressed by this thesis.

An examination of the literature reveals that there are many papers concerning the need for language extensions. In [13] an extension to the C programming language for dealing with mutual exclusion is described. In [22] there is a description of a powerful macro processor which can be used to achieve high-level abstractions. The work of [31] deals with how a language can be extended to make the underlying machine architecture visible at the source level. More recently, [16] has advocated a simple language extension to Modula-2 to allow for special kinds of type checking. The need for language extensions is clear from the literature.

The ability to control optimization is another feature that is very useful. A user can design optimizations that are particularly well suited for his style of programming. Users making heavy use of data abstraction

will benefit from the ability to inline-code the abstraction. General purpose routines for inline-coding are possible, but it is often better to have special purpose routines specificly tailored for inline-coding a given abstraction. Such routines can make use of the properties of the abstraction for achieving a particularly efficient result. Compiler flexibility brings the power of the compiler within the reach of users.

The principal cause of inflexibility in the conventional compiler can be traced back to the organization techiques. Conventional compilers are just not designed to be flexible; they are designed to be efficient. The components of a conventional compiler are fine-tuned to work as an efficient whole. The generality of the compiler's components and their usefulness in other applications is not considered.

Modifying the conventional compiler requires knowledge of a vast amount of representational detail. Representational complexity abounds in the compiler. This is seen in the number of different representations that a source program passes through as it is compiled. At various stages a source program is represented as a character stream, a token stream, a syntactic construct and an aggregate of abstract instructions (intermediate code). Significant user modification of the conventional compiler requires great expertise.

1.2 Compiler Flexibility using Source-to-Source Manipulation

This thesis explores an alternative compiler organization which facilitates modification of the translation process. For such modification to be practical, much of the needless complexity must be abstracted away. It must be possible to access the translation process without being subjected to superfluous detail.

We take the perspective that compilers are just *metaprograms* [4], that is, programs about programs. Many of the responsibilities of the conventional compiler can be delegated to a metaprogramming system. The translation process is then implemented in terms of the metaprogramming system's abstraction.

A metaprogramming system typically provides the following facilities:

- A parser which converts the textual representation of a program to an abstract syntactic representation (a parse tree).

- A pretty-printer which produces a textual representation for a given abstract syntactic construction.
- A set of grammar-based manipulation routines.
- A set of semantic manipulation routines for symbol table construction, context-sensitive correctness-checking and other primitive context-sensitive operations (see 2.4).

The DIANA [10] intermediate representation for Ada programs is a good recent example of a metaprogramming system. The representation was originally designed as an intermediate form for the front end of an Ada compiler. Its usefulness has been extended to make it an abstraction which can be used to implement any Ada language-based tool. The use of DIANA for implementing program transformations is discussed in [25].

There is growing recognition of the fact that a common abstract representation, in terms of which all language-based tools are implemented, is necessary (or at the very least beneficial) for the design of a good programming environment [8]. The use of such a common abstraction facilitates the integration of separate tools into a single environment. This thesis will explore the design of such an abstraction and how it may be effectively used to implement a compiler.

With the conventional compiler, the translation process must contend with all manner of representations and erroneous input. However, in a program manipulation environment, translation begins with an abstract representation of a correct program. Translation proceeds with the source language as the intermediate language. The source program is gradually reduced in complexity through transformations which replace high-level constructions with equivalent lower-level constructions. Individual transformations produce conceptually clear changes and are combined in sequence to yield a greatly simplified source program. Many optimizing transformations may be included in the translation process. The final program is directly translated to object code.

With the source-to-source organization, the various tightly integrated components of the conventional compiler are disassociated into individually useful parts. This not only increases the flexibility of the translator, it introduces a whole new functionality. Other applications can use the facilities which are traditionally available only within the compiler. The source-to-source organization results in a conceptually clear translator and a powerful metaprogramming system.

The source-to-source organization does not rule out the use of an intermediate language for code generation; it simply makes it unnecessary for most applications. Using an intermediate language not only gives rise to the conceptual complexity of yet another language with its own syntax and semantics but requires extra implementation work for dealing with the representation of the intermediate form (e.g. another metaprogramming system). An intermediate language will likely be beneficial only when the same intermediate language is to be used for several different compilers.

Using the source language as the intermediate language has many advantages. The system's pretty-printer can be used to monitor the results of individual transformations. The source language is already familiar, so effects of the transformations are readily understood. The correctness-checking mechanism of the metaprogramming system can be applied to intermediate stages to verify that the transformations have not introduced context-sensitive errors. This kind of monitoring and verification increases the reliability of the compilation process as a whole.

There is an educational aspect to the use of the source language for expressing simplifications. Even relatively naive programmers can observe and comprehend the effects of transformations since all they need is an understanding of the source language. Runtime and performance characteristics are more easily observed in the simplified program. This knowledge can be put to use in designing more efficient programs.

The source-to-source organization provides for user-friendly error messages. Error comments are inserted into the parse tree right where the error occurs. The importance of good error messages is considered in [3, 12]. The separation of syntactic and semantic analysis keeps the two kinds of error messages separate. The first few compilations are often used by programmers just to detect syntax errors, so it is useful to be able to interactively correct syntactic errors. With the source-to-source organization, semantic analysis is always performed on a syntactically correct program, so there will never be errors caused by earlier syntactic errors, as is often the case with conventional compilers. The source-to-source organization supports effective error diagnostics.

A correct source program is maintained throughout the source-to-source phase of the translation process. Therefore the results of optimizing transformations are useful even when the program is destined for compilation by a different compiler. The source-to-source transformations are actually tools in themselves.

Program transformation work has been extensively documented in the literature [21]. This work has concentrated in two main areas: program generation from specifications and performance improvement using source-to-source transformations. The work on program generation from specifications is at a much higher-level than the work we are considering in this thesis. We are not concerned with the use of program transformation as a method for deriving an executable program from a formal specification.

The work on source-to-source transformation for program improvement [7, 17, 18, 23] is more relevant to the work of this thesis. As far back as 1974 Knuth discusses the advantages of being able to express program optimizations at the source-level [15]. The work on source-to-source transformation has concentrated on the design of special program transformation tools. However, the work has not resulted in any production systems. The state of the art seems to be that "[current] transformation systems ... must be considered mainly as experimental tools with restricted abilities" [21]. This thesis aims to integrate the use of source-to-source transformations with the organization of the compiler rather than designing a special source-to-source transformation tool.

The problem of source-to-source transformation is addressed at a lower level in this thesis. We are not concerned with the nature of source-to-source transformations tools but with how source-to-source transformations may be conveniently expressed as programs and how they make be effectively used in a compiler. This thesis presents a useful abstraction for expressing program transformations and a compiler organization which makes practical the inclusion of transformations in the translation process.

The source-to-source organization makes way for modification of the translation process by facilitating:

- Extensions to the type-checking facility of the metaprogramming system.
- Reconfiguration of the source-to-source transformations.
- Extensions to the code generator.

Provision is made in the metaprogramming system to allow the metaprogrammer to supply routines for checking the correctness of special-purpose constructs. This permits the metaprogrammer to extend or restrict the standard semantics of the language. The extension is done using procedure parameters so the abstraction supplied by the metaprogramming system is not breached.

The ability to reconfigure the source-to-source transformations is the most obvious advantage of the proposed organization. Metaprogrammers may add, modify or eliminate transformations to suit the application being compiled. The metaprogrammer has complete control over the translation process.

Extensions to the code generator require a little more expertise on the part of the metaprogrammer since a knowledge of the object code is required. However, most extensions are very easy to express and are integrated into the code generator in a straightforward manner. The extensions to the code generator, like those to the metaprogramming system, are made via procedure parameters.

The three facilities for modifying the translator can be used independently or in combination with each other. For example, the metaprogramming system could be extended to allow for a routine that takes an arbitrary number of arguments and the source-to-source transformations could be modified to include a transformation which replaces calls to the routine with standard constructs. In such an instance the final transformed program is portable despite the fact that the original program made use of extensions.

To explore the multiple source-to-source organization proposed in this thesis, a language on which to experiment is needed. Using the source language as an intermediate language will be simplified if a *wide spectrum* language [2] is used. A wide spectrum language is essentially a language which has both high-level constructs and low-level constructs. A systems programming language is a good candidate because the source-to-source process may be easily applied and systems programming, with its need for automated manipulation of large programs, is one of the areas that would benefit most from flexible compilation. **Modula-2** [32] will be the language used for experimentation.

The remainder of this thesis is organized as follows. Chapter 2 deals with the organization of the metaprogramming system. Chapter 3 examines source-to-source transformation and code generation. Chapter 4 investigates the applications of the tools described in Chapters 2 and 3. Chapter 5 concludes the main body of the thesis. Appendix A gives the GRAMPS grammar for Modula-2. Appendix B provides a detailed description of the abstract symbol table. Finally Appendix C briefly describes the code generator. The index at the end of this thesis can be used to find detailed descriptions of the routines that have been implemented.

2. THE METAPROGRAMMING SYSTEM

The formulation of a metaprogramming system is the most important aspect in the design of a source-to-source compiler. Ease and clarity of expression is determined by the nature of the supplied abstraction. We will therefore pay particular attention to the formulation of this aspect of a source-to-source compiler.

A metaprogramming system is the core around which an integrated programming environment can be built. As such, the requirements of the various tools in the environment must be considered in the metaprogramming system's formulation. We do not want to design a system which is good for implementing a compiler but not much else. Conventional compiler organizations are quite adequate for this.

2.1 GRAMPS

The GRAMPS methodology [4] has been used to generate the grammar-based facilities of a metaprogramming system which shall be referred to as MPS. MPS uses Modula-2 as both the host language and the target language. Hence metaprograms about Modula-2 are written in Modula-2. The complete GRAMPS grammar for Modula-2 is given in Appendix A. We shall now provide a brief description of the GRAMPS approach.

The GRAMPS grammatical formalism uses a variant of BNF to specify the syntax of a language. The formalism restricts each production rule to one of four classes:

- construction rules

  <ProcedureCall> ::= <Procedure:Designator> "(" [<Arguments:ExpressionList>] ")"
- repetition rules

  <ExpressionList> ::= <Expression> {"," <Expression>}
- alternation rules

  <Statement> ::= <Assignment> | <ProcedureCall> | <IfStatement> | <CaseStatement> | <WhileLoop> | <RepeatLoop> | <ForLoop> | <SimpleLoop> | <ExitStatement> | <ReturnStatement> | <WithStatement> | <NullStatement>
- lexical rules

  <Identifier> ::= <letter> {<letter> | <digit>}

As we shall describe below, the different grammar rules and classes of rules are the specification for MPS manipulation routines used for selection, recognition, construction and conversion of nodes (abstract parse trees). MPS also provides a parser for constructing a node and a pretty-printer for producing a textual representation for a node. The GRAMPS grammar is the specification of an abstract data type for treating programs as data objects.

The names of the grammar rules form the identifiers of the enumeration type named **NodeClass**. The routine **NodeType** can be applied to a node to return the **NodeClass** identifier giving the rule used in the construction of that node.

A construction rule specifies routines which are used to select the various components of a node of that type. For example, the **ProcedureCall** rule above specifies the selectors **ProcedureOf** and **ArgumentsOf** for selecting respectively the node which is the procedure and the node which is the argument list. The names of the selection routines appear in the GRAMPS grammar.

Nodes constructed according to repetition rules represent lists. The routines **NthElement**, **Next** and **Previous** are of primary concern in the manipulation of lists. **NthElement** applied to a list node returns the node which is the nth element of that list. **Next** or **Previous** applied to a node which is an element of a list returns the next or previous node in that list. Other list operations such as **Append** and **Concat** are also supplied.

Alternation rules are used to specify recognition routines. For example, the **Statement** rule above specifies a recognizer of the form **StatementQ** which returns **TRUE** when applied to a node constructed according one of the alternatives of the rule. All the other grammar rules specify recognizers as well but the recognizers are most useful for alternations since the routine **NodeType** can be easily used for recognition purposes in the other cases. The special recognizer **EmptyQ** is used for recognizing empty nodes (the missing optional parts of construction rules).

Nodes constructed according to lexical rules form the leaves of the parse tree. A lexical rule specifies a routine for coercing a node of that type to a character-by-character representation. For example, the **Identifier** rule above specifies a routine of the form **CoerceIdentifier** which returns the **StringType** value giving the identifier's spelling.

MPS provides various editing routines such as **Insert**, **Replace**, **Delete** and **Exchange** for modifying existing nodes. Many other utilities [5] are also supplied but are too numerous to mention here.

The GRAMPS grammar for Modula-2 does deviate from the standard grammar. The change to the standard syntax was considered very carefully and is intended to make program manipulation more convenient. The syntax of qualified identifiers has been modified. The standard syntax uses a "." to separate the individual identifiers of a qualified identifier. This is the same syntax used for separating a field from its record, creating an ambiguity between designators and qualified identifiers. The ambiguity serves only to complicate program manipulation, so an "@" is used for the syntax of qualidents. The change to the standard syntax allows context-free parsing into meaningful constructs. It is a very superficial change and proves quite beneficial.

2.2 Context-Sensitive Considerations

In the previous section we examined how the GRAMPS methodology is used to specify the core grammar-based facilities of MPS. Being based on a context-free grammar, the GRAMPS methodology does not deal with the context-sensitive† properties of a language. A node is guaranteed to be syntactically correct but this says nothing about whether it satisfies context-sensitive constraints such as type restrictions and declaration requirements. The remainder of this chapter will examine the issues involved in augmenting the grammar-based facilities with routines which facilitate context-sensitive manipulation.

The following are typical questions which we may ask about a node:

- Where is this name declared?
- What is the type of this expression?

  Is this expression a constant and what is the value of that constant?
- Is this statement context-sensitively correct?

In general, the above questions are extremely difficult to answer. Many program transformations rely heavily on the answers to these questions, so a complete metaprogramming system must deal with these requirements.

Conventional compiler design makes use of a symbol table. The same kind of information that is traditionally available in a symbol table must be available in MPS. An abstract formulation of a symbol table is needed.

† We use the term *context-sensitive* loosely to mean context-dependent or semantic The term has a more formal meaning in language theory.

The road to an abstract symbol table diverges in two directions. The symbol table information for a program can either be recalculated each time it is needed or it can be precalculated and saved. Each technique has its advantages and disadvantages.

Recalculating symbol table information each time has the advantage that changes to the program will always be reflected in subsequent calls to the symbol table routines, whereas the information in a precalculated symbol table may be invalidated by subsequent modification of the program. The disadvantage of repeated recalculation of information is that it may become very costly. However, regenerating the entire symbol table every time a change in the program is made will also lead to inefficiency if the program is changed frequently.

The deciding factor which favours the use of a precalculated symbol table is the fact that in many situations the determination of a single piece of symbol table information can be as expensive as the generation of an entire symbol table. Consider the case in which the size of a record type is to be determined. The size of a record type depends on the sizes of the types of its fields which may in turn depend on the sizes of many other types. Thus the size of every type declared in the program may need to be determined for one operation.

It also turns out that for Modula-2 the symbol table does not need to be regenerated every time a change is made in the source program. The contents of the symbol table depend only on the declarations of the program; changes made to the statements of a program have no effect on the symbol table. There are a great many metaprogramming applications which use only statement-level modification.

Further requirements for regeneration of the symbol table can be alleviated if common modifications of the declaration parts of a program can be made with routines which incrementally modify both the source program and the symbol table. Such routines have been supplied (see 2.4) and the source-to-source compiler implemented in terms of these routines requires no symbol table regeneration.

2.3 Context-Sensitive Restrictions

In the design of a symbol table abstraction for MPS we have carefully explored the properties of the Modula-2 language. This exploration has revealed several problems areas which complicate both the design of the symbol table abstraction, and program transformation in general. To alleviate this problem we have placed two restrictions on the context-sensitive properties of Modula-2. Careful use of restrictions can be very helpful in making programs easier to manipulate, without affecting the correctness of most programs.

A restriction has been placed on the reuse of names in nested scopes:

- A visible name may be redefined only if it denotes a variable or parameter and is redefined to denote another variable or parameter.

A stronger restriction would be to forbid the redefinition of visible names entirely but this would be upsetting to those of us who like to use variables such as "i", "j" and "k" as loop control variables and don't want to come up with different names in nested scopes.

Very few programmers will ever violate the restriction on the reuse of names even if they are not aware of the restriction. It is simply poor programming practice to have an name denote a type at one level and then denote a different type (or something else entirely) at a deeper level. With the restriction, the visibility of the names of constants, types, procedures and modules cannot be hidden by renaming. This allows the metaprogrammer to make simplifying assumptions about the visibility of these objects. The restriction also prevents the renaming of standard identifiers (another bad practice).

Another benefit of the redefinition restriction is the prevention of the kind of ambiguity demonstrated by the following program:

```
MODULE M;

  CONST
    T = 2;

  PROCEDURE P ();

    CONST
      C = T + 2;

    TYPE
      T = POINTER TO INTEGER;
```

```
  BEGIN
  END P;
END M.
```

The reuse of T in the above example shows the confusion that can result from reusing names in nested scopes. It is likely that most compilers will not mark the use of T in C = T + 2 as an error. When the compiler processes the declaration for C the name T will denote a valid constant. However, later on in the procedure P, the name T is redefined to denote a type. Thus either T denotes two things in the same scope or T is used before it is defined.

If situations as in the above example are prevented the metaprogrammer can be assured that a name denotes only one thing in a given scope. Unfortunately, allowing the renaming of variables or parameters can still give rise to a situation like the one above, as in the following example:

```
MODULE M;

  VAR
    X : INTEGER;

  PROCEDURE P();

    CONST
      C = SIZE(X);

    VAR
      X : CHAR;

  BEGIN
  END P;
END M.
```

A variable or parameter can appear in the context of a declaration only as a constant expression. The only expression which yields a constant value from a variable or parameter is the standard procedure SIZE. The abstract symbol table easily checks all calls to SIZE for use-before-definition errors. However, with the redefinition restriction it is not necessary to check all names used in all declarations for possible redeclaration later. The redefinition restriction increases the efficiency of the abstract symbol table, helps to ensure that programs will be more readable and makes programs easier to manipulate.

The second restriction is a restriction on the type of an expression:

- No expression, except for an expression which is a procedure name, may have a type which is declared textually later in the program.

The restriction does not apply to procedure names because this would prevent mutual recursion (Modula-2 does not have forward procedure declarations like Pascal). The following example demonstrates a violation of this restriction:

```
MODULE M;

  PROCEDURE P();

  BEGIN
    W := X + Y * Z;
  END P;

  TYPE
    SetType = SET OF (Red,Yellow,Blue);

  VAR
    W,X,Y,Z : SetType;

END M.
```

The use of the variables `W`, `X`, `Y` and `Z` are all erroneous according the expression-type restriction. The type name `SetType` is visible in the statements of the procedure `P` but is not visible in the declarations of `P`. Therefore, declaring a temporary variable for the expression `Y * Z` would require the following kind of transformation:

```
MODULE M;

  PROCEDURE P();

    TYPE
      DummyType : ARRAY [0..3] OF WORD;

    VAR
      Temporary : DummyType;

  BEGIN
    Temporary := DummyType(Y * Z);
    W := X + SetType(Temporary);
  END P;

  TYPE
```

```
    SetType = SET OF (Red,Yellow,Blue);

  VAR
    W,X,Y,Z : SetType;

END M.
```

Although situations requiring complicated type transfers rarely occur, they must be dealt with by a complete transformation system. With the expression-type restriction, the situation can never occur because the type of an expression will be declared textually before the statement in which the expression is used and can always be made visible to declare a variable of that type. Therefore a temporary of the correct type can be generated for all expressions.

The expression-type restriction does not affect the power of Modula-2. The only place that problems like the one above can occur is in the body of a procedure. Modules can only import objects already declared, so that in the body of a module the types of all expressions are necessarily declared textually earlier in the program. If the body of a procedure must make use of types declared later in the program, the body can always be replaced with a call to a procedure which is itself declared later in the program. The expression-type restriction does not restrict the kinds of algorithms that can be expressed, but it does make transformations much easier to express.

Context-sensitive restrictions can be very helpful for enforcing useful properties on programs. With careful use, they can make program manipulation cleaner without affecting the correctness of most programs.

2.4 The Abstract Symbol Table

We shall proceed with an overview of the symbol table abstraction that has been implemented for MPS. The aim of this section is to provide the reader with an intuition as to what can be done by the routines of the abstract symbol table. A more detailed description of the abstraction is given in Appendix B. The following routines will be briefly described:

```
DEFINITION MODULE Symbol;
 TYPE
   Scope;
   Recognize   = PROCEDURE (Node, Scope) : BOOLEAN;
   TypeChecker = PROCEDURE (Node, Scope) : Node;
```

```
  Checker      = PROCEDURE (Node, Scope);

 PROCEDURE Declare (x : Node) : BOOLEAN;
 PROCEDURE CorrectQ (x : Node; s : Scope) : BOOLEAN;
 PROCEDURE Error (Location : Node; Comment : StringType);
 PROCEDURE ClearSymbolTable ();
 PROCEDURE UserExpressionCheck (p1 : Recognize; p2 : TypeChecker);
 PROCEDURE UserStatementCheck (p1 : Recognize; p2 : Checker);

 PROCEDURE ScopeHandle (x : Node) : Scope;
 PROCEDURE DefiningOccurrence (x : Node; s : Scope) : Node;
 PROCEDURE ConstantQ (x : Node; s : Scope) : BOOLEAN;
 PROCEDURE GetType (x : Node; s : Scope) : Node;
 PROCEDURE QualifyingDesignator (x : Node; s : Scope) : Node;
 PROCEDURE GenerateName (x : StringType) : Node;

 PROCEDURE Rename (x,y : Node);
 PROCEDURE GetVisibleName (x : Node; s : Scope) : Node;
 PROCEDURE GetTypeName (x : Node; s : Scope) : Node;
 PROCEDURE ExpressConstant (x : Node; s : Scope) : Node;
END Symbol.
```

The routines are divided into three classes: those for building the symbol table and checking correctness, those for information lookup and analysis, and those for supporting program manipulation. We will discuss each class in turn.

2.4.1 Symbol Table Building and Correctness Checking

Building the symbol table and checking the correctness of a program are the most complex parts of a conventional compiler. The routines in this section provide a simple interface which hides this complexity.

The procedure **Declare**, applied to a compilation unit, builds the corresponding symbol table and checks the correctness of all declarations. The correctness of statements and expressions may be determined by applying **CorrectQ**. Errors encountered by **Declare** and **CorrectQ** are marked with comments using **Error**. The symbol table may be reinitialized to its startup state using **ClearSymbolTable**.

Declare may also be used to incrementally add new declarations. For example, to add the declaration of a variable the new declaration is inserted in the appropriate place in the declaration list and **Declare** is applied to install the information in the symbol table. Most program manipulations which modify

declarations simply add new declarations, rather than modify existing declarations, so the incremental feature of **Declare** eliminates much of the need for symbol table regeneration.

Declare only checks the correctness of the declarations of the node being processed. The correctness of the statements is determined separately. This is a reflection of the fact that the correctness of the declarations does not depend of the correctness of the statements. Such a division of correctness checking also makes the symbol table more useful in conjunction with a syntactic editor. Most of the time when we are editting programs, we are modifying statements and adding (not modifying) declarations. Thus a symbol table can be maintained as we edit programs making it possible to supply the abstract symbol table routines as editor commands.

To facilitate metaprogrammer extensions to the correctness checking capabilities of MPS, the procedures **UserExpressionCheck** and **UserStatementCheck** have been supplied. These procedures are used to install metaprogrammer-supplied checking routines into MPS. Both extension facilities follow the same basic organization.

To create an extension, the metaprogrammer must supply two procedures. The first procedure is a recognition procedure which takes a node and its scope as arguments and returns **TRUE** if the extension applies to that node. The second routine is called with the same arguments as the first and is called only if the first routine returns **TRUE**. The second routine is responsible for checking the correctness of the node; for expressions it must also return the type name of the node if the node is correct. The checking routine must call the procedure **Error** for any errors that are encountered.

The extensions written by the metaprogrammer are installed in MPS by calling either **UserExpressionCheck** for extensions to expressions, or **UserStatementCheck** for extensions to statements. Whenever MPS needs the type of an expression or checks the correctness of a node, the metaprogrammer supplied recognizers are called in the order in which they were installed. The standard checks are performed only if no extension applies. Section 4.3 shows an application which makes use of these extension mechanisms.

2.4.2 Information Lookup and Analysis

The symbol table building routines of the previous subsection encode program properties in a separate data structure. This information may be accessed using the routines described in this section.

ScopeHandle is used to determine the names that are visible at a given point in a program. It is applied to a node and returns a value of type **Scope** summarizing name-visibility information. Many of the other symbol table routines take a **Scope** argument because the information they supply, or operation they perform, is dependent on name visibility. It is useful to precalculate scope information because it is used repeatedly and allows context-sensitive information about a node to be determined as if it were in a context to which it is not attached.

The meaning of a name is determined by the declaration which defines the name. When we are manipulating programs it is often necessary to find the declaration of a name so that we may determine what kind of object it represents and deal with it accordingly. The procedure **DefiningOccurrence** is supplied to meet this need. It is applied to a name and a scope and returns the identifier in the context of the declaration which defines the name.

In Modula-2 it is very difficult to syntactically determine if an expression denotes a constant. Often in program manipulation, a constant expression must be dealt with as a separate case since such an expression is evaluated at compile-time rather that runtime. Therefore, we have designed the routine **ConstantQ** which determines whether or not a given expression represents a constant.

In Modula-2, all expressions have type. Access to this property of the language is made through the routine **GetType** which is applied to an expression and returns the node which defines the type of this expression. The details of **GetType** are too involved to deal with here but we are all familiar with how a type declaration in a language is used to define a new type. It is the purpose of **GetType** to return such a type declaration.

The routine **QualifyingDesignator** is supplied to deal with the implicit qualification of fields as a result of the with-statement construct. **QualifyingDesignator**, applied to a name denoting a field of a record, returns the designator denoting the record of which the field is a component. Thus if we apply **QualifyingDesignator** to a field which is implicitly qualified by a with-statement, it will return the designator of that with-statement.

The final utility we shall deal with in this subsection is **GenerateName**. There are many occasions when program transformations require the generation of names which will not conflict with existing names. The routine **GenerateName** is given a **StringType** value giving the desired spelling of the new identifier and returns an identifier which is guaranteed to be different from any identifier already declared. The name is made unique by appending a larger trailing number than appears in any declared identifier with the same prefix. For example if the names **a** and **a4** are already declared then **GenerateName(Str("a"))** will return an identifier with the spelling a5.

2.4.3 Manipulation Support

The routines in this subsection are characterized by the fact that they are used to modify both the symbol table and the corresponding program so as to eliminate the need for symbol table regeneration. Metaprogrammers making use of these routines must be aware of the kinds of changes that may result in their programs. We will give examples demonstrating these changes.

First of all, we have a renaming utility. The procedure **Rename** is applied to an identifier which is a defining occurrence and the identifier which is to be the new name, and renames all occurrences of the identifier, in its enclosing compilation unit, to the new name. The concept of renaming is very simple and it should be intuitively clear to anyone familiar with a programming language.

The utility **GetVisibleName** is a very powerful facility for making a name visible in a scope where it is not already visible. Many program transformations such as inline-coding the two routines we shall describe subsequently, require the capability of making a name visible. The following example demonstrates the kinds of changes that **GetVisibleName** may make to a program:

```
MODULE m;

  MODULE m1;

    MODULE m2;
      EXPORT QUALIFIED T;

      TYPE
        T = CHAR;
    END m2;

    VAR
      x : m2.T;
  END m1;
```

```
  MODULE m3;
    VAR
      m1,m2 : INTEGER;
  END m3;
END m.
```

Suppose that in the above example that the declaration of T in the module m2 is to be made visible in the module m3 using **GetVisibleName**. Making T visible in m3 requires the following changes:

```
MODULE m;

  MODULE m1;
    EXPORT m4;

    MODULE m4;
      EXPORT QUALIFIED T;

      TYPE
        T = CHAR;
    END m4;

    VAR
      x : m4@T;
  END m1;

  MODULE m3
  (* The T in "T = CHAR" is now visible as m4@T. *);
    IMPORT m4;
    VAR
      m1,m2 : INTEGER;
  END m3;
END m.
```

The above example demonstrates several important changes that **GetVisibleName** may produce. The module m2 had to be exported out of m1 in order to export m2@T. Importing the module m2 into m3 creates a conflict so m2 is renamed to the generated name m4.

The next example illustrates conflicting names from two definition modules requiring the qualification of one of the names to resolve the conflict:

```
MODULE m;
  FROM d1 IMPORT x,y,z;
  IMPORT d2;

  MODULE m1
    IMPORT y,z;
    FROM d2 IMPORT x;

    MODULE m2;
      IMPORT x;

    BEGIN
      DEC(x)
    END m2;

  BEGIN
    x := y + z
  END m1;

BEGIN
  x := y + z
END m.
```

Suppose that in the above program we wish to make the **x** from the definition module **d1** visible in the module **m1**. The following program will result:

```
MODULE m;
  FROM d1 IMPORT x,y,z;
  IMPORT d2;

  MODULE m1;
    IMPORT x,y,z;
    IMPORT d2;

    MODULE m2;
      FROM d2 IMPORT x;

    BEGIN
      DEC(x)
    END m2;

  BEGIN
    d2@x := y + z
  END m1;

BEGIN
  x := y + z
END m.
```

Making the **x** from **d1** visible in **m1** creates a conflict because the **x** from the definition module **d2** is already visible there. Therefore, the **x** from **d2** is changed to make it visible as the qualified identifier **d2@x**, and the **x** from **d1** imported into **m1**. Notice that the import to the module **m2** is also changed but that its body remains the same.

GetVisibleName has the potential for creating a large number of changes in the program being manipulated but most of these changes will occur only in programs with a lot of name duplication.

The routine **GetTypeName** is very similar to **GetType** described in 2.4.2. As is suggested by its name, **GetTypeName** returns the name of the type rather than the type itself. This type name can be used to declare a variable of that type and is therefore useful in the generation of temporary variables for subexpressions. If the type does not have a name, a name is automatically generated and the program is modified to include the declaration associating the new name with the type. For example:

```
MODULE m;

  TYPE
   ColourSet = SET OF (Red, Green, Blue);

  VAR
   y : ColourSet;

BEGIN
 y := ColourSet {Red}
END m.
```

If we applied **GetTypeName** to the enumeration identifier **Red** in the body of the above module the following program results:

```
MODULE m;

  TYPE
   Type0 = (Red, Green, Blue);
   ColourSet = SET OF Type0;

  VAR
   y : ColourSet;

BEGIN
```

```
  y := ColourSet {Red}
END m.
```

The above example demonstrates how **GetTypeName** automatically generates a unique type name. The other modification that **GetTypeName** may produce occurs when the type name is not visible where it is needed in which case **GetVisibleName** is used to make it visible.

The last routine we shall deal with is **ExpressConstant**. **ExpressConstant** is used to evaluate constant expressions. Like **GetTypeName**, **ExpressConstant** may modify the program in order to make the result of the evaluation of a constant expression visible where it is needed. For example:

```
MODULE m1;

  MODULE m2;
    EXPORT SetType,SetConstant;

    TYPE
      SetType = SET OF (Red,Green,Blue);

    CONST
      SetConstant = SetType {Red};
  END m2;

  VAR
    x : SetType;

BEGIN
  x := SetConstant
END m1.
```

Suppose that in the above program we with to express the value of **SetConstant** in the body of **m1** The following would be the result applying **ExpressConstant**:

```
MODULE m1;

  MODULE m2;
    EXPORT SetType,SetConstant,Type0;

    TYPE
      Type0 = (Red,Green,Blue);
      SetType = SET OF Type0;

    CONST
```

```
    SetConstant = SetType {Red};

  END m2;

  VAR
    x : SetType;

BEGIN
  x := SetConstant
  (* "SetConstant" may now be replaced with the result of
     ExpressConstant, namely "SetType {Red}". *)
END m1.
```

In the above example, **SetType** and **Red** were both required to be visible in the body of **m1**. To make **Red** visible, its corresponding enumeration type had to be given a name so that it could be exported out of **m2**, thereby also exporting **Red** itself. **ExpressConstant** uses **GetVisibleName** to make required names visible.

2.5 Generality

A metaprogramming system can best be thought of as an abstract data type. The concrete representation that actually implements the abstraction remains hidden from the user so that different realizations can be implemented. The generality and usefulness of implementing language-based tools in terms of an abstraction is well recognized [8,19].

The DIANA representation for Ada programs serves as a good basis of comparison with the MPS representation for Modula-2 programs. DIANA includes both an abstract syntax tree that results from a parse of an Ada program, and an attributed parse tree which is derived from the abstract syntax tree through static semantic analysis. As such, DIANA is essentially an intermediate representation of an Ada program than rather simply a representation of an Ada program. However, the source representation can be reconstructed from the DIANA representation.

MPS includes only an abstract syntax tree representation. Static semantic analysis simply encodes information which can be accessed using a small set of routines. The representation of source programs is therefore not tightly bound with various semantic attributes. This eliminates problems with confusing mappings between source constructs and their intermediate representation. The GRAMPS approach emphasizes the importance of the source program.

It will be clear to anyone reading the DIANA reference manual that the sheer complexity of the representation makes it a tool for experts only. The GRAMPS approach is in sharp contrast with this. The use of a simple grammatical formalism for specifying the bulk of MPS results in a system which is both easy to document and simple to use. The abstract symbol table described in this chapter result in a metaprogramming system which rivals the power of DIANA.

In all fairness to the designers of DIANA, the complexity of DIANA is in some respects due to the complex nature of Ada itself. Modula-2 is a much simpler language so a metaprogramming system for Modula-2 is inherently simpler than one for Ada. With all arguments about the best form of a metaprogramming system aside both DIANA and MPS are good paradigms demonstrating the value of metaprogramming systems for implementing not only compilers but all language-based tools.

3. THE TRANSLATOR

Source-to-source transformations will be used to assist in the translation from a high-level language to a low-level language. The transformations deal with the substitution of high-level constructs with equivalent lower-level constructs. The elimination of difficult-to-translate constructs simplifies code generation.

For the purpose of this thesis, source-to-source transformations are used to translate to a low-level language, but the technique applies equally well for the translation to a high-level language [9, 27]. Transformations can be used to reduce the dissimilarity of the source language and target language by eliminating constructs in the source program which cannot be directly translated. There is nothing inherent in the technique dictating that the target language be a low-level language.

Source-to-source transformations can be used to translate Modula-2 to Pascal or C. In a translation from Modula-2 to Pascal there would be a transformation for eliminating local modules. In a translation from Modula-2 to C there would also be transformations for eliminating with-statements and nested procedures. Simplifying the source program prior to translation is very effective for dealing with difficult-to-translate constructs.

The translation process is partitioned into a source-to-source phase and a code generation phase. The division between these two phases is very flexible and is determined by pragmatic concerns. Often there is a choice between eliminating a construct at the source level or extending the code generator to deal with the construct in its unsimplified form. The decisions are guided by the need to increase the efficiency of the generated object code and the need to reduce the overall complexity of the translator.

3.1 Code Generation

We shall proceed with a discussion of the code generator and the decisions that went into its formulation. Appendix C provides further detail about the code generator. The code generator accepts a subset of Modula-2 as input. The nature of this subset determines the transformations required to simplify standard Modula-2 programs. These simplifying transformations will be described in the subsequent section. The code generator is designed to produce MC68020 assembler code for a SUN-3 running under the UNIX operating system [20, 28].

The first decision in the formulation of the code generator was the decision to make the calling convention for subroutines compatible with that of the existing Berkley Pascal compiler. This decision has led to a Modula-2 compiler that produces object code which can be freely linked not only with Pascal routines but C routines as well. It seems only good sense to make the compiler compatible with the existing tools provided by UNIX to avoid re-invention of the wheel.

It was decided that the code generator should accept designators in an almost unrestricted form. The MC68020 has powerful addressing modes which can be used to efficiently implement the various address calculations required for designators. It is possible to eliminate indexed-variables in the source program, but the MC68020 has indexed addressing modes which are much more efficient than source-level address calculations. Another advantage of preserving the form of designators is that transformed programs will be much more readable than they would with designators fragmented into less meaningful address calculations. The only restriction on designators is that the index of an indexed-variable must be a constant or a (possibly type transferred) designator containing no non-constant indices. This restriction eliminates the need for the code generator to generate temporaries.

The use of a type name as the function of a function call is referred to as a type transfer. Type transfer is representation-dependent and operates much like the overlaying of tagless variant records. The code generator expects the argument of a type transfer function to be a designator (rather than a general expression). This allows type transfers to be treated as designators.

There are several restrictions which apply to expressions. Brackets which are not syntactically required are not accepted. The unary "+" (being an identity operation) is not permitted. The short circuit operators "AND" and "OR" are not accepted since their evaluation method deviates from the normal evaluation method of the other dyadic operators. Set-factors must be constant since generating code for set-factors containing variables gets too complicated. And finally, the second operand of dyadic expressions must be a designator or constant rather than a more general expression. Like the restriction on designators the final restriction eliminates the need for the generation of temporaries by the code generator.

The calling convention for functions requires them to return their result in a register. Therefore, the code generator does not accept functions which return values which are too large to fit in a register. In the next section, we shall see that this restriction to the code generator need not apply to the original

unsimplified Modula-2 program.

The code generator does not accept all the various kinds of statements. Only assignments, procedure-calls, single-clause if-statements, simple-loops, exit-statements, return-statements and null-statements are accepted. Thus for-loops, repeat-loops, while-loops, case-statements and with-statements are not accepted.

Appendix C describes the routines supplied for code generation. The code generator is invoked by a call to **Modula2ToSunAssembler** which takes a program module and an output file as arguments, and produces the program module's assembly language equivalent. The remaining code generation routines can be used by the metaprogrammer to add extensions to the code generator.

Four procedures are provided for sending assembler instructions to the output file. The procedures **AssemblerLabel**, **Assembler1**, **Assembler2** and **Assembler3** are used respectively to send a label, a zero operand instruction, a single operand instruction and a two operand instruction to the output file.

The procedure **Op** is given a constant or (possibly type transferred) designator and a scope as arguments and returns the effective address specification which can be used as the operand of an assembler instruction. **Op** may send address calculation instructions to the output file as a side-effect.

The procedure **Gen** is given an expression or statement and a scope and produces the instructions necessary to evaluate the expression or to execute the statement. Expressions leave their results in a register and thus behave very much like statements.

The metaprogrammer may extend the code generator using a technique analogous to the extension of the metaprogramming system. The procedures **UserGen** and **UserOp** are used to extend the domain of **Gen** and **Op** respectively. **UserGen** takes two procedures as parameters. The first procedure is a recognition routine which returns **TRUE** if the extension applies for the given node at the given scope. The second routine is called to generate the appropriate instructions only if the first routine returns **TRUE**. For **UserOp** the second routine must also return a **StringType** result giving the effective address specification of its argument. An example of a code generator extension is given in 4.5.

3.2 Source-to-Source Transformation

We have briefly examined the code generator and the restricted form of Modula-2 that it accepts as input. Source-to-source transformations are used enforce the required restrictions on standard Modula-2 programs. This section will describe these transformations. We will first examine the transformations for simplifying statements and then move on to deal with the transformations for simplifying expressions.

3.2.1 Statement-Oriented Transformations

This subsection deals with the transformations for eliminating or simplifying various kinds of statements. The first three statement transformations are purely syntactic; they do not rely on context-sensitive information. There is a common motivation for their application. Each of the statements being dealt with contains a boolean expression which may need simplification later in the translation process. This often gives rise to temporary variables for subexpressions. There are problems with where these temporaries must be placed. After the application of the following three transformations, subsequent transformations may assume that temporaries can always be placed before the enclosing statement of the expression being simplified.

- **`PROCEDURE WhileLoopToSimpleLoop (x : Node);`**

  This procedure replaces the while-loop given by `x` with a simple-loop.

At first glance it may seem somewhat pointless to remove while-loops since they are already so simple. However, temporaries generated for the condition of a while-loop must be placed both before the while-loop itself and at the end of the while-loop body. To eliminate this kind of undesirable code duplication, we perform the stated transformation:

```
WHILE Condition DO                    LOOP
  Body()                    ==>       IF NOT (Condition) THEN
END                                     EXIT
                                      END;
                                      Body()
                                    END
```

- brackets are only put around Condition if necessary

- **PROCEDURE RepeatLoopToSimpleLoop (x : Node);**

  This procedure replaces the repeat-loop given by **x** with an equivalent simple-loop.

Simplifying the condition of a repeat-loop requires the generation of temporaries which must be placed at the end of the loop body. This makes the condition of a repeat-loop an inconvenient special case for expression simplification. The transformation does the following:

```
REPEAT                              LOOP
  Body()                    ==>       Body();
 UNTIL Condition                      IF Condition THEN
                                        EXIT
                                       END
                                     END
```

- **PROCEDURE IfStatementToSimpleIfStatement (x : Node);**

  This procedure simplifies the if-statement given by **x** by replacing multiple clauses with equivalent nested single-clause if-statements.

The reason for the elimination of multiple clauses is to make it possible to generate temporaries for all clause conditions. There is no valid place to put the temporaries generated for the condition of a clause (other than the first clause) without breaking up the clause list into separate if-statements. The transformation does the following:

```
                                    IF c1 THEN s1()
IF c1 THEN s1()                      ELSE
 ELSIF c2 THEN s2()          ==>      IF c2 THEN s2()
 ELSIF c3 THEN s3()                    ELSE
 ELSE s4()                              IF c3 THEN s3() ELSE s4() END
 END                                   END
                                     END
```

The next three transformations depend on context-sensitive information and so also take a scope argument.

- **PROCEDURE WithStatementToStatementList (x : Node; s : Scope);**

  This procedure replaces the with-statement with an equivalent statement-list. The address of the designator of the with-statement may be saved in a temporary variable if the designator contains non-constant indices or reference-variables. In the body of the with-statement, implicitly qualified fields are explicitly qualified making the with-statement unnecessary.

With-statements are annoying to deal with when transforming programs since they introduce a new scope that must be passed along when traversing statements. With the elimination of with-statements it is assured that all designators will be fully qualified. The following demonstrates the effect of the transformation:

```
WITH a^.f1 DO                          Temporary := ADR(a^.f1);
  f2 := 10;              ==>           Temporary^.f2 := 10;
  f3 := x                              Temporary^.f3 := x;
 END;

- f2 and f3 are fields of a^.f1
- Temporary is "Declare"d to be a POINTER TO the a value of the
  type possessed by a^.f1
```

- `PROCEDURE ForLoopToSimpleLoop (x : Node; s : Scope);`

This procedure replaces the for-loop given by `x` with an equivalent simple-loop.

The for-loop transformation is very tricky because for-loops must behave correctly when the final value of the loop control variable is the maximum value of its type. Thus the following straightforward transformation is is efficient but not correct:

```
                                       Temporary := e2;
                                       i := e1;
FOR i := e1 TO e2 BY c DO     ==>      LOOP
  Body()                                 IF i > Temporary THEN EXIT END;
 END                                     Body();
                                         INC(i,c)
                                       END
```

In the above solution the incrementing of `i` may result in a value for `i` which is out of range for a variable of that type. The following demonstrates how `ForLoopToSimpleLoop` transforms a for-loop:

```
- there are two main cases of the transformation

- if c > 0
      •                                  t1 := e1;
                                         t2 := ORD(e2);
FOR i := e1 TO e2 BY c DO     ==>        IF ORD(t1) <= t2 THEN
  Body()                                 i := t1;
 END                                     LOOP
                                           Body();
                                           IF t2 - ORD(i) < c THEN
```

```
                                                    EXIT
                                                   END;
                                                  INC(i,c)
                                                 END
                                                END
```

```
- if c < 0
                                                t1 := e1;
                                                t2 := ORD(e2);
 FOR i := e1 TO e2 BY c DO          ==>         IF ORD(t1) >= t2 THEN
   Body()                                         i := t1;
  END                                             LOOP
                                                    Body();
                                                    IF ORD(i) - t2 < -c THEN
                                                      EXIT
                                                    END;
                                                   DEC(i,-c)
                                                 END
                                                END
```

```
- t1 is "Declare"d with the type possessed by i
- t2 is "Declare"d with the type CARDINAL
- if the variable i is of type CARDINAL or subrange thereof then the
  calls to ORD are replaced with the argument itself
- if e1 or e2 are constants, they are not assigned to temporaries
  but are used everywhere directly
- the exit condition is changed to ORD(i) = t2 if c is either 1 or -1
```

In the above example, it is important to remember the interpretation that is placed on the application of ORD to INTEGER operands: ORD(MIN(INTEGER)) = 0.

- PROCEDURE CaseStatementToIfStatement (x : Node; s : Scope);

  This procedure replaces the case-statement given by x with a straightforward if-statement.

Since it is difficult to generate code for case-statements the easiest approach is to replace them with if-statements, although this may have its price in program efficiency. The transformation has the following effect:

```
CASE a[i] OF                                t := a[i];
  'a'..'z': s1()                            IF (t ≥ 'a') AND (t ≤ 'z') THEN s1()
  'A'..'Z': s2()          ==>                ELSE IF (t ≥ 'A') AND (t ≤ 'Z') THEN s2()
  '0','1' : s3()                             ELSE IF (t = '0') OR (t = '1') THEN s3()
 ELSE                                        ELSE
  s4()                                        s4()
 END                                         END
```

If multiple-clause if-statements are not desirable, `IfStatementToSimpleIfStatement` may be applied to the if-statement resulting from the above transformation. There are many alternative ways of producing an if-statement from a case-statement. The approach taken here is the most simple approach but an if-statement with a binary branching pattern rather than a linear pattern is also possible.

3.2.2 Expression-Oriented Transformations

This subsection deals with the transformations for eliminating or simplifying various kinds of expressions. The first transformation is purely syntactic and the remaining four depend on context-sensitive information.

- **PROCEDURE** `ListedIndicesToRecursiveIndices (x : Node);`

  This procedure replaces listed array subscripts with recursive subscripts.

Listed indices are eliminated to simplify the application of recursive traversals. When expressions are broken down into simpler subexpressions it may be necessary to break up a list of array subscripts. For this reason having the subscripts already separated makes subsequent transformations easier to express. Since many of the utilities used by the transformations described in this section ignore all but the first subscript of indexed variables, it is important to apply this transformation first. The transformation has the following effect:

```
x[i,j,k]     ==>     x[i][j][k]
```

- **PROCEDURE** `SetFactorSimplification (x : Node; s : Scope);`

  This procedure is used to eliminate non-constant set-factors. The node `x` is an assignment with the expression being a set-factor which is not a constant. The assignment is broken down so that a variable is assigned a set factor consisting of only the constant terms; then the variable terms are added to the set using `INCL`.

This transformation is a handy way to eliminate the concern with set-factors containing variables. The code generator may therefore assume that all set factors are constant (which is the way Modula-2 was originally specified). The transformation has the following effect:

```
                                    t := BITSET {0};
t := BITSET {0, a..b, c}    ==>     FOR i := a TO b DO INCL(t, i) END;
                                    INCL(t, c)
```

It must be pointed out that the above transformation assumes that the value of the set-factor does not depend on t and that the evaluation of t does not produce side-effects. As can be seen in the example, the value of t is modified when the constant part of the set-factor is assigned to it. If it cannot be assured that the set-factor does not depend on the value of t, then the set-factor can be assigned to a temporary and the transformation can then be applied to that assignment.

- **PROCEDURE BooleanSimplification (x : Node; s : Scope);**

  This procedure eliminates the use of the short-circuit operators **AND** and **OR**. The node **x** is an assignment with the expression being a **BOOLEAN** expression.

BooleanSimplification applies itself recursively to the each of the one or two new assignments that it creates. If the expression is not of one of the forms given below then **LinearizeExpression** is applied to that expression. The transformation behaves as follows:

```
t := e1 AND e2          ==>     t := e1;
                                IF t THEN t := e2 END

t := e1 OR e2           ==>     t := e1;
                                IF NOT t THEN t := e2 END

t := NOT (e1 AND e2)    ==>     t := NOT e1 OR NOT e2

t := NOT (e1 OR e2)     ==>     t := NOT e1 AND NOT e2

t := NOT NOT e          ==>     t := e
```

Here is an example demonstrating the result of applying **BooleanSimplification** to a more complex expression:

```
                                t := e1;
t := e1 AND e2 AND e3     ==>   IF t THEN t := e2 END;
                                IF t THEN t := e3 END
```

The above example shows that some redundant tests may be generated since a better result would have been the following:

```
                                t := e1;
 t := e1 AND e2 AND e3     ==>  IF t THEN
                                  t := e2;
                                  IF t THEN t := e3 END
```

```
END
```

However, a simple optimizing transformation can be used to achieve the above result and will also optimize other occurrences of such situations.

- **PROCEDURE LinearizeExpression (x : Node; s : Scope);**

  This procedure is used to reduce expressions to a simple form which is suitable for code generation. This procedure assumes that order of evaluation of operands is irrelevant and that previous simplifying transformations have been correctly applied.

LinearizeExpression is applied after the statements of the program have been simplified. The only statements which remain are: assignments, procedure-calls, single-clause if-statements, simple-loops, exit-statements, return-statements and null-statements. The restriction to just these statements is very convenient because such a restricted program has the property that when a temporary is to be generated for an expression, it can simply be placed before the enclosing statement of that expression.

When we refer to a *designator* in the following description, we will also include type-transferred designators even though they look like function calls. **LinearizeExpression** behaves according to the following description:

Constant expressions are evaluated using **ExpressConstant** and are replaced with the result.

```
x := SIZE(INTEGER) + 1     ==>     x := 5
```

If an index of an indexed-variable is not a constant or designator, or is a designator containing non-constant indices, the index is stored in a temporary and that temporary is used as an index.

```
x := a[b[i]]          ==>          t := b[i];
                                   x := a[t]
```

If an expression that is not a designator is type-transferred, it is stored in a temporary and that temporary is type-transferred. This ensures that all type-transfers can be treated as designators.

```
x := INTEGER (i + j)   ==>          t := i + j;
                                    x := INTEGER (t)
```

If a bracketted expression is found in a context where the brackets are not required, the brackets are eliminated.

```
x := (a + b) + c      ==>      x := a + b + c
```

Signed terms with plus signs are eliminated.

```
x := +a      ==>      x := a
```

If the second operand of any dyadic expression (other than **AND** or **OR** expressions) is not a designator or a constant, it is assigned to a temporary variable.

```
x := a * b + c * d      ==>   t := c * d;
                              x := a * b + t
```

Set factors which are not constants are assigned to a temporary and **SetFactorSimplification** is applied to that assignment. This may result in the construction of for-loops which are eliminated using **ForLoopToSimpleLoop**. All expressions of statements created by **SetFactorSimplification** are also linearized. See the documentation for **SetFactorSimplification** for an example of how this transformation works.

Any expression involving **AND** or **OR** is assigned to a temporary and **BooleanSimplification** is applied to that assignment. See the documentation for **BooleanSimplification** for examples of how this transformation works.

That concludes the description of how **LinearizeExpression** behaves.

- **PROCEDURE FunctionCallToProcedureCall (x : Node);**

  This procedure is used to change function procedures that return records or arrays into regular procedures. This allows the code generator to assume that a function returns only small types but still allows the user the full power of writing a function that returns a value of any type. The node **x** must be a compilation-unit.

FunctionCallToProcedureCall traverses the compilation-unit **x** and all imported definition modules. It changes appropriate function procedure declarations to procedure declarations by deleting the result type and declaring a new var-parameter of that type. It also changes appropriate function

procedure types to procedure types in a similar manner.

```
PROCEDURE foo (x,y : t) : RecordType;      ==>
PROCEDURE foo (x,y : t; VAR ReturnValue : RecordType);

t1 = PROCEDURE (t,t) : RecordType;         ==>
t1 = PROCEDURE (t,t,VAR RecordType);
```

Once the declarations have been changed as shown above, each function call of **x** is checked to see if its procedure type has been modified; if so, it is changed to a procedure call. This is done by first assigning the function call to a temporary (if it is not already directly in the context of an assignment) and then replacing the assignment with a procedure call.

```
Rec := foo(a,b)      ==>      foo(a,b,Rec)
```

Each return-statement is also checked to see if its enclosing procedure has been changed from a function procedure to a regular procedure; if so, the returned expression is assigned to the new parameter and just a return-statement remains.

```
RETURN Rec          ==>      ReturnValue := Rec;
                             RETURN
```

There are several unusual aspects about this transformation. It is the only transformation we have described which modifies existing declarations. It is also the only transformation which modifies a definition module. Normally definition modules are meant not to be modified. However, if all uses of the definition module make the same modification at compile time there will be no inconsistency. This may prove useful for other kinds of transformations which require uniform modification of definition modules (such as the applications which require the addition of monitor variables).

It may seem that since existing declarations have been modified, the symbol table needs to be regenerated. This is only partially true. If we were to apply `CorrectQ` to the procedure calls and return-statements generated by the above transformations, we would most certainly detect errors. However the code generator does not do any further checking of the program and will not detect the

inconsistency, so **Modula2ToSunAssembler** may be applied to produce correct code without regenerating the symbol table.

3.3 Transformation Reconfiguration

The metaprogrammer controls the simplifying transformations of the previous section by writing a special implementation module. This implementation module must have the name **UserTransform** and will export from its corresponding definition module a single procedure named **Simplify**. This procedure will be called by the compiler with a program module as an argument and is responsible for simplifying the module to the form required by the code generator. The object code for the compiler will be available to the metaprogrammer and he may link his compiled implementation module to the compiler's object code to create his own customized compiler. The source code for the compiler need not be accessible to the metaprogrammer.

A minimal implementation of the transformations necessary for simplifying arbitrary Modula-2 programs is given below. We will subsequently discuss its components.

```
IMPLEMENTATION MODULE UserTransform;

  PROCEDURE Simplify (x : Node);

  BEGIN
    Traverse(BlockOf(x), ScopeHandle(NameOf(x)), Stage1);
    Traverse(BlockOf(x), ScopeHandle(NameOf(x)), Stage2);
    FunctionCallToProcedureCall(x)
  END Simplify;


  PROCEDURE Stage1 (x : Node; s : Scope);

    VAR
      ListPosition   : INTEGER;
      ContainingList : Node;

  BEGIN
    IF ExpressionQ(x) THEN
      IF IndexedVarQ(x) THEN ListedIndicesToRecursiveIndices(x) END
     ELSEIF StatementQ(x) THEN
      CASE NodeType(x) OF
        WithStatement : WithStatementToStatementList(x,s) |
        ForLoop : ForLoopToSimpleLoop(x,s) |
        RepeatLoop : RepeatLoopToSimpleLoop(x) |
        WhileLoop : WhileLoopToSimpleLoop(x) |
```

```
        IfStatement : IfStatementToSimpleIfStatement(x) |
        CaseStatement :
          ContainingList := Parent(x);
          ListPosition := Position(x) - Length(ContainingList) - 1;
          CaseStatementToIfStatement(x,s);
          IfStatementToSimpleIfStatement(
            NthElement(ContainingList,ListPosition))
      END
  END Stage1;


  PROCEDURE Stage2 (x : Node; s : Scope);

    VAR
      i : INTEGER;

  BEGIN
    CASE NodeType(x) OF
      Assignment:
        LinearizeExpression(VariableOf(x),s);
        LinearizeExpression(ExpressionOf(x),s) |
      ProcedureCall:
        LinearizeExpression(ProcedureOf(x),s);
        FOR i := 1 TO Length(ArgumentsOf(x)) DO
          LinearizeExpression(NthElement(ArgumentsOf(x),i),s)
        END |
      ReturnStatement:
        IF NOT EmptyQ(ReturnValueOf(x)) THEN
          LinearizeExpression(ReturnValueOf(x),s)
        END |
      IfStatement:
        LinearizeExpression(ConditionOf(NthElement(ClausesOf(x),1)),s)
    ELSE
    END
  END Stage2;

END UserTransform.
```

For the sake of brevity, the imports of the MPS routines have not been shown in the above example. The corresponding definition module, imported by the compiler, appears as follows:

```
DEFINITION MODULE UserTransform;

  FROM m IMPORT Node;

  PROCEDURE Simplify (x : Node);

END UserTransform.
```

Simplify makes use of the procedure **Traverse** which takes a node, a scope and a procedure as arguments. The procedure argument must be a procedure which takes a node and a scope as arguments. **Traverse** is applied to a block and does a post-order traversal of all nested blocks and of all nodes nested in the bodies of those blocks. The procedure given as a parameter to **Traverse** is applied to the nested nodes in a depth-first manner. **Traverse** is a handy utility for applying transformations.

In **Simplify** we see that the program is traversed twice; many transformations are applied in a single traversal. **FunctionCallToProcedureCall** is the final transformation and completes the simplifications required by the code generator.

Most of the simplifying transformations are applied in the first traversal by **Stage1**. Earlier we mentioned that **ListedIndicesToRecursiveIndices** should be applied before any other transformation. **Stage1** ensures that it is applied first, because expressions in the body of a block are all nested in statements and thus **Traverse** will apply **Stage1** to expressions before statements. The simplification of a case-statement results in a multiple clause if-statement so after a case-statement is simplified **IfStatementToSimpleIfStatement** is applied to the resulting if-statement. This is good example of how transformations can be combined to eliminate the need for additional traversals. The remaining transformations used in **Stage1** are independent of each other and are therefore applied in the same traversal.

LinearizeExpression depends on the simplifications of **Stage1** and must therefore be applied in a separate traversal. **Stage2** is used to apply **LinearizeExpression** to the appropriate expressions. **LinearizeExpression** does its own traversal of an expression so **Stage2** applies it to outer-most expressions.

Combining all the simplifying transformations into a single procedure is a handy way of encapsulating the organization of the transformations. The **UserTransform** module is obviously very simple and is easily modified by a metaprogrammer to create a customized compiler. Subsequent examples in Chapter 4 will demonstrate the ease with which extensions may be integrated into this organization.

3.4 Generality

Certainly the idea of compiling a program based on its parse tree representation is not a new idea. In fact the DiANA intermediate representation for Ada was designed for precisely this purpose. The incremental programming environment (IPE) described in [19] also compiles programs directly from a parse tree representation, which is somewhat similar to that of MPS.

The interesting approach that this thesis proposes is the use of source-to-source transformations for simplifying the process of code generation. Source-program simplification relies on the fact that simplifications can be expressed in the source language. However, not all languages are expressive enough to allow high-level constructs to be expressed in terms of lower-level constructs. For example, Pascal does not allow for address calculations so arrays cannot be replaced with address calculations as is possible in Modula-2. If a less expressive language than Modula-2 were to be used, some language extensions might be necessary.

The issue of language extension for the purpose of code generation is a very interesting one that has not been fully explored in this thesis. For the prototype code generator described in this chapter no language extensions were required. However, it would be interesting to explore the use of register variables and how they could be employed to perform register allocation at the source level. In fact, it may well be possible to extend the source language in such a way that all the work of the code generator can be expressed at the source level with only a simple transliteration required to produce assembler code.

We have examined the basic organization of a source-to-source compiler and have demonstrated the feasibility of the organization with a working prototype. This chapter and the previous chapter have described the core facilities that are required to implement a source-to-source compiler. The next chapter will look at the flexibility and functionality that the source-to-source organization affords.

4. APPLICATIONS

The success of the source-to-source compiler organization lies not in the ability to implement a static compiler but in the ability to use and modify the compiler as a flexible tool. This chapter will examine just a few of the many applications for which the source-to-source organization is so aptly suited.

4.1 Interactive Compilation

There is a great deal of current research being conducted into integrated programming environments [8, 19, 23, 29, 39]. The research has focussed on the use of program editors that deal with programs on a syntactic (and in some cases semantic) basis. The various other tools in the environment are closely integrated with the editor through the use of a common representation of programs as data objects. The ultimate use for a source-to-source compiler is as part of such a programming environment.

A program editor could be designed to use the various transformations of the source-to-source compiler as commands. In such an environment the user would have a choice of compilation methods: he could use the system-supplied compiler which makes all required simplifications for standard programs, he could create his own personalized compiler through the modification of the routine `Simplify` discussed in 3.3, or he could apply transformations on an individual basis using commands in the program editor. Such an environment would be an editing environment as well as a program transformation environment.

The concept of interactive compilation is an interesting area for research. The source-to-source organization makes it easy to open the black box of compilation. The educational aspects of an interactive compilation system are very promising. Users could step through the process of source-to-source translation on a transformation-by-transformation basis. The relationship between the interactive commands and the MPS routines that implement them implies that users learning the commands are indirectly learning how to write a metaprogram and in particular a translator.

The ability to monitor the translation process can give users a greater understanding of how their programs are being processed. If various transformations introduce unnecessary inefficiencies into their programs they may even modify the intermediate stages of the translation by hand. For example, if certain runtime checks are inserted in places where the user is sure they are unnecessary, he can

eliminate them. Interactive optimization of programs has the promise of becoming a very important tool for software development environments.

4.2 Instrumentation

We shall now deal with an extension for inserting runtime checks of array indices. Strictly speaking, it is not really an extension at all since a Modula-2 compiler needs to have runtime checking for array indices, subrange conformity, the correct use of variant records, etc. However, in the source-to-source organization, transformations for inserting these runtime checks must be included in `Simplify` and it is therefore up to the metaprogrammer to (conscientiously!) decide which checks he wants to have inserted. The following procedure, which we will subsequently describe, can be used to insert code for checking that array indices are within the bounds of the array:

```
PROCEDURE IndexCheck (x : Node; s : Scope);

  VAR
    TheType, LowerLimit, UpperLimit : Node;

BEGIN
  TheType := GetType(ArrayOf(x), s);
  IF ParameterSectionQ(TheType) THEN
    LowerLimit := MakeInteger(MakeIntegerString('0'));
    UpperLimit := MakeFunctionCall(
                    High,
                    List1(ExpressionList, ArrayOf(x)))
   ELSE
    TheType := NthElement(IndexTypesOf(TheType),1);
    LowerLimit := Minimum(TheType,s);
    UpperLimit := Maximum(TheType,s)
   END;
  Insert(Enclosing(StatementList,x),
         Position(Enclosing(Statement,x)),
         MakeIfStatement(
           List1(
             IfClauseList,
             MakeIfClause(
               MakeAdditiveExpr(
                 MakeBracketedExpr(
                   MakeRelationalExpr(
                     NthElement(IndicesOf(x),1),
                     MakeLessOp(),
                     LowerLimit)),
                 MakeOrOp(),
                 MakeBracketedExpr(
                   MakeRelationalExpr(
```

```
                    NthElement(IndicesOf(x),1),
                    MakeGreaterOp(),
                    UpperLimit))),
                Listl(StatementList, MakeProcedureCall(Halt,NIL)))),
            NIL))
END;
```

IndexCheck assumes that **LinearizeExpression** has already been applied to the program. Thus indexed variables will have only a single index which will be a simplified designator, and a node which must be evaluated before the evaluation of a given expression may be inserted before the enclosing statement of that expression. **IndexCheck** is called with an indexed variable **x** and a scope **s** as arguments. The bounds on the index are determined first. There are two cases depending on whether the array is an open array. If the array is an open array, the lower bound is zero and the upper bound is given by applying **HIGH** to the array name. If the array is not an open array, the declaration of the array's type will have an ordinal type as the index type. The utilities **Minimum** and **Maximum** are used to return the minimum and maximum values of this index type. Once the expressions for the bounds of the index are determined, an if-statement is inserted which calls **HALT** for indices which are out of bounds. The following demonstrates the effect of applying **IndexCheck** to an open array:

```
x := a[i]     ==>     IF (i < 0) OR (i > HIGH(a)) THEN HALT() END;
                      x := a[i]
```

IndexCheck is a simple transformation for inserting runtime checks of array subscripts. It can be installed in **UserTransform** by adding a procedure **Stage3** which applies **IndexCheck** to expressions which are indexed variables. **Stage3** would be used as the third traversal of the program. Since **IndexCheck** generates an expression using **OR** it will be necessary to reapply **Stage2** after the application of **Stage3**. It would be more practical to design **IndexCheck** so that it does not use **OR**, in which case **Stage2** need not be reapplied. In the design of extensions it is important to be aware of such dependencies between transformations.

There are many applications for which automatic insertion of code is useful. The example given in this section eliminates the possibility that an index will exceed the bounds of the array. Similar routines can be designed for checking pointer references, subrange bounds and variant records. With the source-to-source organization, runtime checks are inserted directly into the source program so the metaprogrammer may easily control runtime checking.

The use of automatic code insertion for the purpose of debugging programs is an important area of research [33]. Execution analyzers, which insert code into the source program, are another relevant area of research [11, 24]. Most of such research has focussed on the theoretical aspects of program instrumentation with very little being done about the actual implementation of such tools. The source-to-source organization is well suited for implementing program instrumentation tools.

4.3 Generics

The restrictions of a language often limit the kinds of abstractions that may be expressed [26]. Modula-2 does not allow routines with a variable number of arguments so abstractions which would benefit from such routines must be expressed in a less elegant manner. With the source-to-source organization, there is the facility to allow for the use of generic capabilities when the application dictates [16].

We will examine an example making use of the ability to extend the type checking mechanism of MPS. The example concerns the use of the MPS procedures `NullList` and `List1` through `List5`, which are used to construct lists from individual elements. `NullList` is given a `NodeClass` argument and builds an empty list of that type. The remaining procedures are each given a `NodeClass` argument and the appropriate number of nodes, and build a list out of those nodes. The list building functions were designed to eliminate the use of `Append1` for building small lists. For example, the following two expression are equivalent:

```
List2(Kind,x1,x2)    <==>    Append1(Append1(NullList(Kind),x1),x2)
```

The list building operations are a notational improvement over the use of nested appends. However, what is really needed is a generic list building operation which takes a `NodeClass` argument and an arbitrary number of node arguments and combines the given nodes into a list of the specified type. The problem is of course that a procedure which takes an arbitrary number of arguments cannot be defined in Modula-2. However, the following implementation module, which we shall subsequently describe, makes it possible to extend the compiler to allow for calls to a generic list building function called `List`:

```
IMPLEMENTATION MODULE ListCall;

  VAR
    ListOccurrence, NodeOccurrence, NodeClassOccurrence,
    Append1Occurrence, NullListOccurrence : Node


  PROCEDURE IsListCall (x : Node; s : Scope) : BOOLEAN;

  BEGIN
    RETURN FunctionCall1Q(x) AND
           (DefiningOccurrence(FunctionOf(x),s) = ListOccurrence)
  END IsListCall;


  PROCEDURE ListCallCheck (x : Node; s : Scope ) : Node;

    VAR
      i : INTEGER;

  BEGIN
    IF Length(ArgumentsOf(x)) < 1 THEN
      Error(x,Str('NodeClass argument required'))
     ELSE
      IF GetType(NthElement(ArgumentsOf(x),1),s) <>
         DefiningTypeOf(Parent(NodeClassOccurrence)) THEN
        Error(NthElement(ArgumentsOf(x),1),
              Str('First argument must be of type NodeClass'))
       END;
      FOR i := 2 TO Length(ArgumentsOf(x)) DO
        IF GetType(NthElement(ArgumentsOf(x),i),s) <>
           DefiningTypeOf(Parent(NodeOccurrence)) THEN
          Error(NthElement(ArgumentsOf(x),i),
                Str('Must be of type Node'))
        END
       END
     END;
    RETURN NodeOccurrence
  END ListCallCheck;


  PROCEDURE RemoveListCall (x : Node; s : Scope);

    VAR
      i : INTEGER;
      Substitution : Node;

  BEGIN
    Substitution := MakeFunctionCall(
                      GetVisibleName(NullListOccurrence,s),
                      List1(
                        ExpressionList,
```

```
                        NthElement(ArgumentsOf(x),1)));
      FOR i := 2 TO Length(ArgumentsOf(x)) DO
        Substitution := MakeFunctionCall(
                        GetVisibleName(AppendlOccurrence,s),
                        List2(
                         ExpressionList,
                         Substitution,
                         NthElement(ArgumentsOf(x),i)))
       END;
      Replace(x,Substitution)
    END RemoveListCall;


    VAR
      s : Scope;
      GenericModule : Node;

  BEGIN
    GenericModule := ParseFile(DefinitionModule,
                               Reset(Str('Generic.def')));
    IF NOT Declare(GenericModule) THEN HALT() END;
    s := ScopeHandle(NameOf(GenericModule));
    ListOccurrence := DefiningOccurrence(MakeIdentifier(Str('List')),s);
    NodeOccurrence := DefiningOccurrence(MakeIdentifier(Str('Node')),s);
    NodeClassOccurrence := DefiningOccurrence(
                           MakeIdentifier(Str('NodeClass')),s);
    AppendlOccurrence := DefiningOccurrence(
                         MakeIdentifier(Str('Appendl')),s);
    NullListOccurrence := DefiningOccurrence(
                          MakeIdentifier(Str('NullList')),s);
    UserExpressionCheck(IsListCall,ListCallCheck);
  END ListCall.
```

For the sake of brevity the imports to `ListCall` are not shown. User programs import the generic list operation from the following definition module:

```
DEFINITION MODULE Generic;

  FROM m IMPORT Node,NodeClass,Appendl,NullList;

  PROCEDURE List (kind : NodeClass (* x1..xn : Node *)) : Node;

END Generic.
```

In the body of the module `ListCall`, the definition module `Generic` is parsed and `Declare`d to install it in the symbol table. `Generic` imports the types `Node` and `NodeClass` and the procedures `Appendl` and `NullList`. It does so because the definition of `List` depends on the two imported types and is

implemented in terms of the two imported procedures. The body of **ListCall** makes use of the fact that the procedures and types for which defining occurrences are required are visible in **Generic**. The required defining occurrences are computed and then **UserExpressionCheck** is called to install the routine for checking the correctness of calls to **List**.

The module **ListCall** declares three procedures. One for recognizing calls to **List**, one for checking the correctness of a call to **List** and one for replacing a call to **List** with an equivalent construct expressed in standard Modula-2.

The recognition routine **IsListCall** is very simple. For a node to be a call to **List** it must be a function call and the defining occurrence of the function must the declaration of **List** in the module **Generic**.

The checking routine **CheckListCall** must check that calls to **List** are correctly formed. There must be at least one argument. The first argument must be of type **NodeClass** and any remaining arguments must all be of type **Node**. **CheckListCall** calls **Error** for violations. Checking routines for expressions must return the defining occurrence of the name of the type of that expression, so **ListCallCheck** returns the defining occurrence of **Node**.

The routine **RemoveListCall** is responsible for replacing a call to **List** with standard Modula-2 constructs. There is a very simple equivalence between calls to **List** and the use of the procedures **NullList** and **Append1** which is illustrated as follows:

```
List(Kind,x1,x2)  ==>  Append1(Append1(Append1(NullList(Kind),x1),x2))
```

To include the extension in **UserTransform** the procedures **IsListCall** and **RemoveListCall** are imported from the following definition module:

```
DEFINITION MODULE ListCall;

  FROM m IMPORT Node;
  FROM Symbol IMPORT Scope;

  PROCEDURE IsListCall     (x : Node; s : Scope) : BOOLEAN;
  PROCEDURE RemoveListCall (x : Node; s : Scope);

END ListCall.
```

Importing from **ListCall** will automatically install the routine for type-checking calls to **List**. The two imported procedures can then be used in **Stage1** to eliminate calls to **List** by inserting the statement:

```
IF IsListCall(x,s) THEN RemoveListCall(x,s) END;
```

The extension in very easy to integrate with the existing structure of the compiler.

4.4 Macros

The generic list building function of the previous section is essentially a macro. As such, we may compare the ability to define macros in MPS with the macro preprocessor of the C programming language [14]. The C macro preprocessor is a text-based facility used for notational convenience and efficient inline coding of subroutines. Extensions to MPS can be used to achieve similar effects.

The most commonly cited example of a macro in C is a macro which returns the maximum of two arguments. Such a macro may be defined in terms of a conditional expression as follows:

```
#DEFINE Max(x,y)  ((x) > (y) ? (x) : (y))
```

Max is handy to use and is often more efficient than an actual procedure call. It is also more general than a procedure could be because it applies equally well to any type for which the ">" operator applies. However, there are disadvantages to C style macros which can be seen in this particular example. The arguments are not type-checked until after the macro substitution is made, leading to confusing error messages. Furthermore, arguments which are used more than once in the macro body lead to problems when there are side-effects in the evaluation of those arguments. Even if re-evaluation of the arguments does not lead to erroneous results, it is nevertheless a source inefficiency.

In MPS we can define a macro like **Max** without the disadvantages we have discussed. We may design **Max** to take an arbitrary number of arguments. This an improvement over C macros, which require a fixed number of arguments. **Max** will do appropriate type checking of the arguemnts and takes arguments of any type for which the ">" operator applies. The procedure for replacing calls to **Max** examines its arguments and generates temporaries for those arguments which require them (due to side-effects or efficiency considerations). The following is an example of how such a macro in MPS

would behave:

```
Largest[i] := Max(a[i],f(i),x)     ==>     Result := a[i];
                                           Temporary := f(i);
                                           IF Temporary > Result THEN
                                             Result := Temporary
                                           END;
                                           IF x > Result THEN
                                             Result := x
                                           END;
                                           Largest[i] := Result
```

The main advantages of the C macro preprocessor are the fact that a macro is defined as part of the program and that the definition of a macro is very terse. In MPS the macro must be defined separately and that definition must be compiled and integrated with the compiler before it can be used. However, more powerful macros can be designed in MPS than are possible using the C macro preprocessor. In MPS, macros can make use of program properties (such as type) which are not available to text based macros.

The literature contains many examples of systems that make use of macro preprocessors [13, 22]. The power and unrestricted nature of macros make them a useful tool for extending languages in a convenient way. The preprocessing capabilities of the source-to-source organization can be used to easily implement the kinds of preprocessors that are described in the literature.

4.5 Inline Assembler Code

To complete our discussions of extensions, we shall deal with an application concerning an extension to the code generator. The purpose of the extension is to make efficient use of the hardware instruction for computing a square root. The following implementation module can be used to inline-code, at the assembler level, calls to `Sqrt`:

```
IMPLEMENTATION MODULE InlineMath;

  VAR
    SqrtOccurrence : Node;


  PROCEDURE IsSqrtCall (x : Node; s : Scope) : BOOLEAN;
```

```
  BEGIN
    RETURN FunctionCallQ(x) AND
           (DefiningOccurrence(FunctionOf(x),s) = SqrtOccurrence)
  END IsSqrtCall;


  PROCEDURE InlineSqrt (x : Node; s : Scope);

    VAR
      Arg : Node;

  BEGIN
    Arg := NthElement(ArgumentsOf(x),1);
    IF ConstantQ(Arg,s) or DesignatorQ(Arg) THEN
      Assembler3(Str('fsqrt'),Op(Arg,s),Str('fp0'))
     ELSE
      Gen(Arg,s);
      Assembler3(Str('fsqrt'),Str('fp0'),Str('fp0'))
     END;
  END InlineSqrt;

  VAR
    MathLib : Node;

BEGIN
  MathLib := ParseFile(DefinitionModule,Reset(Str('MathLib.def')));
  IF NOT Declare(MathLib) THEN HALT() END;
  SqrtOccurrence := DefiningOccurrence(MakeIdentifier(Str('Sqrt')),
                                       ScopeHandle(NameOf(MathLib));
  UserGen(IsSqrtCall,InlineSqrt)
END InlineMath.
```

In the body of **InlineMath** the definition module **MathLib**, containing the defining occurrence of **Sqrt**, is parsed and **Declare**d. The defining occurrence for **Sqrt** is determined and saved. Then **UserGen** is called to install in the code generator, the extension for inline coding **Sqrt**.

The recognition routine **IsSqrtCall** tests if its node argument is a function call and if the defining occurrence of the function is the declaration of **Sqrt** in the module **MathLib**. **InlineSqrt** generates the assembler code for calling the floating point hardware instruction. If the argument to **Sqrt** is a constant or designator then **InlineSqrt** uses **Op** to get the effective address of the argument which it then uses as the first operand of the **fsqrt** assembler instruction. Otherwise **Gen** is used to evaluate the expression leaving the result in register **fp0** which is then used as the first operand to **fsqrt**. In either case the result of the operation is put in **fp0**.

The extension to the code generator can be installed as part of the compiler by simply having `UserTransform` import the module `InlineMath`. Nothing else need be done!

With the source-to-source organization, it is very easy to implement extensions involving the assembler output. A facility for extending languages to allow assembler code to be embedded in source programs is described in [31]. An interface to the machine-dependent layer is carefully considered. The source-to-source organization makes the implementation of such language extensions extremely simple. In fact, because the organization is so flexible, programmers requiring access to the machine layer can design the kind of interface that is appropriate for their particular application.

We have examined several different applications which extend the type checking facility of MPS, reconfigure the source-to-source transformations used by the compiler and extend the code generator. These extensions have proven to be both easy to express and easy to integrate into the existing structure of the compiler. The success of the source-to-source organization is clearly demonstrated by the power and flexibility that has been observed in the applications described in this chapter.

5. CONCLUSION

The implementation of a source-to-source compiler is partitioned among three components: the metaprogramming system, the source-to-source transformations and the code generator. Much of MPS is generated from the GRAMPS grammar but the context-sensitive manipulations are highly language-specific and are implemented by hand. The source-to-source transformations and the code generator are both implemented in terms of the abstraction supplied by MPS. The division of effort that went into the implementation of each of the three components is roughly 70% for the context-sensitive manipulations, 20% for the code generator and 10% for the source-to-source transformations.

By far the most effort in the implementation of a source-to-source compiler goes into the design and implementation of the context-sensitive manipulations. This is definitely worthwhile because the metaprogramming system is an extremely general purpose tool that can be used in countless other applications. Thus the cost of implementing MPS is amortized by its general usefulness and the ease with which applications, such as a compiler, can be implemented.

The implementation of the required source-to-source transformations is almost a trivial matter with all the tools provided by MPS. The code generator takes a fair bit more effort to implement than the source-to-source transformations. However, there was often the choice of simplifying a construct at the source-level or designing the code generator to handle it directly. Many such choices were made in favour of a more general code generator, leading to a larger percentage of implementation effort going into the code generator.

The source-to-source organization opens up new areas of research. Compilation algorithms need not be concerned with the textual order of a program since nodes may be traversed in any order. This allows for the design of languages with less restriction on forward reference. Translation algorithms may be simplified by being able to process program components in any order. This capability is useful for Modula-2; a module's exports are most conveniently processed after the declarations of the module but textually appear before.

An interesting area of research is the design of a language which is specifically tailored for source-to-source compilation. Such a language could be designed to allow for the efficient implementation of an incremental symbol table rather than requiring a precalculated symbol table like the one used for Modula-2. Facilities to allow for flexible language extension could be built right into

the definition of the language. Consider the power of having a *metaprocedure* which, instead of specifying the code for carrying out a given operation, specifies how the operation is to be inline-coded. Metaprocedures would perform their own type checking and would therefore be flexible enough to specify any abstraction. In fact, the routine for inline coding **Max** described in 4.4 is a very simple example of a metaprocedure.

The nature of the division between source-to-source simplification and code generation is another area in need of further research. It would be very interesting to investigate the possibility of using language extensions to allow a model of the underlying machine architecture to be visible at the source level. In Modula-2 such extensions could take the form of a special module containing variables corresponding to registers and procedures corresponding to hardware instructions. It may well be possible to express all the work of the code generator in the source program so that only a simple transliteration is needed to convert the program to assembly language.

The source-to-source organization focusses more on the flexibility of the compiler than on its efficiency. The traditional concern for efficiency has lead to organizations consisting of interdependent components that are not generally useful. Hence, much of the effort that goes into the implementation of a conventional compiler cannot be applied to solving other program manipulation problems. The source-to-source organization consists of a toolkit of components. Not only can the compiler be easily modified but the components of the toolkit are useful for other language-based tools such as program editors, debuggers and interpreters. In fact, the organization is such that we have a compiler and a powerful program manipulation system.

Using the source language as the intermediate language is conducive to achieving a thorough understanding of that language. Both users and metaprogrammers may benefit from this. The properties of source language programs are carefully explored and considered in the design of the metaprogramming system. Problem areas in the language are therefore resolved and documented along with the metaprogramming system and the tools of the programming environment. This results in a cleaner environment for program development.

The emphasis on the use of the source language has an educational benefit. Users can learn more about the source language by monitoring the kinds of changes that are made to their program as it is prepared for final translation to assembly language. Seeing the power that is available for writing automated program manipulations should serve as a source of motivation for learning to use MPS. An

understanding of the metaprogramming system for a language leads to a deep insight into the properties of that language.

The flexibility of the source-to-source compiler lies in the ability to modify the translation process. This ability is achieved through modification of type-checking in MPS, reconfiguration of the transformations for simplifying programs in preparation for code generation, and extension to the code generator for special-purpose constructs. The combination of these three facilities provides a degree of flexibility that is not possible with traditional organizations.

Chapter 4 has demonstrated the effectiveness of the facilities for modifying the translation process. We have seen applications for: inserting code for runtime checks, extending Modula-2 to accept a generic function, and extending the code generator to generate efficient code for a numeric operation. These examples show the ease with which extensions can be designed and integrated into the existing structure of the compiler.

When dealing with extensions to a language, the question of portability naturally arises. A program making use of extensions can be difficult to port to a system which doesn't provide the same extensions. However, we have seen that extensions in MPS can act as macros which replace instances of the extension with standard constructs. A program using such macros can be transformed to a program which does not use macros so the program is still easily portable.

The ability to extend the compiler can also be used to eliminate a portability problem. If a program from another system is to be ported, the source-to-source compiler could be modified to accept certain non-standard constructs allowing the program to be compiled in its original form. Therefore, while the ability to extend the compiler can lead to portability problems, this is not necessarily the case and extensions can even be used to solve portability problems.

The source-to-source organization helps in achieving a reliable compiler. With so much work done at the source level the actions of the translator are easily monitored to reveal bugs that might otherwise be very obscure. The tools of MPS can even be applied to ensure that transformations produce context-sensitively correct results. The conceptual simplification alone makes the compiler more reliable simply because it is easier to understand.

The current research into integrated programming environments is also relevant to the source-to-source organization. The various tools of an integrated programming environment must share a common view

of programs as data objects. MPS can be used to achieve such a common view by supplying the abstraction that is used to implement all tools in the environment. A source-to-source compiler, implemented in terms of the abstraction supplied by MPS, is therefore well suited for use in an integrated programming environment. There is even the potential having interactive compilation in such an environment.

The source-to-source organization results in an extremely flexible compiler which may be easily modified to suit the requirements of special applications. Increased reliability is also achieved as a direct result of the conceptual simplification of the translation process. The toolkit configuration of the compiler's components gives rise to a functionality that extends beyond the domain of compilers into the domain of general program transformation.

APPENDIX A - GRAMPS MODULA2 GRAMMAR

1 Program Modules and Declarations

<CompilationUnit> ::= <DefinitionModule> | <ProgramModule>

<DefinitionModule> ::= "DEFINITION" "MODULE" <Name:Identifier> (* *) ";"
[<Imports:ImportList>] [<Exports:ExportDecl>] <Declarations:ExternalDeclList> "END" "."

<ExternalDeclList> ::= <ExternalDecl> {<ExternalDecl>}

<ExternalDecl> ::= <ConstSection> | <TypeSection> | <VarSection> | <ProcedureHeader>

<ProgramModule> ::= [<ImplementationIndication:ImplementationIndication>] "MODULE"
<Name:Identifier> ["[" <Priority:Expression> "]"] (* *) ";" [<Imports:ImportList>]
<Block:Block> "END" "."

<ModuleDecl> ::= "MODULE" <Name:Identifier> ["[" <Priority:Expression> "]"] (* *) ";"
[<Imports:ImportList>] [<Exports:ExportDecl>] <Block:Block> "END" ";"

<ImportList> ::= <ImportSection> {<ImportSection>}

<ImportSection> ::= ["FROM" <ExportingModule:Identifier>] "IMPORT"
<ImportedNames:IdentifierList> (* *) ";"

<IdentifierList> ::= <Identifier> {"," <Identifier>}

<ExportDecl> ::= "EXPORT" [<QualifiedIndication:QualifiedIndication>]
<ExportedNames:IdentifierList> (* *) ";"

<Block> ::= [<Declarations:DeclList>] ["BEGIN" <Body:StatementList>]

<DeclList> ::= <DeclSection> {<DeclSection>}

<DeclSection> ::= <ConstSection> | <TypeSection> | <VarSection> | <ProcedureDecl> |
<ModuleDecl>

<ConstSection> ::= "CONST" <ConstDefs:ConstDefList>

<ConstDefList> ::= <ConstDef> {<ConstDef>}

<ConstDef> ::= <ConstName:Identifier> "=" <Value:Expression> (* *) ";"

<TypeSection> ::= "TYPE" <TypeDefs:TypeDefList>

<TypeDefList> ::= <TypeDef> {<TypeDef>}

<TypeDef> ::= <TypeName:Identifier> ["=" <DefiningType:Type>] (* *) ";"

<VarSection> ::= "VAR" <VarDecls:VarDeclList>

<VarDeclList> ::= <VarDecl> {<VarDecl>}

<VarDecl> ::= <VarNames:IdentifierList> ":" <DeclaredType:Type> (* *) ";"

<ProcedureDecl> ::= "PROCEDURE" <Name:Identifier> <Profile:ProcedureProfile> (* *) ";"
<Block:Block> "END" ";"

<ProcedureHeader> ::= "PROCEDURE" <Name:Identifier> <Profile:ProcedureProfile> (* *) ";"

<ProcedureProfile> ::= "(" [<Parameters:ParameterList>] ")" [":" <ResultType:Name>]

<ParameterList> ::= <ParameterSection> {";" <ParameterSection>}

<ParameterSection> ::= [<VarIndication:VarIndication>] <Parameters:IdentifierList> ":"
[<ArrayIndication:ArrayIndication>] <ParameterType:Name>

<ArrayIndication> ::= "ARRAY" "OF"

<StatementOrDeclSection> ::= <Statement> | <DeclSection>

2 Statements

<StatementList> ::= <Statement> {";" <Statement>}

<Statement> ::= <Assignment> | <ProcedureCall> | <IfStatement> | <CaseStatement> |
<WhileLoop> | <RepeatLoop> | <ForLoop> | <SimpleLoop> | <ExitStatement> |
<ReturnStatement> | <WithStatement> | <NullStatement>

<Assignment> ::= <Variable:Designator> ":=" <Expression:Expression> (* *)

<ProcedureCall> ::= <Procedure:Designator> "(" [<Arguments:ExpressionList>] ")" (* *)

<IfStatement> ::= "IF" <Clauses:IfClauseList> ["ELSE" <DefaultStatements:StatementList>] "END"

<IfClauseList> ::= <IfClause> {"ELSIF" <IfClause>}

<IfClause> ::= <Condition:Expression> "THEN" (* *) <Body:StatementList>

<CaseStatement> ::= "CASE" <Selector:Expression> "OF" <Clauses:CaseClauseList>
["ELSE" <DefaultStatements:StatementList>] "END"

<CaseClauseList> ::= <CaseClause> {"|" <CaseClause>}

<CaseClause> ::= <CaseLabels:SetElementList> ":" (* *) <Body:StatementList>

<WhileLoop> ::= "WHILE" <Condition:Expression> "DO" (* *) <Body:StatementList> "END"

<RepeatLoop> ::= "REPEAT" (* *) <Body:StatementList> "UNTIL" <Condition:Expression>

<ForLoop> ::= "FOR" <LoopVar:Identifier> ":=" <InitialValue:Expression> "TO"
<FinalValue:Expression> ["BY" <Increment:Expression>] "DO" (* *) <Body:StatementList>
"END"

<SimpleLoop> ::= "LOOP" (* *) <Body:StatementList> "END"

<ExitStatement> ::= "EXIT" (* *)

<ReturnStatement> ::= "RETURN" [<ReturnValue:Expression>] (* *)

<WithStatement> ::= "WITH" <Designator:Designator> "DO" (* *) <Body:StatementList> "END"

<NullStatement> ::=

3 Types

<Type> ::= <SimpleType> | <PointerType> | <RecordType> | <SetType> | <ArrayType> |
<ProcedureType>

<SimpleType> ::= <EnumeratedType> | <RangeType> | <Name>

<EnumeratedType> ::= "(" <EnumerationConstants:IdentifierList> ")"

<RangeType> ::= [<BaseType:Name>] "[" <LowerBound:Expression> ".."
<UpperBound:Expression> "]"

<PointerType> ::= "POINTER" "TO" <BaseType:Type>

<SetType> ::= "SET" "OF" <BaseType:SimpleType>

<ArrayType> ::= "ARRAY" <IndexTypes:IndexTypeList> "OF" <BaseType:Type>

<IndexTypeList> ::= <SimpleType> {"," <SimpleType>}

<RecordType> ::= "RECORD" <RecordSections:RecordSectionList> "END"

<RecordSectionList> ::= <RecordSection> {";" <RecordSection>}

<RecordSection> ::= <FixedSection> | <VariantSection>

<FixedSection> ::= <FieldNames:IdentifierList> ":" <Type:Type>

<VariantSection> ::= "CASE" [<SelectionField:Identifier>] ":" <SelectionType:Name> "OF"
<Variants:VariantList> ["ELSE" <DefaultFields:RecordSectionList>] "END"

<VariantList> ::= <Variant> {"|" <Variant>}

<Variant> ::= <CaseLabels:SetElementList> ":" [<VariantFields:RecordSectionList>]

<ProcedureType> ::= "PROCEDURE" "(" [<ParameterTypes:ParameterTypeList>] ")"
[":" <ResultType:Name>]

<ParameterTypeList> ::= <ParameterType> {"," <ParameterType>}

<ParameterType> ::= [<VarIndication:VarIndication>] [<ArrayIndication:ArrayIndication>]
<BaseType:Name>

4 Expressions

<ExpressionList> ::= <Expression> {"," <Expression>}

<Expression> ::= <SimpleExpr> | <Relation>

<SimpleExpr> ::= <Term> | <SignedTerm> | <AdditiveExpr>

<Term> ::= <Factor> | <MultiplyingExpr>

<Factor> ::= <Designator> | <String> | <Number> | <FunctionCall> | <BracketedExpr> |
<NotExpr> | <SetFactor>

<Designator> ::= <Name> | <IndexedVar> | <ComponentVar> | <ReferenceVar>

<IndexedVar> ::= <Array:Designator> "[" <Indices:ExpressionList> "]"

<ComponentVar> ::= <Record:Designator> "." <Field:Identifier>

<ReferenceVar> ::= <Pointer:Designator> " ^ "

<FunctionCall> ::= <Function:Designator> "(" [<Arguments:ExpressionList>] ")"

<BracketedExpr> ::= "(" (* *) <Expression:Expression> ")"

<NotExpr> ::= "NOT" <Operand:Factor>

<SetFactor> ::= [<SetType:Name>] "{" [<Elements:SetElementList>] "}"

<SetElementList> ::= <SetElement> {"," <SetElement>}

<SetElement> ::= <Expression> | <RangeElement>

<RangeElement> ::= <LowerLimit:Expression> ".." <UpperLimit:Expression>

<MultiplyingExpr> ::= <Operand1:Term> <Operator:MultiplyingOp> <Operand2:Factor>

<AdditiveExpr> ::= <Operand1:SimpleExpr> <Operator:AddingOp> <Operand2:Term>

<Relation> ::= <Operand1:SimpleExpr> <Operator:RelationalOp> <Operand2:SimpleExpr>

<SignedTerm> ::= <Sign:Sign> <Operand:Term>

<MultiplyingOp> ::= <TimesOp> | <DivideOp> | <DivOp> | <ModOp> | <AndOp>

<AddingOp> ::= <PlusOp> | <MinusOp> | <OrOp>

<RelationalOp> ::= <EqualOp> | <LessOp> | <LessOrEqualOp> | <GreaterOrEqualOp> |
<GreaterOp> | <NotEqualOp> | <InOp>

<Sign> ::= <PlusSign> | <MinusSign>

<Number> ::= <Integer> | <RealNumber>

<Name> ::= <Identifier> | <QualIdent>

<QualIdent> ::= <Module:Identifier> "@" <LocalName:Name>

5 Lexical Rules

<ImplementationIndication> ::= "IMPLEMENTATION"

<QualifiedIndication> ::= "QUALIFIED"

<VarIndication> ::= "VAR"

<Identifier> ::= <letter> {<letter> | <digit>}

<Integer> ::= <digit> {<digit>}

<RealNumber> ::= <integer> ["." <integer>] ["E" [<sign>] <integer>]

<String> ::= "'"{<character>}"'" | '"'{<character>}'"'

<TimesOp> ::= "*"

<DivideOp> ::= "/"

<DivOp> ::= "DIV"

<ModOp> ::= "MOD"

<AndOp> ::= "AND"

<PlusOp> ::= "+"

<MinusOp> ::= "-"

<OrOp> ::= "OR"

<EqualOp> ::= "="

<LessOp> ::= "<"

<LessOrEqualOp> ::= "<="

<GreaterOrEqualOp> ::= ">="

<GreaterOp> ::= ">"

<NotEqualOp> ::= "<>"

<InOp> ::= "IN"

<PlusSign> ::= "+"

<MinusSign> ::= "-"

APPENDIX B - SYMBOL TABLE ROUTINES

1 Introduction

This appendix describes the symbol table routines for dealing with context-sensitive properties of Modula-2. A rudimentary knowledge of Modula-2 and Multi-MPS is assumed [4, 5, 32]. The complete definition module is given in section 3.

2 Scopes, Denotations, Declarations and Correctness

2.1 Name Layers

Every name occurring in a correct Modula-2 program has an associated meaning. The meaning associated with the use of a name depends on context since this determines name visibility. It is essential that we have a good understanding of exactly how the meaning of a name is determined from its context so that we may manipulate the program correctly.

We will follow an approach for describing the scope rules of Modula-2 that is closely tied to the GRAMPS grammar. We notice that the visibility of names changes at the following critical points in the grammar:

1. A definition module, program module or local module brings into visibility the names of its imports and declarations, obstructing the visibility of all names outside the module.
2. A procedure brings into visibility the names of its declarations, obstructing the visibility of only those names that are redefined.
3. A with-statement brings into visibility the field names associated with its designator, obstructing the visibility of only those names that are duplicates of the field names.
4. A component variable brings into visibility the field names associated with its record, obstructing the visibility of all other names.

Let us define a *name layer* to be a mapping of names to meanings and say that every `DefinitionModule`, `ModuleDecl`, `ProgramModule`, `ProcedureDecl`, `WithStatement` and `ComponentVar` has an associated name layer. There is also a global name layer, associated with the universe enclosing all compilation units, which contains the names of definition modules and of the

standard identifiers. Let us not worry for the moment about how a name comes to appear in a given name layer and concentrate instead on how the name layers can be used to determine the meaning of a name.

To determine the meaning of a name we must search through some name layers until we find a mapping defined on the name. The algorithm **ScopeHandle** is given an arbitrary node **x** and computes its *scope*, the list of name layers used to determine the meaning of names at the node **x**. Assume that we have an opaque Modula type **Scope**, a function procedure **AppendLayer** which returns the result of appending a name layer to a **Scope** and a function procedure **NameLayer** which returns the name layer associated with its node argument.

```
PROCEDURE ScopeHandle ( x : Node ) : Scope;

BEGIN
  CASE NodeType(Parent(x)) OF
    Empty:
      RETURN GlobalNameLayer
    |
    DefinitionModule, ProgramModule, ModuleDecl:
      RETURN NameLayer(Parent(x))
    |
    ProcedureDecl:
      RETURN AppendLayer(NameLayer(Parent(x)),
                         ScopeHandle(Parent(x)))
    |
    WithStatement:
      IF x = BodyOf(Parent(x)) THEN
        RETURN AppendLayer(NameLayer(Parent(x)),
                           ScopeHandle(Parent(x)))
       ELSE
        RETURN ScopeHandle(Parent(x))
       END
    |
    ComponentVar:
      IF x = FieldOf(Parent(x)) THEN
        RETURN AppendLayer(NameLayer(Parent(x),
                           ScopeHandle(Parent(x)))
       ELSE
        RETURN ScopeHandle(Parent(x))
       END
    |
    ImportSection:
      IF x = ExportingModuleOf(Parent(x)) THEN
        RETURN ScopeHandle(Enclosing(ModuleDecl,x))
       ELSE
        RETURN ScopeHandle(Parent(x))
       END
```

```
    ELSE
      ScopeHandle(Parent(x))
    END
  END ScopeHandle;
```

In the above algorithm it is interesting to note that the exporting-module of an import-section begins its scope outside the enclosing module. This is so because the specified exporting module is not itself imported and its name is therefore not visible in the module containing the import section.

The list of name layers returned by **ScopeHandle** can have one of three forms:

1. If **x** is the exporting module of an import section or has an empty parent then a list containing just the global name layer is returned.
2. If **x** is a field then a list containing a single component variable name layer is returned.
3. Otherwise, a list containing zero or more with-statement name layers followed by zero or more procedure name layers followed by the name layer of a module is returned.

2.2 Denotations

The meaning of a name is referred to as its *denotation* and this denotation is determined by the *defining occurrence* of the name. The defining occurrence of a name is the identifier node at which the name is conceptually associated with its various properties. We can now more concretely define a name layer as a mapping of names to defining occurrences since this uniquely determines meaning. We will characterize all the possible contexts in which defining occurrences appear, the denotations implied by those contexts, and the name layer(s) in which the names appear:

1. **DefinitionModuleQ(Parent(x)) AND (x = NameOf(Parent(x)))**
   If **x** satisfies the above condition then **x** is the name of a definition module and is said to denote a definition module. The name of a definition module appears in the global name layer.
2. **ProgramModule(Parent(x)) AND (x = NameOf(Parent(x)))**
   If **x** satisfies the above condition then **x** is the name of a program module and is said to denote a program module. If the condition **NOT EmptyQ(ImplementationIndicationOf(Parent(x)))** is true then we may say that **x** denotes an implementation module. Interestingly enough, the name of a program module does

not appear in any name layer.

3. `ModuleDeclQ(Parent(x)) AND (x = NameOf(Parent(x)))`

   If **x** satisfies the above condition then **x** is the name of a module declaration and is said to denote a local module. The name appears in the name layer of the nearest enclosing procedure declaration, module declaration or program module.

4. `ProcedureDeclQ(Parent(x)) AND (x = NameOf(Parent(x)))`

   If **x** satisfies the above condition then **x** is the name of a procedure declaration and is said to denote a procedure. The name appears in both the name layer of the `Parent(x)` as well as the name layer of the nearest enclosing procedure declaration, module declaration or program module.

5. `ProcedureHeaderQ(Parent(x)) AND (x = NameOf(Parent(x)))`

   If **x** satisfies the above condition then **x** is the name of a procedure header and is said to denote a procedure. The name appears in the name layer of the enclosing definition module.

6. `ConstDefQ(Parent(x)) AND (x = ConstNameOf(Parent(x)))`

   If **x** satisfies the above condition then **x** is name of a constant definition and is said to denote a constant. The name appears in the name layer of the nearest enclosing definition module, module declaration, procedure declaration or program module.

7. `TypeDefQ(Parent(x)) AND (x = TypeNameOf(Parent(x)))`

   If **x** satisfies the above condition then **x** is the name of a type definition and is said to denote a type. The name appears in the name layer of the nearest enclosing definition module, module declaration, procedure declaration or program module.

8. `EnumeratedTypeQ(Parent(Parent(x)))`

   If **x** satisfies the above condition then **x** is the name of an enumeration constant of an enumerated type and is said to denote an enumerated value. The name appears in the name layer of the nearest enclosing definition module, module declaration, procedure declaration or program module.

9. `FixedSectionQ(Parent(Parent(x)) AND`
   `(Parent(x) = FieldNamesOf(Parent(Parent(x)))`

   If **x** satisfies the above condition then **x** is the name of a field in the field name list of a fixed section of a record type and is said to denote a field. The name appears in the name layer of each `WithStatement` having a designator possessing the type

Enclosing(RecordType,x) and each **ComponentVar** possessing the type **Enclosing(RecordType,x)**.

10. **VariantSectionQ(Parent(x)) AND**
(x = SelectionFieldOf(Parent(x)))

If **x** satisfies the above condition then **x** is the name of the selection field of a variant section of a record type and is said to denote a tag field. We may also state that **x** simply denotes a field. The name appears in the name layer of a **WithStatement** having a designator with **Enclosing(RecordType,x)** as its type and a **ComponentVar** having a record with **Enclosing(RecordType,x)** as its type.

11. **ParameterSectionQ(Parent(Parent(x))) AND**
((Parent(x) = ParametersOf(Parent(Parent(x)))

If **x** satisfies the above condition then **x** is the name of a parameter in a parameter section and is said to denote a parameter. If **x** has an enclosing procedure declaration then the name appears in the name layer of **Enclosing(ProcedureDecl,x)**. Notice that the names of parameters of procedure headers do not appear in any name layer.

12. **VarDeclQ(Parent(Parent(x))) AND**
(Parent(x) = VarNamesOf(Parent(Parent(x)))

If **x** satisfies the above condition then **x** is the name of a variable in a variable declaration and is said to denote a variable. The name appears in the name layer of the nearest enclosing definition module, module declaration, procedure declaration or program module.

2.3 Imports and Exports

Imports and exports provide controlled duplication of names between name layers. Name duplication can be explicitly controlled with export declarations and import sections. There are also several forms of implicit importation and exportation to be discussed subsequently.

Export declarations may appear only in local modules. The GRAMPS grammar allows export declarations in definition modules but they are totally ignored in accordance with Wirth's 3rd edition "Report on Modula-2" [32]. If an identifier **x** is exported qualified then the name, qualified with the name of the exporting module, will appear in the name layer of the module's nearest enclosing procedure declaration, module declaration or program module. If an identifer **x** is exported unqualified then **x** itself will appear in the name layer of the module's nearest enclosing procedure declaration, module declaration or program module.

Import sections may appear in any kind of module. If the exporting module is specified then the name layer of that module is used as a source of names and the module is treated as if its exports are unqualified. If no exporting module is specified then the name layer(s) of the node enclosing the importing module are used as a source of names and the names are all treated as if they are exported unqualified. The effect of importing a name from some source to some destination is exactly the same as if that source exported the name to the destination, so the rules for exporting can be applied to describe the effect of importing.

The standard identifiers are "pervasive", that is, they are automatically imported into all modules and may not be explicitly imported or exported. All names declared in a definition module are implicitly exported qualified. An implementation module's name layer automatically contains all the names appearing in the name layer of its associated definition module. However, the names of the opaque types and procedure headers must map to defining occurrences appearing in the implementation module since these names must be redefined.

Imports to a module may not be subsequently exported. The names of individual enumerated values may not be imported/exported since they are automatically imported/exported when the name of their type is imported/exported. If the name of a module is imported/exported the names exported by that module are also imported/exported. The exportation of module names can lead to multiple levels of qualification as shown in the following example:

```
MODULE Program;

  MODULE m1;
    EXPORT QUALIFIED m2;

    MODULE m2;
      EXPORT QUALIFIED x;

      VAR x : INTEGER;
    END m2;
  END m1;

  MODULE m3;
    FROM m1 IMPORT m2;

    MODULE m4;
      FROM m2 IMPORT x;

    BEGIN
      x := 0    (* notice how the qualification is undone *)
```

```
    END m4;

  BEGIN
    m2@x := 1  (* notice how importing m2 also imports m2@x *)
  END m3;

BEGIN
  m1@m2@x := 2 (* notice the multiple levels of qualification *)
END Program.
```

2.4 Standard Identifiers

The standard identifiers are not declared anywhere so there is no actual defining occurrence. However, the predefined names can be thought of as corresponding to the following declarations:

```
TYPE
  BITSET   =  BITSET;
  BOOLEAN  =  (FALSE,TRUE);
  CARDINAL =  CARDINAL;
  CHAR     =  CHAR;
  INTEGER  =  INTEGER;
  LONGINT  =  LONGINT;
  LONGREAL =  LONGREAL;
  PROC     =  PROCEDURE ();
  REAL     =  REAL;

PROCEDURE ABS   (x : ARRAY OF WORD) : WORD;
PROCEDURE CAP   (x : CHAR) : CHAR;
PROCEDURE CHR   (x : CARDINAL) : CHAR;
PROCEDURE DEC   (VAR x : ARRAY OF WORD; y : CARDINAL);
PROCEDURE EXCL  (VAR x : ARRAY OF WORD; y : WORD);
PROCEDURE FLOAT (x : CARDINAL) : REAL;
PROCEDURE HALT  ();
PROCEDURE HIGH  (VAR x : ARRAY OF WORD) : CARDINAL;
PROCEDURE INC   (VAR x : ARRAY OF WORD; y : CARDINAL);
PROCEDURE INCL  (VAR x : ARRAY OF WORD; y : WORD);
PROCEDURE MAX   (x : WORD);
PROCEDURE MIN   (x : WORD);
PROCEDURE ODD   (x : WORD) : BOOLEAN;
PROCEDURE ORD   (x : WORD) : CARDINAL;
PROCEDURE SIZE  (x : WORD) : CARDINAL;
PROCEDURE TRUNC (x : REAL) : CARDINAL;
PROCEDURE VAL   (x,y : WORD) : WORD;
```

It is convenient to use the above declarations to represent the defining occurrences of the standard identifiers. Standard identifiers thus have a defining occurrence with enough context to reflect their denotation. Notice however, that many of the above declarations are not valid declarations nor do all the declarations of the standard procedures correctly specify the parameter types.

The standard module SYSTEM is represented as follows:

```
DEFINITION MODULE SYSTEM;

  TYPE
    ADDRESS = ADDRESS;
    WORD    = POINTER TO ADDRESS;

  PROCEDURE ADR (VAR x : ARRAY OF WORD) : ADDRESS;

END SYSTEM.
```

3 Basic Symbol Table Routines

The following is the complete definition module for the symbol table routines:

```
DEFINITION MODULE Symbol;

 FROM m IMPORT Node;
 FROM String IMPORT StringType;

 TYPE
   Scope;
   Recognize   = PROCEDURE (Node, Scope) : BOOLEAN;
   TypeChecker = PROCEDURE (Node, Scope) : Node;
   Checker     = PROCEDURE (Node, Scope);

 PROCEDURE ClearSymbolTable ();
 PROCEDURE Declare (x : Node) : BOOLEAN;
 PROCEDURE ScopeHandle (x : Node) : Scope;
 PROCEDURE DefiningOccurrence (x : Node; s : Scope) : Node;
 PROCEDURE CorrectQ (x : Node; s : Scope) : BOOLEAN;
 PROCEDURE ConstantQ (x : Node; s : Scope) : BOOLEAN;
 PROCEDURE ExpressConstant (x : Node; s : Scope) : Node;
 PROCEDURE GetType (x : Node; s : Scope) : Node;
 PROCEDURE GetTypeName (x : Node; s : Scope) : Node;
 PROCEDURE QualifyingDesignator (x : Node; s : Scope) : Node;
 PROCEDURE GenerateName (x : StringType) : Node;
 PROCEDURE Rename (x,y : Node);
```

```
PROCEDURE GetVisibleName (x : Node; s : Scope) : Node;
PROCEDURE ActivationRecordSize (x : Node; s : Scope) : INTEGER;
PROCEDURE Offset (x : Node; s : Scope) : INTEGER;
PROCEDURE UserExpressionCheck (p1 : Recognize; p2 : TypeChecker);
PROCEDURE UserStatementCheck (p1 : Recognize; p2 : Checker);
PROCEDURE Error (Location : Node; Comment : StringType);

VAR
  BitSet, Boolean, Cardinal, Char, Int, LongInt, LongReal, Proc,
  Real, Abs, Cap, Chr, Dec, Excl, Float, Halt, High, Inc, Incl,
  Max, Min, Odd, Ord, Size, Trunc, Val, System, SystemAddress,
  SystemWord, SystemAdr : Node;

END Symbol.
```

Node is imported from the module **m** which contains the grammar-based MPS routines. The variables in the **Symbol** definition module contain the defining occurrences of the standard identifiers as described in the previous section. Notice that the defining occurrence for **INTEGER** is given by **Int**. This is done to avoid conflict with the **NodeClass** identifier which is named **Integer**.

If we wish to use the concepts described in the previous sections to assist in program transformation, the program must be preprocessed to compute the contents of all name layers. The following routines give the metaprogrammer this ability.

- **PROCEDURE ClearSymbolTable();**

  This procedure initializes the data structures used by the symbol table. It is implicitly called when the module **Symbol** is imported and may be called again to restore the symbol table to the startup state.

- **PROCEDURE Declare (x : Node) : BOOLEAN;**

  This procedure is used for building the symbol table. The node type of **x** must be one of **DefinitionModule**, **ProgramModule**, **ModuleDecl**, **ProcedureDecl**, **ConstDef**, **TypeDef**, **ParameterSection** or **VarDecl** and **x** must either be a **CompilationUnit** or have an enclosing **CompilationUnit**. If any errors are encountered, they are marked with comments and **FALSE** is returned.

Using **Declare** on nodes other than compilation units allows for the addition of new declarations. It is essential to be aware that the declaration which is added is processed as if it were the last declaration to appear in its enclosing declaration list. If the metaprogrammer actually places it elsewhere in the declaration list, it is his responsibility to ensure that no names are used before they are defined. For example, in the following module, adding the declaration **x : T1** using **Declare** will not result in

Declare returning **FALSE** even though redeclaring the entire module would result in an error being detected:

```
                                          MODULE m;
                                            VAR
MODULE m;                                     x : T1
  TYPE                    ==>
    T1 = INTEGER;                           TYPE
END m.                                       T1 = INTEGER;
                                          END m.
```

Incremental modification of definition modules is possible but it should be done very cautiously since definition modules are meant to be quite static. For example, if a variable is added to a definition module, then the corresponding implementation module must be recompiled and the metaprogrammer may not even have access to this implementation module. However, if all uses of the definition module have the same modification then there will be no inconsistency. See the documentation for the **FunctionCallToProcedureCall** transformation for an example of a valid incremental modification of a definition module.

Declare processes only declarations and does not check the context-sensitive correctness of statements appearing in the nested blocks. This is in keeping with the conceptual division of a program into a declaration part and a statement part, as well as a reflection of the fact that the correctness of declarations does not depend on the correctness of the statements. The procedure **CorrectQ** can be used to check the context sensitive-correctness of statements and expressions.

A declaration which produces a name conflict will be ignored in whole or in part. Notice that this may result in large portions of the program being ignored since the declaration causing the conflict may be a procedure or module and thus all its internals will be ignored. As much of a declaration is processed as possible in order to detect more than a single error at a time.

For the time being, the symbol table routines assume that any definition modules which are mentioned in the import-list of a compilation unit will appear in the current UNIX directory under the name of the definition module with a '.def' suffix. Eventually some kind of database containing the mapping of definition modules to file names may be used for representing the global name layer.

In order to reduce the complexity caused by duplicate names, the following restriction is placed on the redefinition of names. If a name is visible in the block of a declaration then that declaration may redefine the name only if the name denotes a variable or parameter and is redefined to denote another variable or parameter. Thus the following segment of code is incorrect:

```
MODULE m;

  PROCEDURE p();
    TYPE
      t = SET OF BOOLEAN;
  END p;

TYPE
  t = INTEGER
  (* This declaration is visible in a block containing another
     declaration of "t" so one of the declarations must
     be changed. *);
END m.
```

The following segment of code correctly redefines the variable i but demonstrates another kind of error:

```
MODULE m;
  VAR
    i : BOOLEAN;

  PROCEDURE p();
    CONST
      c = SIZE(i)
      (* This use of "i" is not permitted because "i" is redefined
         later in this block. *);

    VAR
      i : INTEGER
      (* Another declaration of "i" is visible here but it also
         defines "i" to be a variable so this is allowed. *);
  END p;
END m.
```

The above program fragment demonstrates a case where the particular i being referred to in c = SIZE(i) is ambiguous. It is necessary for a name to have the same denotation throughout a given declaration list so the i in c = SIZE(i) should refer to the declaration i : INTEGER. However, if i

does refer to `i : INTEGER` then `i` is used before it is defined and so this is an error. Notice that the restriction concerning the redefinition of names implies that the only way this kind of error can occur is by the use of the standard procedure **SIZE** on a variable or parameter which is redefined later in the block.

One last point to mention about **Declare** is that the short-hand notation for multi-dimensional arrays `ARRAY T1,T2 OF T3` is not allowed, so the notation `ARRAY T1 OF ARRAY T2 OF T3` should be used. See section 4.2 for a discussion of this matter.

The procedure for computing a scope handle is provided for the metaprogrammer. It is very useful to precalculate a scope handle because it is often constant over a large section of program and is used repeatedly.

- `PROCEDURE ScopeHandle (x : Node) : Scope;`

  This procedure is given an arbitrary node **x** lying in a context which has been previously processed using **Declare** and returns the scope information necessary for fast access to visible names. If the surrounding context contains errors such as a with-statement with an invalid designator or a module not previously declared, **NIL** is returned.

Now we need a procedure which will map names to defining occurrences. The following procedure provides this facility.

- `PROCEDURE DefiningOccurrence (x : Node s : Scope) : Node;`

  This procedure is given a node **x** which is a **Name** and a scope handle **s** and returns the defining occurrence of **x** in **s**. **DefiningOccurrence** returns **NIL** if **x** is not defined in **s**.

The call `DefiningOccurrence(x,ScopeHandle(x))` will return the defining occurrence of **x** in its context except for two special exceptions in which **x** is itself a defining occurrence:

1. The case where **x** is the defining occurrence of the name of some kind of module is an exception because the name of a module is not visible in that module; **ScopeHandle** returns the scope for things visible inside a module when applied to the name of a module.
2. Parameter names of a procedure header are an exception because these names are not used anywhere and thus are not visible anywhere. The parameter names of a procedure header are just place-holders so that the syntax of a procedure header looks like that of a procedure declaration.

An extremely useful facility is the ability to check whether a given statement or expression is context-sensitively correct in a given scope. This facility is provided by the following procedure.

- **PROCEDURE CorrectQ (x : Node; s : Scope) : BOOLEAN;**

  This procedure checks the context-sensitive correctness of the statement or expression **x** in the scope **s** and returns the result of this check. Any errors encountered are marked with comments.

To check the correctness of an exit-statement it must be determined that the exit-statement has an enclosing loop-statement. This is the only situation where the node to which **x** is attached is examined. In all other cases the required contextual information is summarized by the scope argument **s**. In other words, for all cases except exit-statements it doesn't matter whether or not **x** is actually attached to some context since it will be treated as if it were in the context of the scope given by **s**. It is beyond the scope of this documentation to give a detailed description of what constitutes a correct expression or statement.

4 The Type of an Expression

With the exception of an expression that is just a type name (as in type-name arguments to the standard procedures **MAX**, **MIN**, **SIZE** and **VAL**), an expression has a type. However, the type of a constant expression may be ambiguous and depends on the context in which the expression is used. The type of an expression determines the kind of values that can be represented and the context in which the expression may be used.

4.1 Constants

Type transfer is not defined on constant arguments since the result of a type transfer is dependent on internal representation, which is unspecified for a constant argument. Thus there are only the following kinds of constants:

1. Integer constants represent the values of signed whole numbers. An expression which is an integer constant may be said to be of type **INTEGER** if the value of the expression is between **MIN(INTEGER)** and **MAX(INTEGER)**. The same can be said for **CARDINAL** and **LONGINT**. Integer constants have an ambiguous type.
2. Decimal constants represent the values of signed real numbers. An expression which is a

decimal constant may be said to be of type **REAL** if the value of the expression is between **MIN(REAL)** and **MAX(REAL)**. The same can be said for **LONGREAL**. Decimal constants have an ambiguous type.

3. **NIL** is used to represent the nuil value for all pointer and opaque types.
4. Character constants are strings of length zero or more. A character string of length one can be said to be of type **CHAR**. A character string can be used to represent a constant value for a character array. Character constants have an ambiguous type.
5. Set constants have an unambiguous set-type and may represent a constant value for only that type.
6. Enumerated values (including **TRUE** and **FALSE**) also have an unambiguous type, namely, the type in which they were defined.

We have two routines for dealing with constant expressions:

- **PROCEDURE ConstantQ (x : Node; s : Scope) : BOOLEAN;**

  This procedure determines whether or not the expression **x** represents a valid constant in **s**.

- **PROCEDURE ExpressConstant (x : Node; s : Scope) : Node;**

  This procedure returns the result of evaluating the constant expression **x** in **s**. **NIL** is returned if **x** is not a valid constant. This procedure may make global changes to the declarations using **GetVisibleName** in order to make the required names visible.

See subsection 2.4.3 in the main body of the thesis for examples.

4.2 Types

An expression which is neither a type name nor an ambiguously typed constant has a unique type. The type of an expression is represented by the node which defines the type, much like the meaning of a name is determined by its defining occurrence.

- **PROCEDURE GetType (x : Node; s : Scope) : Node;**

  This procedure returns the node which gives the type of the expression **x** in **s**. It returns **NIL** when applied to a type name, an ambiguously typed constant or an invalid expression.

Most often, **GetType** returns a node which is derived from the grammar production for **Type**. For example, suppose we have the variable declaration **x : ARRAY CHAR OF ARRAY BOOLEAN OF INTEGER;**.

The type of `x["a"]` is given by the node **ARRAY BOOLEAN OF INTEGER**. This also demonstrates the reason why the short hand notation **ARRAY CHAR,BOOLEAN OF INTEGER** is not permitted, since such a declaration would not allow us to return a type node which denotes the type of `x["a"]`. Remember that all the standard identifiers also have type definitions which **GetType** can return.

There are two special cases where the node returned by **GetType** will not be a node derived from the **Type** grammar production. The first case is the situation where the expression is the name of an open-array parameter, in which case **GetType** will return the parameter-section which defines the type of the parameter. The second case is the situation where the expression is a procedure name, in which case **GetType** will return the profile of the procedure.

- **PROCEDURE** `GetTypeName (x : Node; s : Scope) : Node;`

  This procedure works like **GetType** except that the name of the expression's type is returned. A type name is automatically generated if the type does not already have a name. It returns **NIL** for the same cases as **GetType** as well as for the case in which **x** is an open-array or a procedure name, because these have types which cannot be named. **GetVisibleName** is used to make type names visible.

See subsection 2.4.3 of the main body of the thesis for examples.

5 More Symbol Table Operations

5.1 Fields of Records

- **PROCEDURE** `QualifyingDesignator (x : Node; s : Scope) : Node;`

  This procedure is given a node **x** denoting a field in **s** and returns the designator of which **x** is a component. **NIL** is returned if there is no qualifying designator.

This procedure returns the designator of the with-statement which qualifies **x**, or the record of the component variable enclosing **x**. It is most useful in the first case, since the second case can be handled syntactically (i.e. RecordOf(Parent(x))).

5.2 Name Uniqueness, Renaming and Name Visibility

- **PROCEDURE GenerateName (x : StringType) : Node;**

  This procedure returns a name which is guaranteed to be different from any other name already declared. Any trailing digits in the string **x** are ignored. A unique name is produced by appending a larger trailing number than appears in any other declared name with the string **x** as a prefix. For example, if the names **'a1'** and **'a13'** are defined somewhere then **GenerateName(Str('a'))** will return the identifier with the name **'a14'**.

- **PROCEDURE Rename (x,y : Node);**

  This procedure is given a defining occurrence node **x** and renames all occurences in the enclosing compilation unit containing **x** to be the new name **y**. Any conflicts created by the renaming are marked with comments.

- **PROCEDURE GetVisibleName (x : Node; s : Scope) : Node;**

  This procedure is used to return a name visible in **s** which has **x** as its defining occurrence. If the scope **s** contains the name layers of with-statements or component-variables, that part of the scope is ignored. This procedure may be applied to any defining occurrence which is not an enumerated value, a field or a module with unqualified exports. **NIL** is returned for erroneous situations in which the name cannot be made visible.

Although this last procedure sounds simple, it can produce a large number of changes to the declarations of the program. The name may need to be exported and/or imported across several modules and there may be name conflicts along the way which must be resolved. We shall describe how the name is imported/exported and how name conflicts are resolved.

To make a name visible in an arbitrary scope, the name must first be exported out to a procedure or module which encloses both the defining occurrence of the name and the scope in which the name is to be made visible. Notice that a name cannot be made visible outside its containing procedure, so a name cannot be made visible everywhere. If a qualified name is to be exported, the name of its left-most qualifying module is exported. If there is no existing export declaration, an unqualified export statement is introduced. Adding a name to an export declaration makes it visible (with qualification if it is exported qualified) everywhere the name of the exporting module is visible, and this may create conflicts. Once the name is exported out as far as need be, it is imported inward to the required destination scope.

Name conflicts must be resolved to maintain a correct program. However, we cannot simply rename any conflict that comes along since the names used for definition modules and the procedures and variables they contain are important for linkage between separate compilations. When a name conflict is encountered, the object being made visible is renamed, unless it is from a definition module, in which

case the object with which it conflicts is renamed, unless it too is from a definition module, in which case a qualified version of the name is made visible and the unqualified name is eliminated. Examples are given subsection 2.4.3 of the main body of the thesis.

5.3 Runtime Information

Most program transformations do not rely on information related to runtime storage allocation. However, if object code is to be generated for a given program, the size of the address space for local variables and the offsets of variables within that space are very important. Two routines have been supplied to provide this kind of information.

- **PROCEDURE ActivationRecordSize (x : Node; s : Scope) : INTEGER;**

  This procedure is given a node **x** denoting a procedure in **s** and returns the size in bytes of the space needed to store that procedure's local variables.

- **PROCEDURE Offset (x : Node; s : Scope) : INTEGER;**

  This procedure is given a node **x** denoting a variable, parameter or field in **s** and returns the offset of the variable or parameter from the base of the activation record, or the offset of the field from the base of its record.

The information returned by the above two procedures is dependent on the machine for which code is to be generated. See Appendix C for hardware related details.

6 Metaprogrammer Extensions

Metaprogrammers may wish to cerate extensions to the type-checking mechanism of the symbol table. The procedures **UserExpressionCheck** and **UserStatementCheck** have been provided to serve this purpose. These routines allow the metaprogrammer to over-ride the type checking of statements and expressions before the standard checks are applied.

To create an extension the metaprogrammer must supply two routines. The first routine is a recognition routine which takes a node and its scope handle as arguments and returns **TRUE** if the extension applies to the given node. The second routine is called with the same arguments as the first routine and is called only if the first routine returns **TRUE**. It is responsible for checking the correctness of the node and for expression nodes it must also return the defining occurrence of the type name of the node. The checking routine calls **Error** to mark any errors that it detects.

- **PROCEDURE Error (Location : Node; Comment : StringType);**

  This procedure is used to mark errors with a **Comment**. The error comment is placed as close as possible node to the node given by **Location**. This procedure is in particular meant to be used by the user-defined checking routines to communicate to the symbol table the fact that an error has been encountered.

- **PROCEDURE UserExpressionCheck (p1 : Recognize; p2 : TypeChecker) : Node;**

  This procedure is given two procedures as parameters which will be installed in a list of other such procedures. Whenever and expression is to be checked or the type of an expression is to be determined by the symbol table, the recognizer **p1** is applied. If it returns **TRUE** the type checker **p2** is applied to return the defining occurrence of the type name of the expression. The procedure **Error** is called by **p2** to indicate where errors have been encountered. The type name returned by **p2** is ignored if **Error** has been called. The recognition routines are applied in the order in which they are installed by calls to **UserExpressionCheck**, and standard checks are performed only if no recognizer returns **TRUE**.

Notice that type names must be returned as opposed to just types. This means that metaprogrammer-extended expressions can only have types which are named. This restriction is not very severe since the same restriction applies to the types of parameters and the return types of function procedures. The returned type name must be a defining occurrence because the name might not be visible in the scope of the expression being checked.

- **PROCEDURE UserStatementCheck (p1 : Recognize; p2 : Checker) : Node;**

  This procedure works analogously to **UserExpressionCheck** except that no type name is returned by **p2** because statements don't have types.

The above routines are extremely general and may be used in very uncontrolled ways to make arbitrary changes to the standard Modula-2 semantics. It must be stressed that these routines should be used carefully. There may be subtle interactions between extensions and with existing aspects of the type checking system. One of the main applications of user extension is the definition of procedures with special kinds of parameters such as a procedures which take an arbitrary number of arguments or arguments requiring special type checking.

APPENDIX C - CODE GENERATION

1 Introduction

This appendix describes code generation for simplified Modula-2 programs. Sun Assembler code is generated for an MC68020 with an MC68881 floating point coprocessor. It is beyond the scope of this documentation to describe in detail all relevant aspects of the assembler and the underlying hardware; refer to the references for details [20, 28]. The generated code was designed to be compatible with that produced by the Pascal and C compilers, making it possible to link Pascal and C routines with Modula-2 programs.

We will first give the definition module containing the routines to be described and then proceed with a brief discussion of the hardware and the assembler.

```
DEFINTION MODULE Generate;

  FROM String IMPORT Text, StringType;
  FROM m IMPORT Node;
  FROM Symbol IMPORT Scope, Recognize;

  TYPE
    Generation = PROCEDURE (Node,Scope);
    MakeSpecification = PROCEDURE (Node,Scope) : StringType;

  PROCEDURE Modula2ToSunAssembler (f : Text; x : Node);
  PROCEDURE Assembler1 (x : StringType);
  PROCEDURE Assembler2 (x,y : StringType);
  PROCEDURE Assembler3 (x,y,z : StringType);
  PROCEDURE AssemblerLabel (x : StringType);
  PROCEDURE Op (x : Node; s : Scope ) : StringType;
  PROCEDURE Gen (x : Node; s : Scope);
  PROCEDURE UserGen (p1 : Recognize; p2 : Generation);
  PROCEDURE UserOp (p1 : Recognize; p2 : MakeSpecification);

END Generate.
```

2 Overview

The Sun-3s for which code is generated have: eight address registers, `a0` through `a7`; eight data registers, `d0` through `d7`; and eight floating point registers, `fp0` through `fp7`. Memory is byte-addressable and there are no alignment restrictions.

Register `a7` is used as the stack pointer and the stack starts in high memory and grows downward. The stack pointer always points at the last byte on the stack and there are predecrement and postincrement addressing modes for efficient stack manipulation.

Register `a6` is used as a frame pointer. The frame pointer always points into the stack and just below it is the address space for the local variables. The frame pointer points at the value of the old frame pointer and above that are the return address, display pointers and finally the parameters. The parameters are addressed with positive offsets from the frame pointer and local variables with negative offsets. Parameters are pushed onto the stack in reverse order.

Registers `d0` and `fp0` are used for the return values of functions and the results of expressions. Register `fp0` is used for floating point results and `d0` is used otherwise. Registers `d1`, `a0` and `a1` are used as temporaries for address calculations and other things. The remaining registers are available for general use but the code generator does not use them.

The most important concept of concern in the assembler is that of effective address specification. The MC68000 family have very uniform address modes which can be used with most instructions. These modes are adequately described in the references and are too numerous to discuss here. The description for `Op` gives some examples of the kinds of operand specifications that are used.

3 The Translator

- **`PROCEDURE Modula2ToSunAssembler (f : Text; x : Node);`**

  This procedure writes the Sun Assembler equivalent of the program module `x` to the file `f`.

The program module `x` must be in simplified form. The only statements allowed are assignments, procedure-calls, single-clause if-statements, simple-loops, exit-statements, return-statements and null-statements. Expressions must be simplified as described for **`LinearizeExpression`** in Chapter 3.

The result type of functions must be small enough to fit in an appropriate register.

There is no code generation for definition modules because all the required code for them is generated when their corresponding implementation module is compiled. The assembly language programs generated by **Modula2ToSunAssembler** may subsequently be assembled and linked using the Pascal compiler "pc".

4 Metaprogrammer extensions

The following routines can be used by the metaprogrammer to define special purpose extensions to the code generator. The extension mechanism is analogous to the one used to extend the symbol table checking mechanism.

- **PROCEDURE UserOp (p1 : Recognize; p2 : MakeSpecification) : StringType**

  This procedure is given two procedures as parameters which will be installed in a list of other such procedures. Whenever an operand specification is to be generated, the recognizer p1 is called with a constant or (possibly type-transferred) designator and its scope handle as arguments. If it returns **TRUE**, p2 is called with the same arguments to return the specification of the effective address of the argument. The recognition routines are applied in the order in which they are installed by calls to **UserOp**

The routine p2 which returns an operand specification, may generate instructions to calculate the effective address as a side-effect, just as Op does. The way registers are used should be considered very carefully to make sure that contlicts do not arise. For example, register d0 should not be used since important values may be stored in d0 when an operand specification is generated. One important application for user extension is the definition of register variables so that certain variables will actually map to registers.

- **PROCEDURE UserGen (p1 : Recognize; p2 : Generation);**

  This procedure is analogous to **UserOp**. Whenever code is to be generated for an expression or statement (not a designator or constant), the routine p1 is called with the expression or statement and its scope handle as arguments. If it returns **TRUE**, p2 is called with the same arguments to generate the assembler code.

The following four procedures are used to write Sun Assembler instructions out to the file given by the call to **Modula2ToSunAssembler**. Each call to any of the four procedures produces a new line of output in the assembly output file.

- **PROCEDURE AssemblerLabel (x : StringType);**

This procedure writes the string **x** immediately followed by a colon.

- **PROCEDURE Assembler1 (x : StringType);**

This procedure writes the string **x**.

- **PROCEDURE Assembler2 (x,y : StringType);**

This procedure writes the string **x**, some spaces, and the string **y**.

- **PROCEDURE Assembler3 (x,y,z : StringType);**

This procedure writes the string **x**, some spaces, the string **y**, a comma, and the string **z**.

The following two routines can be used by the metaprogrammer to employ existing code generation facilities in the extensions that he writes. These are precisely the routines that can be extended by calls to **UserOp** and **UserGen**.

- **PROCEDURE Op (x : Node; s : Scope) : StringType;**

This procedure is used to return the operand specification for (possibly type-transferred) designators and constants. Assembler instructions for calculating the effective address may be generated as a side-effect. Registers **a0**, **a1** and **d1** may be used.

Depending on the kind of argument given to **Op**, certain registers may or may not be used. Registers **a1** and **d1** are used only if the argument to **Op** is a designator containing non-constant indices. Register **a0** is used only if the argument to **Op** is a designator containing a reference variable, an indexed variable with non-constant indices, a var-parameter, an open-array parameter or a non-local non-global variable. (Global variables are those variables which are not enclosed by a procedure.)

A few examples of what **Op** does will be given here, but to get a good idea of how both **Op** and **Gen** work, just apply the routines to simple examples and examine the output. When **Op** is applied to an integer constant such as **1234** it returns **#1234**. Applying **Op** to a variable named **x** in a program module named **m** returns **_m.x**. Applying **Op** to a local variable of a procedure returns **a6@(-offset)**. Applying **Op** to a global pointer variable named **p** in a program module named **m** results in **a0@** with the instruction **movl _m.p, a0** begin generated as a side-effect.

PROCEDURE Gen (x : Node; s : Scope);

This procedure is used to generate assembler code for expressions and statements. Expressions leave their result in register **fp0** if their result is **REAL** or **LONGREAL**, and in register **d0** otherwise.

REFERENCES

1. Aho, A.V., and Ullman, J.D., *Principles of Compiler Design*, Addison-Wesley, Reading, Mass., 1977.

2. Bauer, F.L., Broy, M., Gnatz, R., Hesse, W., Kreig-Bruckner, B., Partsch, H., Pepper, P., Wossner, H., "Lecture Notes in Computer Science", *Towards a wide spectrum language to support program specification and program development*, Springer-Verlag, Berlin, 1979, pp. 543-552.

3. Brown, P.J., '"My system gives excellent error messages" -- or does it?', *Software-Practice and Experience*, Vol. 12, No. 1, January 1982, pp. 91-94.

4. Cameron, R.D., "Grammar-based definition of metaprogramming systems", *ACM Transactions on Programming Languages and Systems*, Vol. 6, No. 1, January 1984, pp. 20-54.

5. Cameron, R.D., *MPS Reference Manual*, SFU CMPT Internal Document, 1987.

6. Chirica, L.M., Martin, D.F., "Toward Compiler Implementation Correctness Proofs", *ACM Transactions on Programming Languages and Systems*, Vol. 8, No. 2, April 1986, pp. 185-214.

7. Darlington, J., and Burstall, R.M., "A system which automatically improves programs", *Acta Informatica*, Vol. 6, No. 1, Jan 1976, pp. 41-60.

8. Delisle, N.M., Menicosy, D.E., and Schwartz, M.D., "Viewing a programming environment as a single tool", *SIGPLAN Notices*, Vol. 19, No. 5, May 1984, pp. 49-56.

9. Freak, R.A., "A Fortran to Pascal translator", *Software-Practice and Experience*, Vol. 11, No. 7, July 1981, pp. 717-732.

10. Goos, G., Wulf, W.A., Evans, A., and Butler, K.J., "Lecture Notes in Computer Science 161", *DIANA: An intermediate language for Ada*, Springer-Verlag, Berlin, 1983.

11. Huang, J.C., "Program instrumentation and software testing", *Computer*, Vol. 11, No. 4, April 1978, pp. 25-32.

12. Johnson, C.W., and Runciman, C., "Semantic Errors -- Diagnosis and Repair", *SIGPLAN Notices*, Vol. 17, No. 6, June 1982, pp. 88-97.

13. Kilian, M.F., "A conditional critical region preprocessor for C based on the Owicki and Gries scheme", *SIGPLAN Notices*, Vol. 20, No. 4, April 1985, pp. 42-57.

14. Kernighan, B.W., Ritchie, D.M., *The C Programming Language*, Prentice-Hall, 1978.

15. Knuth, D.E., "Structured programming with go to statements", *ACM Computing Surveys*, Vol. 6, No. 4, December 1974, pp. 261-301.

16. Leeson, J.J., Spear, M.L., "Type independent modules: The preferred approach to generic ADTs in Modula-2", SIGPLAN Notices, Vol. 22, No. 3, March 1987, pp. 65-70.

17. Loveman, D.B., "Program improvement by source-to-source transformation", *Journal of the ACM*, Vol. 24, No. 1, January 1977, pp. 121-145.

18. Maher, B., and Sleeman, D.H., "Automatic program improvement: variable usage transformations", *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 2, April 1983, pp. 236-264.

19. Medina-Mora, R., Feiler, P.H., "An incremental programming environment", *IEEE Transactions on Software Engineering*, Vol. 7, No. 5, September 1981, pp. 472-482.

20. Motorola, *MC68020 32-Bit Microprocessor User's Manual*, Prentice-Hall, Englewood Cliffs, N.J., 1984.

21. Partsch, H., and Steinbruggen, R., "Program transformation systems", *ACM Computing Surveys*, Vol. 15, No. 3, September 1983, pp. 199-236.

22. Pelacani, G., "A software development system based on a macro processor", *Software-Practice and Experience*, Vol. 14, No. 6, June 1984, pp. 519-531.

23. Pepper, P., *Program Transformation and Programming Environments*, Springer-Verlag, Berlin, 1984.

24. Power, L.R., "Design and use of a program execution analyzer", *IBM Systems Journal*, Vol. 22, No. 3, 1983, pp. 271-294.

25. Rosenblum, D.S., "A methodology for the design of Ada transformation tools in a DIANA environment", *IEEE Software*, Vol. 2, No. 2, March 1985, pp. 24-33.

26. Ruiz-Huerta, G., "The programmable compiler", *Computer*, Vol. 16, No. 3, March 1983, pp. 35-39.

27. Slape, J.K., and Wallis, P.J.L., "Conversion of Fortran to Ada using an intermediate tree representation", *Computer Journal*, Vol. 26, No. 4, 1983, pp. 344-353.

28. Sun Microsystems, *Assembly Language Reference Manual*, Revision F of 17, February 1986.

29. Teitelbaum, T., Reps, T., and Horwitz, S., "The why and wherefore of the Cornell Program Synthesizer", *SIGPLAN Notices*, Vol. 16, No. 6, June 1981, pp. 8-16.

30. Teitelbaum, T. and Reps, T., "The Cornell Program Synthesizer: a syntax-directed programming environment", *Communications of the ACM*, Vol. 24, No. 9, September 1981, pp. 563-573.

31. Turba, T.N., "A facility for the downward extension of a high-level language", *SIGPLAN Notices*, Vol. 17, No. 6, June 1982, pp. 127-133.

32. Wirth, N.W., *Programming in Modula-2*, Third Corrected Edition, Springer-Verlag, Berlin, 1985.

33. Wu, J.C.L., *Program Debugging with Toolkits*, Masters Thesis, in preparation, 1987.