1989

Implementation of a Modula 2 subset compiler supporting a 'C' language interface using commonly available UNIX tools

Raymond Shear F.

Rochester Institute of Technology
School of Computer Science And Technology

Implementation of A Modula 2 Subset Compiler Supporting a 'C' Language Interface Using Commonly Available UNIX Tools

by
Raymond F. Shear, Jr.

A thesis, submitted to
The Faculty of the School of Computer Science and Technology,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science.

Approved by:

Professor Peter Lutz 4/17/89

Professor Andrew Kitchen 4/17/89

Professor Peter Anderson 17 Apr 89

April 2, 1989

Implementation of a Modula 2 Subset Compiler

Supporting a 'C' Language Interface

Using Commonly Available UNIX Tools

I Raymond F. Shear, Jr. hereby grant permission to Wallace Memorial Library, of RIT, to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Raymond F. Shear

April 19, 1989

April 19, 1989

ACKNOWLEDGEMENTS

This document represents not just the efforts of its author but the intellectual, emotional and spiritual support of a great number of people over the duration of the thesis project and the educational experiences which have lead up to it.

My committee has been unstinting in giving me time and encouragement as well as steering me toward resouces critical to the completion of the task at hand. Professors Lutz, Kitchen, and Anderson never failed to return my calls or schedule appointments when the need arose for a face to face encounter. Their interest in seeing me succeed at this task has had significant positive effect on the outcome.

In a similar vein, the administrative staff in the Graduate Computer Science Department has saved me innumerable times from difficulties caused by my inattention to the details of my relationship with the school.

The Compter Science Librarian, Mrs. Bower, provided a gold mine of references which took many months to consume. Her efficiency at computerized searches of the literature make one wonder if perhaps she is as much database expert as librarian.

Mark Lessard and Bill Eign kept a very fragile computer alive long enough to finish programming and documentation. Without them I would be without several thousand dollars or without a degree (or both).

Special consideration must be given to the engineering staff of EDICON who took a green and reluctant Pascal programmer and produced a 'C' programmer able to write semantic routines for a Modula 2 compiler.

I would also like to thank Professor Heliotis for his timely comments on Augmented Linear Form.

But most of all, gratitude must be expressed to Eileen and Greg who waited more or less patiently while energies which rightly should have been focused on them were spent in dry, intellectual pursuits. Maybe now we can take that canoe trip....

ABSTRACT

Modula 2 has been proposed as an appropriate language for systems programming. Smaller than PASCAL but more structured than 'C', Modula 2 is intended to be relatively easy to implement. A realization of a subset of Modula 2 for the MC68010 microprocessor is presented. Widely available UNIX tools and the 'C' language are used for the implementation. A mechanism for calling 'C' language functions from Modula 2 (and vice versa) is suggested. Critical source code, grammar, and an extensive bibliography pertinent to the implementation are included as appendices.

Categories and Subject Descriptors:

D.3.4 **[Programming Languages]**: Processors - compilers, parsing,translator writing systems. D.2.2 **[Software Engineering]**: Tools and techniques. D.3.3 **[Programming Languages]**: Language Constructs. D.1.3 **[Programming Techniques]**: Concurrent Programming

General Terms: Languages

Additional Keywords and Phrases: Symbol Tables

TABLE OF CONTENTS

I. Introduction....4

II. Functional Specification....5

III. Architectural Design....6

IV. Module Design....8

A. Processor Modules....8

0. System Driver....9

1. Lexical Analysis....10

2. Parsing....11

3. Scoped Symbol Management....16

Symbol Processing Overview....16

Demands of Modula 2 on a Symbol Processing Mechanism....18

The Current Symbol Processing Mechanism....22

4. Intermediate Tree Construction....26

5. Initial Treewalk....31

6. Semantic Checking and Tree Rewriting....37

a. Type Checking....39

b. Translation of Complex Names to Address Arithmetic....41

c. Transformation of BOOLEAN Expressions to Jump Code....42

7. Preparation for Code Generation....48

8. Code Generation....55

9. Separate Compilation Facilities....67

10. A Digression on the Linked List Utilities....71

11. Implementation Notes....73

B. Data Bases....73

V. Verification and Validation....74

A. Lexical Analyzer....74

B. Parser....75
C. Symbol Table....75
D. Second Pass....76
E. Preperation for Code Generation....77
F. Code Generation....78
VI. Conclusions....79
A. Paths Not Taken....79
B. Things that Would be Done Differently....79
C. Suggestions for Further Work....81
VII. Bibliography....83
Appendix A -- Lex Rules
Appendix B -- Yacc Grammar (without actions)
Appendix C -- Symbol Table Design Document
Appendix D -- Symbol Table Types Include File
Appendix E -- Tree Types Include File
Appendix F -- Attribute and Type Descriptor Include File
Appendix G -- 'C' Language Interface

Table of Figures

Figure		Page
1	grammar fragment for designator	14
2	forward reference example	20
3	multiple definition example	21
4	symbol mechanism schematic	>23
5	identifier data flow	>26
6	block structure tree	>27
7	a simple tree building example	29
8	prototype list creation semantic action	29
9	constant expression example	31
10	a nested scope example	32
11	a postorder traversal of the block tree	35
12	illustration of need for runtime check of priority	37
13	yacc rule for the Modula 2 statement	40
14	a complex designator instance	40
15	example of need for short circuit evaluation	42
16	tree for (A OR B) OR (C AND D)	43
17	code generated from figure 16	44
18	direct generation of jump code from expression of figure 16	45
19	dataflow in boolean expression tree structure	46
20	a labeling algorithm for jump codes	48
21	'C' language algorithm for labeling treenodes	51
22	tree for c := a.f1[32, c+d]^.f2	52

23 address arithmetic tree for a.f1[32, c+d]^.f2.....>52

24 code generation algorithm for arithmetic trees.....59

25 'C'-like pseudocode for boolean expression code generation.....62

26 A Modula 2 Procedure Stack Frame.....>64

I. INTRODUCTION

A would-be compiler writer intending to create a language processor for a modern Algol-like language is faced with a peculiar difficulty. Despite steady advances in the compiler writer's art over the past decade, it is nearly impossible to find a complete example of a full language implementation using widely available tools and bottom up parsing methods. The project herein described was begun with the intent of collecting, in one document, all of the tools, techniques, and references required to create a nearly complete bottom up implementation of Modula-2.

Modula-2 was chosen because it is (allegedly) easier to implement than other Algol-like languages; because it is interesting in its own right as a language; and because it has great utilities for those researchers wishing to use a structured language for systems programming.

UNIX is the inevitable choice for a development environment because the most popular tools for creating compilers are available under UNIX and portability was a powerful factor in the choice of methodology. At the outset it was supposed that these mature and stable tools (lex, yacc, "C" language, and possibly prep) were well documented and easy to use. However, aside from the ubiquitous desk calculator created using UNIX language tools and the original papers on lex [LESK 75] and yacc [JOHN 75], very little meaningful documentation on the use of these tools was extant when this project was begun. Existing documentation and ease of use of development tools were not the primary consideration.

The central theme and purpose of the report to follow is the elucidation of the implementation of Modula 2 using the chosen toolset. With the language specification in one hand and the many thousand available pages of compiler writing theory and practice in the other it is hoped that a path

to the realization of the language will emerge in the reader's mind. For those seeking the security of knowing THE WAY to implement Modula 2, this epistle will no doubt prove unsettling. But for anyone satisfied with A WAY the following discussion may prove adequate.

II. FUNCTIONAL SPECIFICATION

The language processor described here is intended to conform to the language specified by Wirth [WIRT 87]. Although the language implemented is a subset, the parser is not. That is to say, the lex rules and yacc grammar are complete Modula 2 (with the ETH "FORWARD" construct supported but not required). And while reals are not semantically supported, they are included in the lexical analyzer, the yacc grammar, and the symbol attribute structure.

The command syntax for compiling Modula 2 modules takes the following form:

```
mod2 [-oc] filename[ ".def" | ".m"]
```

When compiling definition modules, the ".def" extension is required. The ".m" extension is optional in the command line (it is supplied by the command line handler) but is required in the file name as it exists on secondary storage.

Flags to invoke optimization ("-o") and syntax checking only ("-c") are accepted. Optimization is a noop in the current version. The syntax only check goes as far as building the initial intermediate tree, but no farther.

III. ARCHITECTURAL DESIGN

At first glance the structure of the Modula 2 translator is very much what one would expect in a classical Unix based compiler. A front end built around lex and yacc constructs an attributed symbol table and intermediate representation. Several subsequent passes over the intermediate form ultimately end up producing code (for correct input strings).

Lex passes tokens and, in the case of literals and identifiers, places name strings on the value stack of the yacc based parser. As will be discussed in greater detail later, the current system differs from traditional techniques in that symbols are not entered into the symbol table during the scan.

Rather than produce tuples as the **Intermediate Representation** (IR), an **Abstract Syntax Tree** (AST) is constructed. Although trees as intermediate forms have been around for a long time, most authors of compiler texts focus their attention on the tuple form of IR. This no doubt reflects the paucity of available memory on early systems as well as the desire to produce code in one pass. This latter requirement has led to the development of a large body of theory concerning the various aspects of single pass translation. These developments range from methods of "on the fly" attribute evaluation to methods of backpatching jump labels in "if" statements. Because the specification of Modula 2 forces a translation to proceed in multiple passes, much of the literature in standard texts is somewhat less useful.

The loss of so many difficult techniques may *seem* like an empty tragedy. After all, things are so much simpler without them. The problem is that there appears to be very little extant literature on anything but single pass techniques. This view is not universally shared, however: A test by Anklam, et al declines to discuss the matter of tree based systems because of the

"extensive literature on the subject"! Few texts devote more than a half dozen pages to trees while hundreds are spent discussing the techniques associated with single pass compilation such as tuples, l-attributed and s-attributed grammars and so on. Much of the work discussed here was, for the foregoing reasons, created out of whole cloth. The attribute record structure and the expression tree code generator were discovered in Fischer and LeBlanc [FISC 88]. The former seemed like such a vast improvement over the version developed by this author that the persistent part of the symbol table mechanism was completely rewritten to accommodate it. Albeit expensive, this rewrite has proven cost effective and has the added virtue of making the persistent part of the Modula 2 symbol mechanism similar to that used in Ada implementations.

The IR for the current compiler is a tree and the process of translation is like an old fashioned European Christmas party wherein the yule tree [TANN 83][TANN 84] is decorated, redecorated, and ornaments moved and adjusted until the party is over (code is emitted).

Semantic actions called from within the parser build the AST and the attributed symbol table. An initial post parse walk of the AST rearranges lists of declared variables and inserts a "module call" node into the subtree of a module for each submodule below it (recursively). Modules are treated as special forms of parameterless procedures without local variables. This stems from 2 features of Modula 2: 1)All variables in a module exist at the same nesting level (NOT scope) as the variables of the surrounding scope; and 2) statement lists in a module must be executed before the statements lists of the surrounding scope. This reordering of the tree is simple but does require a separate pass (see section on initial treewalk).

The second tree walk performs a variety of type and other semantic checking as well as determining the Sethi-Ullman numbering of expression trees. The dual register set of the MC68000 family of computers make the implementation details of tree numbering a complex matter.

The third major pass over the AST emits MC68000 Assembler code.

A paper describing a Modula 2 compiler by Powell [POWE 84a] was the primary inspiration for this work. The final form of the current implementation differs markedly from Powell's in that the resolution of identifiers in statements occurs at the end of the first pass and no p-code form is emitted as a second intermediate form.

IV. MODULE DESIGN

A. **Processor Modules** - The major functional modules comprising the compiler system are: 1) the "lex based" lexical analyzer, 2) the "yacc based" parser, 3) the symbol table management system, 4) the intermediate tree construction system, 5) the second pass "treewalk" functions, 6) third pass checking and tree rewriting, 7) preparation of the intermediate tree for code generation, and finally, 8) code generation. To this list may be added the command line parser and multipass "driver" which coordinates the function of the entire system as well as facilities for separate compilation (including export and import). The linked list routines have proved extremely useful and are a ubiquitous part of the compiler. As such they merit a brief section of their own. The order of discussion will roughly parallel the order of invocation during the compilation process.

0. **System Driver** -- A 'C' language main program drives the compiler. Command line arguments are evaluated, initialization is performed, the parser is called, and depending upon flags from the command line, treewalks and code generation are called.

A function for evaluating the inline arguments uses the "-" character as a key, thus eliminating position dependence of command flags. Flags may appear before after or around the source filename. The command line parser was created in ignorance of the UNIX System V command line standards. The SYSTEM V PROGRAMMER'S GUIDE describes a set of command line parsing routines which may be substituted for those found in this compiler.

One of the initialization tasks is to load the configuration file. Pervasives such as "CARDINAL" and "CHAR" are defined and sized in the configuration process. The configuration file is created offline by running the program "buildper". Buildper relates the primitive Modula types to primitive 'C' types and emits a configuration file which contains the names, sizes and token identifiers of the pervasive types. Because buildper is written in 'C' we may be assured that there exists a one to one correspondence between primitive 'C' and Modula types. Ideally, by replacing the configuration file and the routine which emits code, one could retarget the compiler to any other Unix based machine. This is not strictly true in the current case but the tools and framework to ease this transition are in place. Making the attributes of pervasives completely table driven and nonprocedural would be an excellent research project.

Initialization also encompasses the creation and/or validation of symbol table and "with" stacks and the hash table. At this time the source file is opened and ".m" catenated to the name if no extension specifier is found.

1. **Lexical Analysis** -- A lex [LESK 75] based lexical analyzer extracts Modula 2 tokens from the input string. In classical fashion, the token is identified and the token type returned to the parser. Rather than install the symbol into a symbol table at this time, space is allocated for the string and an entry is created in a hashed name table. A pointer to the installed name is placed on the yacc value stack before the token type is returned to the parser.

   The lex rules conform to 3rd edition Modula 2 with the following extensions:

   a. The REM (remainder) token is supported. Wirth apparently added this to the grammar after the 3rd edition of the reference manual [WIRT 85a] was published (and after the first major extension was published [WIRT 85]). The reference does appear in the Modula 2 based revision to Wirth's Algorithms text [WIRT 86a].

   b. The FORWARD token is supported. Because the ETH compiler is a one pass, top down compiler, it labors under severe restrictions with respect to the ability to handle undefined forward references. Because other programs may take advantage of this undocumented relaxation of the forward reference specification the feature was included. The attribute records for procedures had to be extended to accommodate the FORWARD feature, but the lexical analysis was, of course, simple.

   c. The ETH compiler makes fewer assumptions about valid numeric literals than does the most recent language specification. For instance, the hexadecimal constant corresponding to decimal "13" must be written as

"DH". However, the ETH compiler takes advantage of the fact that the appearance of alphabetic characters in the range of A-F in an otherwise numeric string could be used to signal a lexical analyzer that a hexadecimal number is to be detected. Hence, in the ETH compiler, what would normally be written as "DH" according to the lexical specification of Modula 2 may be written as "0D". The current version of the lexical analyzer supports this extension. Supporting this new unwritten convention causes potential conflicts with previous formalisms. The 8 bit quantity corresponding to a decimal 2 is written "2C". The *implicit* conversion to hex would make this a decimal 44. The current system places the rules for byte and octal literals ahead of those for hexadecimal literals. By ordering the rules in this fashion, an alphanumeric string will be interpreted as hexadecimal only if it can be interpreted as no other form of numeric literal.

The only input type not handled by lex rules is the comment. Lex detects the beginning of a comment ("(*") and dispatches a recursive 'C' routine which strips the (possibly nested) comment from the input string. No token is returned in this case.

2. Parsing -- Yacc [JOHN 75] is used to construct a table driven bottom up parser for the compiler. The grammar was abstracted from the 3rd edition of Wirth with modifications to support the extensions described in the foregoing section on lexical analysis. Because Wirth favors top down parsing, his grammar specification is written with all left recursion removed. Right recursive grammars are space consuming when used with LALR parser generators such as yacc. The yacc grammar has thus been written as a left recursive version of the EBNF specified by Wirth. After the grammar was nearly complete an LALR specification by Spector came to light [SPEC 83]

and an attempt was made to use it as a basis for a yacc grammar. This proved to be a fruitless effort due in no small part to this author's lack of experience with yacc. The Spector grammar produced several hundred conflicts when used directly as a yacc grammar. The attempt to port the grammar was abandoned.

Because an Abstract Syntax Tree (AST) was the primary output of the parse phase of compilation, the task of writing a useful grammar was greatly simplified. A structure representing the program is extant at all times during the parse so attributes may effectively be inherited, synthesized, created or deleted at any time. This sidesteps the need for on the fly attribute evaluation which requires that the grammar, in the case of yacc, be s-attributed. Because of the permissiveness of Modula 2 with regard to use of undefined identifiers in statements, there exists no opportunity to compile an input string in one pass. If multiple passes are used, on the fly attribute evaluation is not required. The ultimate resolution of attributes may in some cases happen only after several passes over the AST constructed by the parser.

The current grammar produces no conflicts when processed by yacc. This was accomplished by taking some liberties with Wirth's EBNF specification of the language. There is necessarily a conflict between qualidents and designators. Consider the following fragment of Modula 2 EBNF.

```
qualident = ident { "." ident }.
designator = qualident {"." ident | "[" Explist "]" | "^" }.
```

It should be intuitively obvious to the most casual observer that no LR (or LL) parser can differentiate between a designator and a qualident when specified in this form. Yacc produces shift/reduce conflicts in this situation.

The problem, however, is more one of form than of substance. The intent of the specification for designators is to allow the alternative production { "." **ident** } to appear anywhere in the designator string after the *first occurrence* of "[" **Explist** "]" or "^". By allowing { "." **ident** } immediately after the qualident in the designator specification, a conflict is generated. Because yacc favors a shift in this case the conflict is "safe". But in the spirit of producing a clean grammar, the rules have been written to reflect Wirth's ultimate intent by removing the conflict. In retrospect, this was hardly worth the effort. See figure 1 for the full yacc representation of a designator.

The compiler sports one "feature" which is not in keeping with tradition; symbols are installed from the yacc grammar semantic actions rather than from the lexical analyzer. How this impacts the flow of information and control within the compiler is outlined in greater detail in the following section on scoped symbol management.

```
designator     :
                    qualident desigcomplist nonqualist
               |    qualident desigcomplist
               |    qualident nonqualist
               |    qualident
                    ;

desigcomplist :
                    desigcomplist nonqualist deslist
               |    nonqualist deslist
                    ;

deslist        :
                    deslist desident
               |    desident
                    ;

nonqualist     :
                    nonqualist nonqualitem
               |    nonqualitem
                    ;
```

```
nonqualitem :
                    '[' ra_elementlist ']'
                |   '^'
                ;

ra_elementlist      :
                    ra_elementlist CATELEM
                      explist
                |   explist
                    ;

desident        :
                    '.' IDENT
                    ;

qualident       :
                    IDENT qualimore
                |   IDENT
                    ;

qualimore       :
                    qualimore desident
                |   desident
                        ;
```

Figure 1: grammar fragment for designator

A new token type had to be introduced to prevent the generation of conflicts when parsing Pascal style array addressing. In Modula 2 and Pascal

```
a[x,y,z]
```

is exactly equivalent to

```
a[x,y][z]
```

and must be handled by the compiler. Yacc cannot find a way to tell whether an instance of array addressing is to be reduced or if more subscripting follows when it encounters ']'. A new token type, CATELEM, was created to handle this problem. CATELEM is ']' followed by any amount of white space followed by '['. Thus yacc need never know that an array was potentially terminating unless it actually had ended.

Yacc allows (and in the case of grammars for full blown languages, nearly forces) one to type the value stack. The Modula 2 yacc grammar used here defined 4 types as follows:

a. string pointers (char *)

b. list head pointers (listhead_t *)

c. tree node pointers (treenode *)

d. and, integers.

Yacc strictly enforces type checking and will not produce a parser if any type mismatch occurs in any rule. This is a valuable way to weed out potential errors but can produce mystical error messages in complex grammars with embedded actions. The complexity introduced by typing the value stack has a profound influence on compiler writers using yacc. There exists a strong tendency to use as few types as possible. The pressure to reduce the number of types can lead to the creation of large, intricate data structures. The compiler discussed here is no exception. Tree nodes are an amalgam of interior node, leaf, designator, list, and attribute. A more detailed discussion concerning the form of trees and their constituents will be found in the sections on intermediate tree construction and walking.

As mentioned above, value stack type checking can produce error messages which are difficult to interpret. These errors are most difficult to diagnose when embedded actions appear in the grammar. Because of this phenomenon, embedded actions have been virtually eliminated from the compiler.

3. Scoped Symbol Management

Symbol Processing Overview

The choice of a symbol processing mechanism has a profound influence on the structure and performance of a compiler. Because symbols are constantly being searched , installed, and researched during the parse the cost in time imposed by the symbol table routines can have a significant impact on the overall performance of the compiler. Symbol searches are primarily based on name strings but secondary search keys can be other attributes such as symbol type and scope. For instance, in Modula 2, a type and a variable may have the same identifier in a given scope. The symbol searching mechanism must be able to discriminate between a **type** occurrence and a **var** occurrence of an identifier.

Strategies for implementing the symbol processing mechanism range in complexity from loosely structured lists to general purpose database systems. Somewhere between these two extremes lie special purpose systems which attempt to balance complexity, performance, and generality in meeting the needs of a specific language.

Most popular among implementations of multiply scoped languages such as Pascal is the tree based design which was used in the original P-4 system [BARR 81]. Each scope in the P-4 compiler gets its own tree based symbol table. The scope hierarchy is searched starting at the bottom. If a symbol is not found in the tree at the current scope, the tree for the immediately enclosing scope is searched, and so on. For simple tree symbol tables without complicated access support, fixed space overhead may involve as little as one pointer per scope. Time efficiency depends on the access method used to get at the individual symbol in each tree. A b-tree would be efficient for large symbol sets but would be overkill for the number of symbols typically

associated with a scope. Finding the database associated with the enclosing scope could be time consuming.

Another approach to symbol table design is the hash table / symbol stack method [KNUT 73], [TREM 82], [FLOY 87]. Using this scheme, identifiers are hashed to find the head of a conflict chain in a pointer array. Symbols which hash to the same address are added to the conflict chain. Blocks in the conflict chain point to a symbol "stack". New symbols are added at the top of the stack. When a scope exit is encountered, all the symbols for that scope are removed from the symbol stack. One method for implementing scopes is to create a second scope stack. Each time a scope is entered, the address of the top of the symbol stack (pointer to the next free symbol) is pushed onto the scope stack. Thus the scope stack "marks" the scopes on the symbol stack. This mechanism works fine for single pass compilers where a procedure exit signals the end of code generation for that scope. However, for multiple pass implementations, symbols must be persistent and not disappear when the scope is exited during the parse.

Exactly what constitutes the best approach to the problem of symbol management in a compiler is not a dead issue. Many consider the hashed technique to be the best performer (in the time domain) [AHO 86] while others [FISC 88] find the same technique inadequate on grounds of performance when used in multiple pass compilers.

When implementing a language such as Modula 2 or Ada, one must also keep in mind the utility of maintaining a project-wide symbol database. This would prove useful for such features as incremental compilation, source control, and documentation for system maintenance. A truly general purpose database could use scope as part of the search key, or better, as a constraint on the view of the database which varied depending on context. The

development of Ada under the sponsorship of the Department of Defense has done much to further the notion of global symbol databases.

Symbol tables for a wide variety of languages may be constructed from a set of abstract specifications. With respect to high level specification of symbol databases, the work of Reis [REIS 83] on the PECAN project is generally considered to reflect the current state of the art [LAMB 87]. Reis describes a symbol processing language and gives a yacc grammar for same.

Demands of Modula 2 on a Symbol Processing Mechanism

Modula 2 allows for both open (procedures) and closed (modules) scopes. Open scopes have an unrestricted view of symbols declared in surrounding open scopes out to the most closely nested closed scope. That is if procedure "c" is enclosed in procedure "b" which is enclosed in module "a", then "c" may see all symbols in "a" and "b" but may see none of the symbols in scopes surrounding "a" (unless such symbols have been imported into "a"). The lexical nesting of open and closed scopes in unrestricted.

Closed scopes may be local (lexically adjacent and/or nested in the same file) or global (in separate files and separately compiled). Global modules share data and procedures through a **definition module** which defines the interface to an **implementation module**. All symbols in a definition module constitute the export list of that module. Definition modules are automatically (by action of the compiler) imported into implementation modules of the same name. An excellent discussion of closed scopes in Modula 2 is found in the text by Christian [CHRI 86].

Symbols imported from a definition module are qualified with the name of the definition module. This qualification may be removed by using the FROM clause. If the FROM clause is used the symbols in the import list

must be entered into the local symbol table. Otherwise they are accessed through a lookup of the module name.

In addition to nested open and closed scopes and their visibility rules, Modula 2 offers the symbol table designer other problem solving opportunities. Most significant among these is the permissiveness with respect to the use of undefined identifiers in statements (but not in declarations). This language feature dictates that Modula 2 be implemented in multiple passes. If multiple passes are used (as they must be) symbols persist long after closed and open scopes have been parsed. This is a pivotal issue in the realization of Modula 2 and has been a source of contention among language implementors for some time [GRAH 79].

The symbol table design is complicated not only by the need for persistence of defined symbols but by the need to propagate undefined symbols to the environment of enclosing scopes at open scope exit so that forward references may be properly resolved. It is clear that any symbols which remain undefined at the time of open scope exit must be copied into the new "current" scope. Any symbols remaining undefined at closed scope exit are considered declaration errors. This rule applies to ALL local closed scopes including the outermost scope. Imported undefined symbols will not be declared as part of the current scope and thus are immune from generating errors at the time of exit from scopes into which they were imported. Note that undefined forward references arise from 3 and only 3 conditions; 1) calls to procedures as yet undefined but at the same nesting level as the calling procedure, 2) EXPORT from lexically adjacent but as yet undefined local modules, and 3) definition of symbols used by a procedure which occur after the definition of the body of that procedure. Export lists in local module headers cause the scope of the symbols in those lists to expand into

the surrounding scope. To get a sense of the difficulties which may arise because of forward references, consider figure 2.

```
PROCEDURE foo;

TYPE
  a : RECORD
    f1 : INTEGER;
    f2 : CARDINAL;
   END a.

VAR
  v : CARDINAL;
  a : a;

BEGIN
  WITH a DO
    WITH b DO
      v := v + f1;
      v := v * f2;
    END; (* with b *)
  END; (* with a *)
END foo;

MODULE LocalModule;
EXPORT b;
TYPE
  a : RECORD
    f1 : CARDINAL;
    fld2 : POINTER to INTEGER;
  END (* record *);

VAR
   b : a;
   etc.
```

Figure 2: forward reference example

Clearly, the assignment statements in the WITH clause of PROCEDURE foo require careful symbol table handling. The control variable "b" is undefined at the time that the statement is parsed, yet the designator "f1" in the first assignment of procedure foo MUST refer to "LocalModule.b.f1". Designators in WITH statement lists cannot be bound to symbol table entries as long as any control variables in any enclosing WITH clauses are undefined. Even though instances of v, f1, and f2 may be found in the symbol table as it

exists at the time the WITH statement is parsed, fields of "b" are preferred candidates for the resolution of all of these designators. A list of all candidates for name resolution in the WITH statement must be provisionally bound to all symbols in the statement if and only if any of the names in the WITH clause are undefined at the time the enclosing WITH is parsed. The WITH statement provides the most difficult conundrum for symbol table design. The lengths to which any given implementation will go to resolve these difficulties appears highly variable [CHRI 87]. The strategies used here are discussed in detail in a later section.

A feature which Modula 2 shares with Pascal is the requirement that a variable and a type of the same name not cause a double definition error. Consider the program fragment in figure 3.

```
MODULE MainGuy
  TYPE
    a : CARDINAL;
  VAR
    a : a;

  PROCEDURE SubGuy
    TYPE
      b : a;
    VAR
      a : b;
```

Figure 3: multiple definition example

Notice that the same symbol ("a") is legally defined three times within the module. The symbol processing mechanism must accommodate this requirement.

Type transfer functions present an interesting challenge with regard to symbol type discrimination. A type transfer function may appear as a designator anywhere a function call is valid. If a function call designator is not found on the first lookup, the symbol search routines must research the

symbol table with new criteria (type rather than procedure). To make matters even more difficult, it appears that the Modula 2 specification allows types to be undefined in statements just as any other symbol. Thus, great care must be exercised in resolving forward references of procedures.

The Current Symbol Processing Mechanism

Armed with the opportunities for creative software engineering outlined above, we are now sufficiently prepared to discuss the approach taken in the current design. At the outset it must be said that the level of effort to implement the symbol table was far greater than had been anticipated and even Wirth [WIRT 77] has acknowledged that a satisfactory implementation is a surprisingly complex task. Undefined forward references are the prime culprit.

The driving force in chosing a method is sufficiency: Will the design accommodate all aspects of the language? Performance is a secondary but important factor. The only real choice was between tree based tables and a hashed global table. Tree implementations seemed primitive in that a worst case binary tree unfolds to a linked list. Uplevel references required accessing multiple tables and researching on failure. All in all the tree based approach does not appear to offer good performance.

A hashed global table has the advantage of providing fast lookup of all visible symbols if some sort of constraint can be imposed on the view of the access mechanism. The paper by Graham [GRAH 79] describes such a mechanism and provides confidence that a design using hashed tables is sufficient to the needs of a multipass Modula 2 compiler. The central idea of this scheme is to add new symbols to the conflict chain at the head of the list. This speeds lookup and allows multiple occurrences of name (from

different scopes) in the same chain. The first symbol encountered is the "correct" instance even though other instances may occur farther down the chain. Appendix C contains a design document outlining the actions taken with respect to the symbol table at various points in the parse. Figure 4 illustrates the relationship between the name hash table, the symbol hash table, the conflict chains, the symbol stack and the scope stack.

When a symbol is encountered for the first time either through a declaration or a forward reference, it is entered into the scoping mechanism (hash table, name table, and symbol stack). At the same time a persistent object consisting of an attribute treenode (leaf) is created and referenced by the symbol stack entry. This persistent object will ultimately be bound to the abstract syntax tree if it is ever used in a statement or referenced by another declaration (as in type declarations).

When a symbol definition is detected by the parser, a symbol installation routine is called. Symbol installation hashes the name and searches the conflict chain of the symbol hash table entry for duplicate occurrences of the name. If duplicates occur the nesting level of the scopes for these names is checked. If two identical names occupy the same scope or one of them is PERVASIVE (in the outermost scope surrounding all user defined symbols) then a definition error occurs. The current implementation uses volatile symbol and scope stacks. The symbol stack is little more than an array of pointers to persistent data objects. The casual observer may feel that a stack is superfluous inasmuch as there exists a set of persistent objects to which conflict chain elements may point. However, the export clause makes the symbol stack worth the effort. If a module exports symbols, these symbols are placed on the symbol stack before any others in that module. At the time of module exit, the scope frame pointer of the surrounding scope is

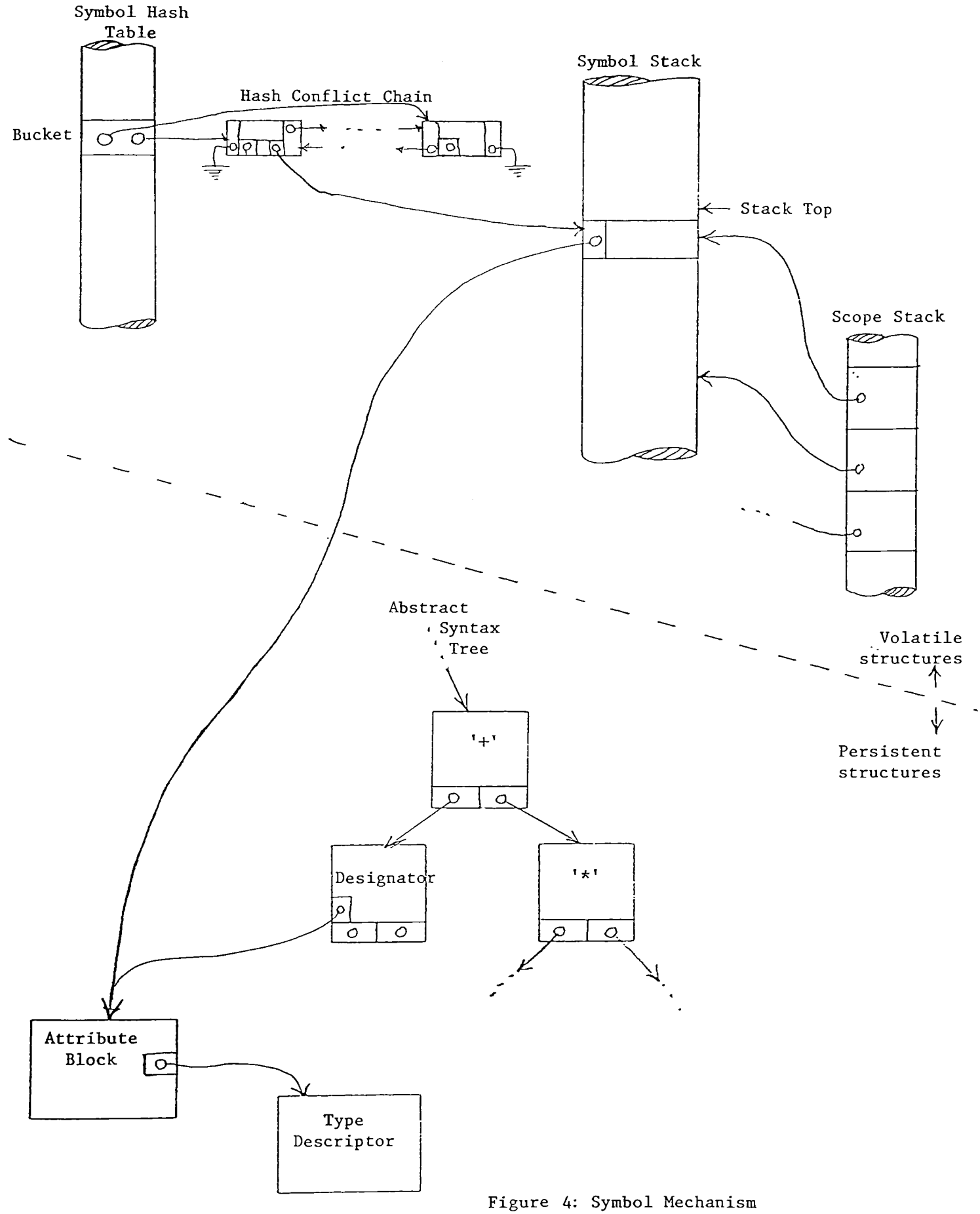

Figure 4: Symbol Mechanism

adjusted to include these exported symbols. Only one index is changed and no data need be copied.

When symbol uses occur, the conflict chain head is found by hashing the name as before and the conflict chain is searched for the first occurrence of that name. Searching of the conflict chain continues until a name is encountered which is below the visibility limit. The visibility limit is the nesting level of the enclosing closed scope (the module which contains the current environment).

Undefined forward references are more difficult to handle. The first use of an undefined symbol triggers the installation of that symbol as "UDEF". "UDEF" symbols are allocated space on the symbol stack and in the persistent symbol data base just as any other symbol. They are also copied into a list or "bag" of undefined symbols attached to the scope stack entry for the current scope. This bag of undefined symbols is propagated from open scope to enclosing scope until all of the symbols in it are defined. If a closed scope exit is encountered and symbols remain in the bag, a symbol definition error occurs. At open scope exit symbols in the bag must be copied into the bag of the enclosing scope and onto the symbol stack of the enclosing scope so that they may be searched normally. Care must be taken to ensure that the conflict chain entries are not destroyed for "UDEF" symbols at the time of open scope exit. We may want to add to the notion of an undefined variable the "provisionally" defined variable. This is required for forward references of procedures with a single parameter which may ultimately turn out to be a type transfer function and for variables which fall within the "scope" of undefined WITH control variables.

The problem of properly handling undefined forward references in WITH clauses is solved by an extension to a standard method for handling the

WITH clauses. The current system uses a WITH stack which pushes the symbol used in each WITH clause and pops same when the WITH statement is complete. This activity is syntax directed and takes place during the parse. When a name occurrence is searched, the WITH stack is always checked first. The symbol closest to the top of the stack which can resolve the new name is used for that resolution. If an undefined symbol is encountered in the WITH stack before any symbols which may resolve the identifier, then that symbol, the first symbol on the stack which can resolve the identifier and any intervening undefined WITH stack symbols are bound to the identifier. If no other *defined* control variables in the WITH stack can resolve the symbol, the first symbol in the hash table conflict chain (up to and including the visibility limit) which matches the undefined symbol is added to the candidate list. After a complete textual scan of a closed scope all names will have been defined (else compilation cannot continue). A link between the undefined control variables and all of the enclosed undefined variables must exist if unqualified symbols are to be resolved by the end of the first pass. The first element of the candidate list which can resolve the identifier is the correct parent of that identifier.

This mechanism should allow a nearly transparent implementation of one of Anderson's methods of removing the qualification of imported variables [ANDE 86]. In this modification of the Modula 2 language specification, it is possible to allow *local* removal of (module) symbol qualification through the use of the WITH statement.

As mentioned earlier, the installation of symbols is syntax directed and initiated in the parser, not the lexical analyzer as is more common in single pass lex/yacc based systems. Using the conventional technique, a set of flags is set and reset by actions at various stages of the parse. This state vector is

a representation of the state of the parse and may be used to attribute the symbol entry from the lexical analyzer. Clearly, the flags must reflect a complex state. Vector components of this state are "symbol phase" (whether the symbol is being *defined* or *used*) and "declaration class" (if the parser is in a symbol define state, is the declaration of a TYPE or a VAR or a literal). The reliance on these flags obfuscates the natural relationship between parser state and symbol state. For this reason the use of flags has been largely abandoned and only names of identifiers are passed to yacc from lex.

Incoming identifiers are installed in a hashed "name table" by the lexical analyzer. A pointer to the installed identifier string is passed to yacc through the standard yylval mechanism (The symbol is placed on top of the value stack) and yacc uses local context to create the attributes for the concomitant symbol. The use of a separate name table to store names may reduce space requirements in some cases but more importantly, the name table has utility the export-import system discussed later.

During the parse, the symbol table is accessed exclusively through the scoping system. During later phases of compilation, access to the symbol database (SDB) is by means of traversal of the intermediate representation of the program, the AST. Figure 5 illustrates the flow of data from input string to symbol table.

4. Intermediate Tree Construction -- The intermediate form of the input program is a binary tree. Members of a tree structure may be trees (interior nodes and their children), leaves (simple names in declarations or constants anywhere), or designators (use phase identifiers). Trees are constructed by grammar actions. The fundamental structure in a tree is the "block body".

The schematic representation of the block body in figure 6 is the required representation of every block. Because macros are used to access nodes within the structure tree, every node must be extant at the time the macros are used (else addressing errors occur). Other implementation options involve the use of procedural access mechanisms and, if interior nodes are allowed to be absent, tagged nodes (nodes *are* tagged in the current implementation). Even in the current implementation 7 of the nodes are essentially superfluous. The nodes labeled PARAMS, LOCALS, CONST, and TYPE were added to the tree too guard against the possibility that the symbol processing mechanism would not be adequate to handle the needs of persistent data. These blocks were essentially a crude tree based hierarchical symbol table. It now appears that they are not required or even useful and may be discarded. The TEMPORARIES node is still required for adjusting the runtime environment during code generation if the modified Sethi-Ullman algorithm runs out of registers. The MODULE node may be eliminated if the syntax driven module processing is used (see section on the first tree walk). The DCL module was used to get around yacc's difficulty with inherited attributes and did nothing more than link a modules name to the declaration block. The attribute structure abstracted from Fischer and LeBlanc obviated the need for this block but because so much of the code used the existing structure, it was allowed to remain. The interior CODE node is also not strictly required but was merely a (misguided) attempt to improve the readability of the tree.

A block body is created as soon as the first declaration in a block is seen or (if there exist no declarations in a block) when statement lists are bound to blocks. Interior nodes in the block body may serve the dual purpose of list head and structural node. "Leaves" in the block body are always used as

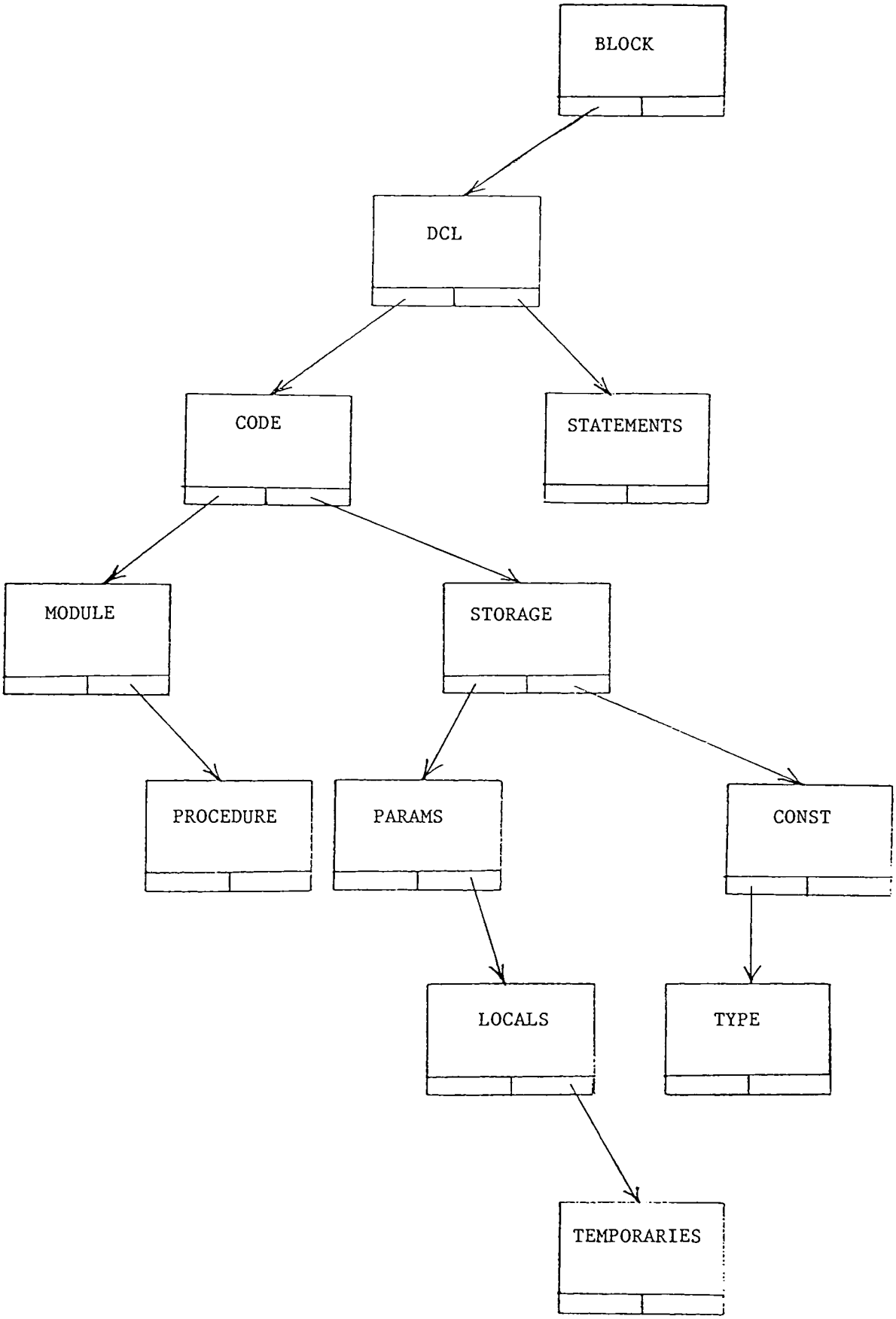

Figure 6: Block Structure Tree

list heads. All modules declared within a block are attached the module list head in the order in which they appear. Procedure, variable, type, and constant declarations as well as statement lists are handled in the same fashion. When a procedure (or module) is declared and added to the list of procedures (or modules), the entity which is added to thc list is a block body.

The entities in constant, type, and variable declaration lists are attribute blocks (treenode variants, the body of which is in the form of a LEAF).

Statements and statement sequences too are treenodes. If a treenode is not a leaf or designator, then it must have a "nodetype" (as opposed to the broader classification treetype). This node type corresponds in most cases to the token type associated with the construction of that node. In the case of statements, sample nodetypes are ASSIGN for assignment statements, IF for if statements, etc. Node types are defined by "TOKEN" assignments in the yacc grammar, ensuring that nodetypes have the same value as the corresponding value returned by lex. This enhances readability in the tree construction statements and guarantees that each node type will be unique. Those nodes which have no lexical equivalent are given a token assignment in the yacc grammar anyway. Examples of such nodes might be the set "or" operator which is lexically overloaded with addition. During the process of operator identification during the second walk of the tree, the nodetype for such a node might get changed from '+' to 'SETOR'. Because of this use of the TOKEN assignment, the file "y.tab.h" is included in compiler modules where it might not otherwise be expected.

The process of tree construction as it relates to various steps of the parse is, for the most part, straight forward as figure 7 illustrates:

```
simpexpr        :
                        factor {$$= $1;}
                |       simpexpr '+' simpexpr
                          {
                          $$=mknode('+',$1,$3);
                          }
                |       simpexpr '-' simpexpr
                                etc.
                                etc.
```

figure 7: a simple tree building example.

The yacc production for an "add" simple expression calls the mknode action which takes as its arguments the nodetype to be created, the left subtree, and the right subtree. In this case, an interior expression treenode with type '+' is returned and placed on the top of the value stack.

As mentioned earlier, a treenode may often be a list head or list element. One of the elementary productions used in variable declarations is the construction of identifier lists as seen in figure 8.

```
identlist       :
                        identlist ',' IDENT
                          {
                          link_pure($1,$2,AT_HEAD)
                          $$=$1;
                          }
                |       IDENT
                          {
                          $$=list_create();
                          link_pure($$,$1,AT_HEAD);
                          }
                ;
```

figure 8: Prototype list creation semantic action.

The details of the linked list system will be described in later section but it may be instructive to point out here that the set of grammar rules illustrated in figure 8 above is prototypically the way which lists are handled in the yacc grammar. Note that the definition of the list is left recursive and that there are 2 possible productions. One production is a list followed by a list element and the other is a list element by itself. The production which consists of a list element by itself is the action taken when the first element

of the list is encountered during the parse. At this time a list header is created and the a "pure" list element is added to the list. A pure list element is nothing more than a set of pointers which allow it to be a member of any list and a pointer to data. In this case the data to which the pure block points is the name placed on the value stack by the lexical analyzer. This simple form appears over and over in the yacc grammar and understanding this form is critical to understanding the parser. Schreiner provides a short discussion of lists and left recursion and a good tour through UNIX language tools in general [SCHR 85].

As the work on the parser was nearing completion a paper appeared which describes a language, constructed using yacc, which builds trees in yacc grammars. Park [PARK 88] calls his processor y+ and anyone seriously considering implementing a yacc based parser which emits trees would do well to investigate his work. A yacc based system for implementing attribute grammars in one pass [KATW 83] has been described. It might be possible to adapt this technique to aid in the attribution of tree nodes and thereby make the process more abstract and general. An inexpensive, commercially available software package (P-TREE) provides a very useful technique for managing the definition and use of syntax tree nodes. While this software is simple, clever, and makes good use of UNIX utilities, it is not in the public domain. Because this implementation is intended for public access, the P-TREE system was not used.

The Modula 2 syntax definition provides an opportunity for inexpensive constant folding in "constant expressions". A constant expression is an expression which consists of only literal constants or manifest constants. When used in the declaration section of a module, constant expressions may be evaluated at run time (no identifier used in a declaration section may be

undefined). In figure 9 below MaxOccupancy may be calculated by the compiler.

```
RoomsOnShip = 100;
BedsInRoom = 2;
PersonsPerBed = 2;
MaxOccupancy = RoomsOnShip * BedsInRoom *
                PersonsPerBed;
```

Figure 9: Constant expression example.

This produces a somewhat larger and slower compiler than would be expected if code were generated for manifest constants but the run time is smaller and faster. In any event, nearly all of the routines used in constant evaluation are also used by the operator identification phase of type checking of expressions (non-constant) in expressions.

5. **Initial Treewalk** -- In the section on system architecture it was mentioned that an initial walk of the tree was necessary to correctly handle modules which appeared in other blocks. The specification of Modula 2 is careful to do everything possible to force initialization of submodules. To this end, part of the specification states that the code for any module must be executed before the code of any enclosing environment. The Modula 2 definition also specifies that variables declared in a local module share the run time environment with the environment surrounding the local module. Figure 10 will help illustrate the implications of these specifications.

```
MODULE Main;
  FROM Foo IMPORT externvar;
  VAR
    x : INTEGER;

  MODULE Mod1;
    VAR
      x : CARDINAL;
    BEGIN
      x:=1;
    END Mod1;

  PROCEDURE Proc1;
    VAR
      x:CHAR;

    MODULE Nested1;
      VAR
        x:INTEGER;
      BEGIN
        x:=-1;
      END Nested1;

  (* body of Proc1 *)
    BEGIN
      x:=0C;
    END Proc1;

(* body of mainline program *)
  BEGIN
    x:=0;
    Proc1;
  END Main;
```

Figure 10: A Nested scope example

In figure 10 above, module Mod1 and procedure Proc1 are at the same scope level within module Main. Module Nested1 is one scope level deeper (within Proc1). The two rules discussed above dictate that the first statement to be executed will be the Mod1 statement "x:=1"; The second statement will be "x:=0" in Main. The two instantiations of "x" initialized so far are separate entities, yet share the same runtime environment. The "x" of Mod1 is at *scope level* 2 while the "x" of Main is at *scope level* 1. Both these instances of "x" are at *nesting level* 1. This clearly illustrates that module scope is a strictly compile time (in particular, first pass) constraint on the view of the

symbol database while nesting level is an attribute describing the run time environment. Scope level is incremented on open and closed scope entry but nesting level is incremented only on open scope entry.

Because module Nested1 is within Proc1 its code will not be executed until Proc1 is called. At that time the statement list of Nested1 will be executed before the statement list of Proc1 and that the variable "x" in Nested1 is allocated on the stack of Proc1.

The central issue here is that a module has no separate run time environment. It is merely a scoping convention which affects execution order and storage views. This applies to all modules including the compilation unit itself if one assumes an implicit environment which surrounds all compilation units.

It is incumbent upon the compiler to ensure the specified order of module execution. Given the block body tree structure described above (with lists of modules attached as subtrees below the enclosing module), there are two obvious ways of meeting this requirement for modules local to the compilation unit. One method is to move the statement list of the submodule up to the head of the enclosing level statement list during a second pass walk of the tree, thus ensuring that the enclosed list will be executed first. The alternative method is to treat modules as special parameterless subroutines and to insert a "call" to the module at the head of the enclosing statement list. Both methods require copying the variable declarations up to the surrounding block declaration list and recalculating the total offset for the surrounding block and individual offsets for moved variables. By prepending either a statement list or a "module call", proper execution order is guaranteed by a preorder traversal of the block tree.

A method not given much consideration early in the development was a syntax driven gathering of statements into the proper block body during the parse. Initially, this method was discarded for three reasons: 1) the enclosing level would be an input variable (inherited attribute), 2) the module "call" technique was required for non-local modules, and 3) the syntactic and semantic handling for both modules and procedures would be nearly identical during the parse if a separate treewalk were employed. Inherited attributes are a problem in yacc for obvious reasons. The arbitrary position of enclosing nesting level block bodies on the value stack make the design of copy rules seem a formidable task.

One could use a global pointer which contained the address of the enclosing block body. A single pointer is insufficient, however, after an open scope boundary has been crossed. Because an arbitrary number of open scope boundaries may be crossed, a stack or list of enclosing block bodies is required. Through this method statement lists would be added to the head of the statement list of the block pointed to by the element on the top of the nesting stack. Individual statements could NOT be added at the head of block statement lists, however, as this would invert the order of the statements! Statements must be collected into lists on a "per scope" basis then the correctly ordered list must be added to the head of the block statement list. New block bodies are created at open scope entry and a pointer to the block is placed on top of the nesting stack. At open scope exit the block pointer on the nesting stack is popped.

The tradeoffs between a separate treewalk and parse time collection center around the cost of the treewalk versus the complexity of parsing modules in a fundamentally different way than procedures. The cost of a tree walk which copied the variable list up to the enclosing open scope and inserted a

"module call" seemed to be the least expensive and most easily understood. Figure 11 illustrates the skeleton of the code which handles the submodule problem.

```
walksubmodule(encloser, sublist)
char * encloser;
listhead_t * sublist;

{
  char * block;

  GETHEAD(block,sublist);
  while ( NULL != block)
    {
    copyvars(encloser, block);
    insert_mod_call(encloser, block);
    append_mod_return(block);
    walksubmodule(encloser, SUBMODULE(block) );
    walksubproc(SUBPROC(block));
    GET_NEXT(block);
    }/* end WHILE over sub block list */
}
```

Figure 11: a postorder traversal of the block tree

Using the submodule technique one might observe that, once the variable addressing considerations were appropriately handled, the order of the statement lists would not need to be explicitly changed. By choosing an appropriate order of traversing submodules on the code generation pass (depth first) one would automatically produce correct (in terms of execution order) code! This would not, of course, solve the problem of bringing variable declarations for submodules into the correct environment (block subtree).

Although the current implementation uses a separate treewalk for gathering statement and variable lists, it has become clear that the parse time approach is more appropriate to the meaning of the module construct. The list processing facilities make it a simple matter to collect statementlists

and to prepend the collected list to the head of the enclosing open scope at parse time.

One final problem with the parse time approach is the handling of priorities. The module "priority" is used (in the MC68000) to set the 3 bit section of the status word which serves as the interrupt mask. Because the module body of local modules may have a different priority than the surrounding environment, some mechanism would be required to set and reset this priority upon entry and exit of the module body. Because the inline technique has no convenient "jsr" entry point, explicit inline code must be generated to push the old status word and set the mask in the current status word on entry of the module body. On exit from the module body the old status word must restored from the stack. The syntax directed action on encountering a priority in a local module would be to push a treenode in the statement list which would trigger the appropriate code generation on the third pass treewalk.

Procedure parameters provide an even more exciting problem for the compiler writer. Because procedures take on the priority of the enclosing module if such a priority exists, priority is an attribute of the procedure. Because procedure parameters may be assigned values at runtime, and because the language specification stipulates that a procedure may NOT call another procedure of higher priority, checking of proper procedure call protocol *must* be a runtime check! This is at odds with the underlying philosophy of Modula 2 and is not supported by the current implementation. Figure 12 below illustrates how the difficulty may arise.

```
MODULE ProgMod;
  MODULE Low [5]; (* priority of Low is 5 *)
    EXPORT LowProc;

    PROCEDURE LowProc;
      BEGIN
        (* do some stuff.... *)
      END LowProc;

    BEGIN (* body of Low *)
    (* do some stuff *)
    END Low;


  MODULE High[7];  (* priority of High is 7 *)
    IMPORT LowProc, SomeExpProc;

    VAR
      ProcVar : PROCEDURE;
      SomeCondition : BOOLEAN;

    PROCEDURE HighProc;
      BEGIN
      (* do some stuff *)
      END HighProc;

    BEGIN (* body of High *)
      SomeCondition := SomeExpProc;
      IF SomeCondition THEN
         ProcVar := LowProc;
      ELSE
         ProcVar := HighProc;
      ProcVar;
    END High; (* body of High *)

  BEGIN (* ProgMod *0
    (* do some stuff *)
  END ProgMod.
```

Figure 12: Illustration of need for run time check of priority

6. **Semantic Checking and Tree Rewriting** -- At this point all identifiers have been resolved, all variables are in the proper context, and statements are properly ordered. The AST is complete and "correct" but 3 major issues remain before it is reasonable to commence with the preparation for code

generation. These major tasks of the current phase are briefly described below with more complete descriptions of the problems to follow.

1. **Type Checking** -- Although the AST is *syntactically* correct it is not necessarily in complete harmony with the Modula 2 language specification. For instance, it is perfectly correct syntactically to index an array using an identifier which is the name of a simple type. This is not, of course, correct semantically. The compiler must therefore verify usage of names in statements.
2. **Translation of Complex Names to Address Arithmetic** -- The fragment of the yacc grammar given in figure 1 produces a list of "nodes" which is a preliminary form for designators. This representation is unwieldy from the standpoint of code generation. Because code generation in the current implementation uses a tree based algorithm, it would be convenient to have no need to process designator "lists". In this phase, these lists are converted to address arithmetic trees.
3. **Short Circuit Transformation** -- Boolean expressions as used in WHILE, IF, and CASE statements are currently simple expression trees which produce a TRUE or FALSE value after evaluating the entire tree at runtime. This method would be perfectly acceptable if the Modula 2 AND and OR operations were the same as their namesakes in Pascal. Modula 2 supports (requires) short circuit evaluation, making the AND operator into the AND THEN operator. The OR becomes OR ELSE. For this reason, BOOLEAN expression trees must be transformed into a form called *jump codes* [FISC 88].

This list is expanded below. The tasks are accomplished by an order indifferent walk of the tree, looking only at statements. While the walk of

the structure tree is not sensitive to order, the walks of subtrees (notably expression trees) is highly dependent on order. This alteration between preorder, inorder, and postorder traversals of the various subtrees may, in a certain sense, be thought of as a real manifestation of the partitioning of an attribute grammar. Waite and Goos [WAIT 83] provide an excellent discussion of attribute grammars and, as an added bonus, use SIMULA for many of the coding examples!

The detailed discussion of the tasks of this phase follows.

Type Checking -- The fundamental program unit upon which type checking operates is the statement. For instance, if a statement is a procedure call, checking of this subtree is handled in the fashion appropriate to subroutine calls. Parameters are checked to make sure they correspond to their associated formal parameters in type and "writability". If a formal parameter is passed as VAR, then no readonly actual parameters may be passed in this position (eg. function calls, opaque vars, or constants). Expressions passed as parameters must be checked as described in following sections, both for internal type consistency as well as consistency with the declared type of the formal parameter.

The other statement forms (see figure 13 below) are processed individually, with a 'C' case statement for each type.

```
statement      :
                proccall
               | assignment
               | ifstatement
               | casestatement
               | whilestatement
               | repeatstatement
               | forstatement
               | loopstatement
               | withstatement
               | EXITTOK
               | RETURNTOK
               | RETURNTOK expression
                ;
```

Figure 13: Yacc rule for the Modula 2 statement

Type checking constitutes the bulk of the activity in the procedures which process the statement trees. Types are checked when [CHRI 86]:

* variables and constants are used in expressions,
* value is assigned to a variable,
* parameters are passed to procedures,
* values are returned by procedures, and
* array indices are used.

In addition to the instances cited above, CASE control expressions and labels must be of compatible type and the FOR statement index variable must have base type INTEGER or CARDINAL.

Of the type checking tasks, validating the type of a designator is the most difficult and complex task. The designator in figure 14 is syntactically valid.

```
modulename.rec.fld1^[32*a[b.f1[6,i]^].aptr^^,fcall(x,y)+1]
```

Figure 14: A complex designator instance.

Here we have a record which has been exported from modulename. The first field of the record is a pointer to a two dimensional array. The array

indices are expressions; one a constant multiplied by a designator; the other a function call added to a constant. Type checking procedures must allow the possibility that the first name of a designator may be a module name if and only if the next "token" in the designator is a "." and must disallow a module name in all other circumstances(?). The right number of expressions must occur in array addressing expression lists and must be of the right type. The "right" number of expression lists, by the way, depends on the context in which an array element is used. In an assignment statement any number of subscripts (expressions) are allowed up to and including the total number of dimensions of the array. In all other circumstances, the number of subscripts must equal the total number of dimensions of the array. When the pointer dereference operator is used, we must verify that the preceding name represents a pointer variable. Names and types of record fields must be checked for proper usage.

Translation to Address Arithmetic -- A designator is represented here as a terminal treenode of a special kind. It is a listhead which contains a pointer to the attribute record which resolves the first identifier in the designator list. A node in the designator list may be a DESIGNODE, a REFNODE, or an ARRAY_LMT. A DESIGNODE points to the next element in the list and to a name string unless it is the first element is the designator list. If it is the first element, it points to an attribute block of the name which resolved the identifier during the parse and not to a name string. Subsequent nodes in the designator list are resolved (checked against record fields or module export lists) during the pass currently under consideration. ARRAY_LMT nodes point to the next element and to expression lists. REFNODES only point to the next element in the list. For the purposes of parsing, name resolution and type checking, this form for the designator is

entirely adequate. However, as we will see later, it detracts from the generality of the address generation aspects of code generation and might well be rebuilt during this pass as the individual "semantics" of each designator are understood. In particular, it would worth the investment in time and effort to convert designators into address arithmetic trees at this time.

The effort to build designator trees will be rewarded when register counting and code generation occur. When designing the designator tree structure and supporting code, one must keep in mind the location of data and code which must be addressed. For Modula 2 the following is one possible list of such places on the MC68000:

1. Module data and code regions (accessed as displacements off the module base address)
2. In the current activation record (accessed as displacements off the current data region offset)
3. In a previous activation record (accessed in a similar fashion as 2 above after chaining through the correct number of static links)

The designator tree will be useful only if it facilitates address generation in the subsequent two phases of compilation. A more detailed discussion of the tradeoffs in the design of the designator tree follows in the section on preparation for code generation.

Transformation of BOOLEAN Expressions to Jump Code -- Examine the IF statement in figure 15.

```
IF NIL # SomePointer AND SomePointer^.Val < 0 DO
```

figure 15: Example of need for short circuit evaluation.

If "SomePointer" *is* NIL it is inadvisable to continue evaluation of the IF statement. In most languages, the outcome of the above sequence of

operations is at best indeterminate and could introduce subtle, difficult to diagnose bugs into a system. Because this is a natural way to think about the problem and because it is widely practiced in those environments which support it, Wirth has made short circuit evaluation a specified feature of the Modula 2 language.

For the user of yacc implementing a correct version of Modula 2, short circuit evaluation provides something of a problem. Examine figure 16 below.

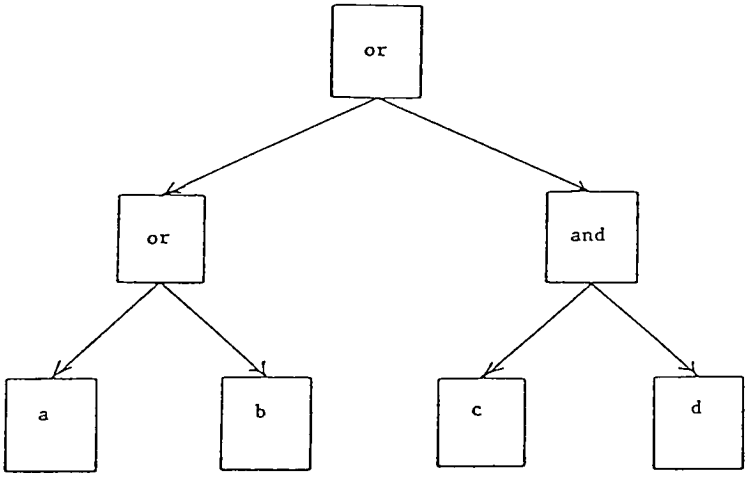

figure 16: tree for (A OR B) OR (C AND D).

If the expression above were to appear in an IF statement, it would be correct to jump directly to the fall through (TRUE) statement list if it determined that "A" were true. The information required to generate the appropriate jumps flows through the syntax tree from top to bottom. These inherited attributes are difficult to propagate in bottom up parsers such as yacc. While it may be possible to write the yacc rules so that the jump address could be set at parse time, it is unlikely that this approach would yield as sparse a syntax tree as a separate traversal combined with a tree rewrite. This is because a parse generated jump code could yield at best a jump span of "n" nodes in the tree where n is determined by the length of a given production (or group of productions if one is facile at producing copy

rules). Note that this observation is merely that and no formal proof of the statement has been attempted for LALR parsers. No doubt some curious undergraduate will examine the issue and prove the above assertion false. (The prophesy has been fulfilled [ATTE 89]! It now seems obious that the *right* way to do this is to label the tree during the parse using a right recursive rule [hoping yacc will not blow up] and generating code directly from the labeled tree.)

A separate inorder treewalk will provide jumps which span an arbitrary number of nodes by propagating the jumps of OR statements from the top of the tree down. Note that, using this technique, AND and OR evaluations disappear completely from the code and evaluations of individual objects followed by jumps constitute the realization of the expression.

Suppose the expression in figure 16 were to appear in an IF statement and that the label attached to the statement list to execute if the outcome were TRUE is denoted by $L100 and the label for the FALSE outcome statement list is $L101. Then figure 17 represents in (optimized) pseudocode what might be generated from the transformed expression tree.

```
mov   A, D0
bne   $L100
mov   B, D0
bne   $L100
mov   C, D0,
be    $L101
mov   D, D0
be    $L101
bne   $L100        ;just in case tree order does not
                   ; permit fall through
```

figure 17: code generated from figure 16.

Note that no "anding" or "oring" appears in the code. NOT operators may be implemented by appropriate swapping of jump addresses in OR statements and changing the test generated in AND statements.

In closing this section it might be pointed out that, given the current representation of the AST, it is not at all clear that a separate tree walk is necessary to accomplish short circuit evaluation. The decision to prepare BOOLEAN expression trees with jump labels in a separate pass rather than generate jump codes directly is based partially on esthetic considerations (directly generated code is ugly) and partially on efficiency considerations. See figure 18 for an example of directly generated jump code.

```
                br      L100        ;jump to beginning
L103:
                mov     (d),d0      ;see if d TRUE
                brne    Lfalse
                br      Ltrue       ;last test
L102:
                mov     (d),d0      ;see if c TRUE
                be      Lfalse
                br      L103        ;go test d TRUE
L101:
                mov     (b),d0      ;see if b TRUE
                bre     L102
                br      Ltrue       ;TRUE=>expr TRUE
L100:
                mov     (a),d0      ;check a TRUE
                be      L101
                br      Ltrue       ;TRUE=>expr TRUE
Ltrue:
                .       ;some assembly code sequence...
                .
                .
Lfalse:
```

figure 18: direct generation of jump code from expression of figure 16.

By employing a separate labeling phase, code generation may engage in a preorder traversal of the BOOLEAN expression subtree. This has the great advantage of laying down assembler in the order in which it is to be executed. There is a high probability that an optimization phase or even an efficient generation algorithm may get rid of half of the jumps generated (by allowing a fall through). One can see from figure 18 that the direct generation method would require gross rearrangement of the generated code.

BOOLEAN expressions realized as jump codes require that information (jump targets) flow from right to left in a tree and that control flow from right to left at run time. This requires an unusual traversal of the expression tree which is referred to here as "mirrored" postorder:

1) visit the right subtree
2) visit the left subtree
3) visit the root

The term "mirrored postorder" was coined because no reference to a similar traversal could be found in the traditional literature; not even Knuth seems to acknowledge its usefulness. This unusual traversal is used to propagate the jump targets to the left side of the tree from the right and is responsible for the unpleasant jumping found in the directly generated code. Figure 19 below illustrates the flow of jump targets in the expression tree of figure 16.

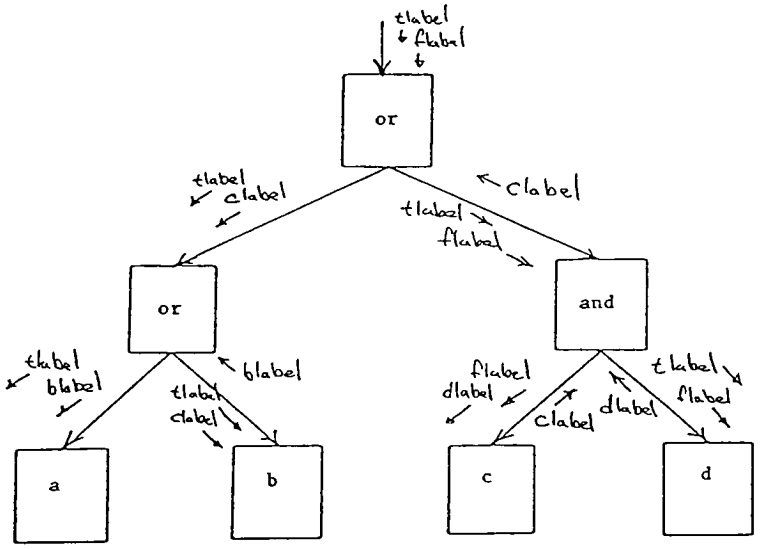

figure 19: data flow in expression tree structure.

Note that the flow of data in this figure, while similar to a Yourdonesque structure chart is quite distinct from that form. Figure 19 is merely a replication of the figure 16 expression tree attributed with jump target flow.

The ordering of the legs of the tree is the exact opposite of the ordering of a Yourdon structure chart in terms of execution order. Here the right leg of the tree is examined first rather than last. Seen in this light one wonders if there might not be a syntax directed labeling of the tree from LALR style parsers. Figure 20 is a 'C-like' pseudocoding of the labeling algorithm.

```
// return a label string which is "the jump target"
char *
GenJumpLab ( tree, fjump, tjump )
treenode * tree;              // expression tree
char * fjump;                 // false target of parent
char * tjump;                 // true target of parent
{
  char * ReturnJump, * LeftJump;
  switch ( tree->treetype) {
    case (TREE)
      if ( logop ( tree ) )
        {
        switch ( NODE ( tree ).nodetype ) {
          case ANDTOK:
            LeftJump = GenJumpLab ( tree->rightleg,
                    fjump, tjump );
            return ( GenJumpLab ( tree->leftleg, fjump,
                    LeftJump ) );

          case ORTOK:
            LeftJump = GenJumpLab ( tree->rightleg,
                    fjump, tjump );
            return ( GenJumpLab ( tree->leftleg,
                    LeftJump, tjump ) );

          case NOTTOK:
            // note swap of sense
            return ( GenJumpLab ( tree->leftleg,
                    tjump, fjump ) );

         default:
           {
           // Horrible Malformed Node ERROR
           } /* end DEFAULT case on nodetype */
        } // end of SWITCH over nodetype
       } // end of IF logical operation
      else
       return ( label_tree ( tree ) );

    case DESIGNATE:
      setjumplabs( fjump, tjump );
      return ( label_tree ( tree ) );
```

```
    case LEAF:
      setjumplabs( fjump, tjump );
      return ( label_tree ( tree ) );

    default:
      {
      // Horrible Malformed Tree ERROR
      }
    } // end SWITCH over treetype
} // end of char * FUNCTION GenJumpLab
```

figure 20: a labeling algorithm for jump codes

7. **Preparation for Code Generation** -- The generation of code from expression trees depends on a technique known as Sethi-Ullman Numbering [FISC 88], [AHO 86], [GISS 86] wherein a depth first traversal of the expression tree labels each node in the tree with the minimum number of register required to evaluate the subexpression represented by the node. The implementation of this algorithm is affected by several factors among which the most important are:

1. uniformity of the register operations in the target ISA
2. data access methods supported by the source language (eg records, pointers, and arrays)
3. support of procedure nesting in the source language
4. support for separate compilation in the source language

The Motorola MC68XXX architecture is the target ISA of this implementation. If one were to rate CISC processor architectures based on the amenability of the ISA to Sethi-Ullman Numbering, the MC68XXX would fall somewhere between the Intel 80X86 series (difficult to implement) and the DEC pdp/VAX series (easy to implement).

The DEC style architecture which includes the National 32X32 series, the AT&T 32X00 series, and the Zilog z80000 series provides a highly uniform

register file and powerful instructions which operate uniformly on all data be it register or memory. Loading an address and performing arithmetic on that address (as in array subscripting) is no different than performing arithmetic on any other data. In the most general case, all registers work with all instructions and all instructions work with all addressing modes. This greatly simplifies the process of allocating registers for address generation and expression evaluation. With a truly general architecture the code generation algorithm for expressions and address generation is simple, elegant and eminently easy to understand. Wirth, in a paper on the relative efficiency of language implementation for various architectures addresses this and other issues for the above architectural styles [WIRT 86].

The difference in the above processors is, however, one of degree. All of the current crop of CISC architectures will support, albeit with some difficulty, the code generation algorithm used here. Other possible machines, notably the IBM 370 series, provide great difficulties for the designer wishing to use Sethi-Ullman numbering; although ,even here, a solution is possible [AHO 86].

The code generation algorithm used here depends upon the following general instruction format:

```
BINOP        register, (memoryloc)
```

In this case the source operands are a register and the contents of an address in memory. The binary operation leaves the result of the operation in the operand on the left; in our case a register.

If operations directly on memory are not supported by the ISA (as in RISC), the algorithms for code generation and register counting are more complicated. The emitted assembler:

```
BINOP       Regx, (memloc)
```

for a CISC machine becomes:

```
LOAD        Regy, memloc
ADD         Regx, Regy
```

on a RISC machine. It seems that RISC machines require an extra volatile register when evaluating expression trees. Perhaps designating one register to this task and removing it permanently from the list of available registers (in the same way the frame pointer is removed) would be sufficient. Because RISC machines typically have register files of over 100 registers, the dedication of a single member of this set as a work area might be the most efficient solution to the problem.

Before discussing the implications of the chosen architecture, it may be profitable to present a simplified subset of the tree labeling algorithm. For a pdp-11 style architecture and a language supporting no complex designators or unary operators, the algorithm "registerneeds" in figure 21 describes the labeling technique.

```
registerneeds ( tree )
treenode_t * tree;
{
  if ( tree->treetype != TREE )
    {
    if ( tree->branch == RIGHT )
      tree->RegCount = 0;
    else
      tree->RegCount = 1;
    }
  else // we have an expression subtree
    { // depth first!
    registerneeds ( tree->leftleg );
    registerneeds ( tree->rightleg );
    if ( tree->rightleg->RegCount ==
                tree->leftleg->RegCount )
      tree->RegCount = tree->rightleg->RegCount + 1;
    else
```

```
        tree->RegCount =
                MAX ( tree->rightleg->RegCount,
                tree->leftleg->RegCount );
    } /* end ELSE expression subtree */
} /* end void FUNCTION registerneeds */
```

figure 21: 'C' language algorithm for labeling treenodes

If the left leg of a tree is a leaf (terminal identifier) and the right leg is a TREE, we can see from the algorithm above that an extra register is used to evaluate the parent tree than if the right and left leg are swapped or, more properly, commuted. The current implementation checks binary operations for this condition and, if it pertains, the binary operator is examined for the property of commutability. If the operator proves to be commutable the subtrees are interchanged to decrease the demands on the register file. The reason that one less register is used if the LEAF is in the right subtree is that the register used to load the left subtree will be the ultimate destination of the calculation. If a LEAF is on the right side, a register-memory operation will obviate the need for a register in that subtree [AHO 86].

The bipartate register set of the MC68xxx combined with the separate compilation and nested scope features of the Modula 2 language (as well as support for pointers) greatly complicate the registerneeds algorithm. The introduction of base-offset addressing and uplevel references require that the MC68xxx address registers be loaded and used for address arithmetic in address generation for complex designators, data in previous activation records, and data and procedures in separately compiled modules. These address registers, like the data registers implicitly used in figure 21 are a resource which requires compile time management. Therefore the single field of tree node, "RegCount" is insufficient and we need to reflect the needs of the language and architecture in the tree node structure with the fields "DataRegCount" and "AddrRegCount".

The initial version of this compiler did not admit the use of complex designators, uplevel references, or separate compilation. For this simple subset, the tree labeling routine above (registerneeds) is adequate because one need only deal with the data registers set. But now consider the following statement.

```
c := a.f1[32,c+d]^.f2;
```

Assume "c" is an integer in a previous activation frame and "a" is the name of a module. Further assume "f1" to be a 2 dimensional array of pointers to a record type variable may be an INTEGER or CARDINAL. The simple assignment tree in figure 22 below is constructed by the parser.

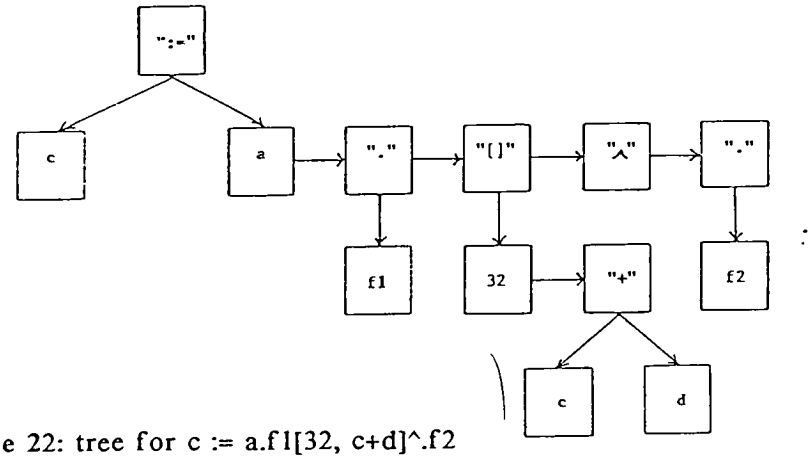

figure 22: tree for c := a.f1[32, c+d]^.f2

To enable the compiler to fully support the addressing required by Modula 2, the tree in figure 23 (or something roughly akin to it) must be generated for subsequent tree registerneeds labeling and code generation. The managing of the two separate genera of registers on the MC68xxx make the determination of register needs for any given tree node a non-trivial task. Note that the nodes labeled "UPLEVEL" and LOAD BASE trigger the loading of a new address register. We can see that the address register requirement of this assignment statement is 3. One register is used to get the l-val of "c" (l-vals are always addresses), another is used to get the address of the array base

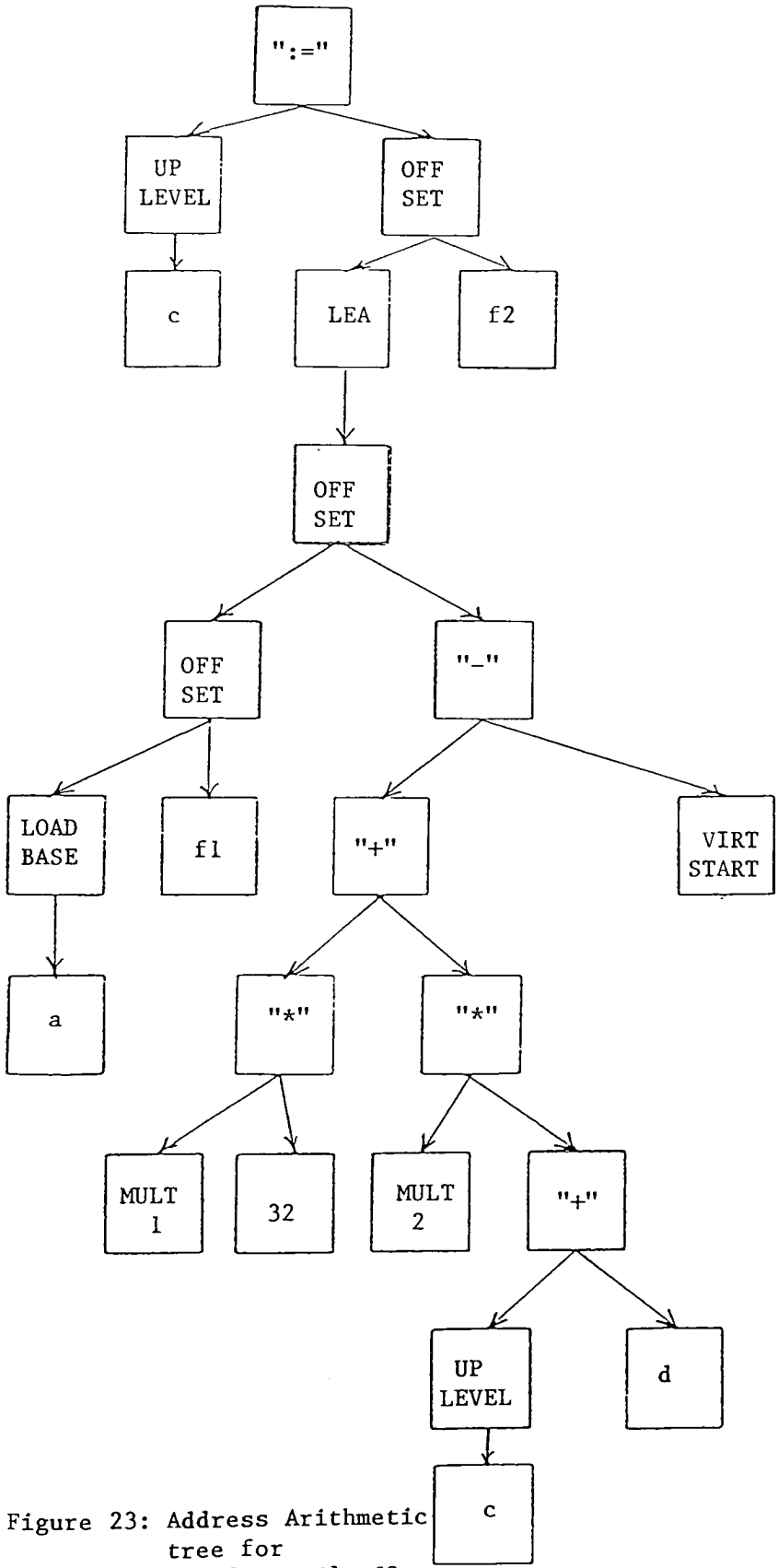

Figure 23: Address Arithmetic tree for a.f1[32,c+d] .f2

and the last is used to calculate the value of "c" used in an expression to calculate the 2^{nd} subscript of the array element.

The diagram in figure 23 is offered more as an aid to making the intuitive connection between the statement and the tree than as a representation of a form from which code might be generated. In figure 23 the l-val for "c" and the value of "c" are represented by the same form. Some sort of "value" [ANKL 82] or "indirect" [AHO 86] operator must be used to load the contents of the variable "c".

The OFFSET node is merely the Motorola MC68xxx "address add" operator in tree form and is required because of the lack of generality in the MC68xxx register descriptions. In VAX style architectures this would simplify to the ordinary 2 address add operator.

In the tree from which code is finally generated, the calculation of "d" may use the OFFSET operator as well. If "d" is in the current activation frame, the value associated with "d" may be had by adding the known (at compile time) offset of from the frame base to the frame pointer. Thus the OFFSET operator might be used to get the address of the contents of "d" in the expression tree. The code generation modules, however, might chose to dispense with the "address add" operation and indicate the contents of "d" by:

```
mov   offset_d(fp),a?
```

rather than

```
add   (offset), fp
mov   (fp), a?
```

Exactly where to place the intelligence which decomposes the name into addressing is a matter of taste. The only hard and fast requirement is that sufficient information be present during the labeling phase to allow "registerneeds" to produce a correct result. Although the tree in figure 23 is

generated from the structure in figure 22 during the previous pass, it was thought inappropriate to discuss this issue before the registerneeds algorithm had been presented.

Addressing array elements takes advantage of the information available at compile time to reduce the runtime overhead. For an "n" dimensional array, the $MULT_k$ of figure 23 for element "k" is calculated by:

$$MULT_k=(upperbnd_{k+1} \quad lowerbnd_{k+1} + 1) * MULT_{k+1}$$

where $MULT_n$ = element size (of the base type of the array). These multipliers generate an expression for calculating the offset from the array base as follows:

$$MULT_k*[index_k - lowerbnd_k]$$

When rewritten as:

$$[\quad MULTk*index_k] - VIRT\_START$$

where

$$VIRT\_START = \quad MULT_k * lowerbnd_k$$

one can easily see the relationship to the lower right region of the addressing tree in figure 23 [ANKL 82].

Open array parameter definitions would appear to complicate this scheme somewhat because the bounds information is unknown at compile time. But because only singly dimensioned arrays are permitted as open array parameters, the equations above collapse acceptably. When "n"=1 the summation and calculation of $MULT_{k>1}$ disappear.

Arrays as procedure parameters require passing a block of addressing information to the procedure. If done at run time this block is historically known as the "dope vector". In the current implementation a dope vector

should not be necessary because all of the information required for addressing should be known at compile time. For open array parameters, VIRT_START is equal to 0 and $MULT_k$ is equal to element size.

8. Code Generation -- The code generation phase of this compiler does not, as is often the practice, produce an intermediate linear form from the tree. Powell [POWE 84a] generates something similar to the original Pascal p-4 code [PEMB 82]. A separate static interpretation process transforms the p-code file into target ISA assembly code. While the use of P-code has been criticized as being not amenable to optimization, there can be little doubt that a compiler which generates P-code would be highly portable. It was originally intended that P-code be used in the current implementation but the time needed to realize this phase was thought to be prohibitive. Perhaps better than P-code is M-code formulated by Wirth for the Lilth machine. This form should map the Modula 2 language more efficiently than P-code but may depend specifically on the Lilith architecture [WIRT 86]. Little information seems to be available on the M-code instruction set. Other intermediate forms are available. Frailey [FRAI 79] offers a tuple based linear form which does support a range of optimizations. A rich literature exists for intermediate languages [CHOW 83], [LUTZ 77], [OTTE 84].

Code generation here consists of two principle modules, code generation and code emission. This dichotomy was used in the hope that, by isolating the emission of code, processor independence would be easier. In particular, it was hoped that the UNIX processor independent instruction set might be used to emit code which was capable of being assembled and run on any reasonable UNIX machine. Code generation embodies the intelligence needed to understand trees and addressing while code emission takes the

address and operator information gathered by generation and formats it into assembly code in an output file.

Processor independence has not come to pass but the achievement of this goal is not far off. A motivated individual should be able to take the output of the current compiler (MC68xxx UNIX assembler) and transform it into code for the AT&T 32200 series machine or even VAX or National 32x32 code. By mapping the special purpose registers of the machines (sp, fp, pc, etc.) from the MC68xxx onto the target machine registers and choosing any one-to-one mapping of the remaining registers, the translation should be simple. Any macro processor such as M4 should be sufficient to the task.

For RISC machines with large register files (greater than 16) the transformation should be only slightly more complex, provided that register scoreboarding is available in hardware and not in compile time software. For other architectures such as the Intel 80x86 series, it is possible that only the register list initialization code and emission code be modified.

The generation routines -- Those parts of the compiler which understand the relationship between trees and sequences of operations on address locations are the generation routines. Trees come in two distinct flavors, structure trees and free form trees. Structure trees are best illustrated by the block structure tree of figure 6. These trees have a compiler defined structure which is invariant over instances of the form. This definition breaks down somewhat for IF-trees because ELSEIF-trees may be appended *ad infinitum.*

Free form trees are those trees about which we may have no expectations as to form. Examples of free form trees are arithmetic and boolean expression trees as in figure 22.

. The most meaningful discriminator which may be applied to differentiate between free form and structure trees is the manner in which they are handled by the generation routines. Each structure tree is processed by a unique and dedicated set of routines. The block tree is traversed by a set of postorder treewalk routines which manage such things as depth of nesting tracking (for address generation) register tracking (for save and restore runtime operations), etc. The basic form of these traversal routines is similar to those of figure 11. The WHILE tree is processed by the WHILE routines which "know" about the tree structure of the WHILE loop. Most statement types are associated with a structure tree and each of these structure trees is processed by a unique set of routines. The processing associated with structure trees is straight forward and, for the most part, uninteresting.

Free form trees, on the other hand, are an entirely different matter. The section of this document relating to preparation for code generation presented a technique called Sethi-Ullman numbering. This labeling of every expression (or free form) tree node with the register requirement at the node allows a recursive function "treecode" to use a list [FISC 88] or stack [AHO 86] representation of the currently available registers to generate code for arbitrary expressions. This function is shown in simplified form in figure 24.

```
int
treecode ( t, datareglist )
treenode * t;
listhead_t * datareglist;

/* ======================================== */
{
  treenode    * left_tree, * right_tree, * temp_tree;
  char        * dr2;
  treenode    * r1, * r2;
  listhead_t    * Remaining_dataregs,* dregs_dataregs;


  r1 = (treenode *)pure_head ( datareglist );
  Remaining_dataregs = tail ( datareglist );

  switch ( t->treetype ) {

    case LEAF:
      generate ( LOAD, t, r1 );
      break;

    case TREE:
      right_tree = t->rightleg;
      left_tree = t->leftleg;
      if ( 0 == right_tree->DataRegCount )
        {
        /* >> RIGHT TREE MUST BE CONSTANT
             ( OR DESIG ) << */

        treecode ( left_tree, datareglist );
        generate ( NODE ( t ).nodetype, right_tree, r1 );
        }
  /*
   --     BELOW THIS POINT, BOTH LEGS OF THIS
TREE MUST BE TREES AND INVOKE
     RECURSIVE CALLS TO "treecode"
  */
      else if ( ( left_tree->DataRegCount >=
              listlength ( datareglist ) ) &&
          ( right_tree->DataRegCount >=
              listlength ( datareglist ) ) )
        {
        /* -- MUST SPILL A REGISTER -- */
        treecode ( right_tree, datareglist );
        get_storage ( temp_tree );
        generate ( STORE, temp_tree, r1 );
        treecode ( left_tree, datareglist );
        generate ( NODE ( t ).nodetype, temp_tree, r1 );
        }
      else /* -- ONE OR BOTH SUBTREES DO NOT
              NEED ALL OF THE REGISTERS -- */
        {
        Remaining_dataregs = tail ( datareglist );
```

```
        r2 =
          (treenode *)pure_head(Remaining_dataregs );
        if ( left_tree->DataRegCount >=
            right_tree->DataRegCount )
          {
          treecode ( left_tree, datareglist );
          treecode ( right_tree, Remaining_dataregs );
          generate ( NODE ( t ).nodetype, r2, r1 );
          } /* end IF need more or = regs on left than
              on right */
        else
          {
          dregs_dataregs = tail ( Remaining_dataregs );
          /* >> BUILD LIST WITH ORDER OF R2 AND
               R1 SWAPPED FROM ORIGINAL << */

          dr2 = link_pure (
                dregs_dataregs,r2,AT_HEAD );
          treecode ( right_tree, dregs_dataregs );

          dr2 = (char *)unlink_pure (
                  dregs_dataregs, r2 );
          treecode ( left_tree, dregs_dataregs );
          generate ( NODE ( t ).nodetype, r2, r1 );
          free ( dregs_dataregs );
          } /* end ELSE right subtree needs more
              regs than left subtree */
        free ( Remaining_dataregs );
        }
      break;

    default:
      {
      /* HORRIBLE MALFORMED TREE ERROR */
      }
    } /* end of SWITCH over treetype */

} /* end of int FUNCTION  treecode */
```

figure 24: Code generation algorithm for arithmetic trees.

The code figure 24 does not acknowledge the existence of complex designators or uplevel references. To handle the these cases, an address register list must be included as well as a data register list. The code has several key decision points:

1) is the current node a terminal

2) if not, are there enough registers to generate code without allocating temporary storage.

3) if there are enough registers, do the first two registers in the list need to be swapped? Put another way, does the right tree use more registers than the left tree, thus leaving the result in the wrong register if they are not reordered on entry to the next level of tree code?

What the code does not ask (and should to be adequate to modula 2 and the MC68xxx) is:

4) is this a unary op (eg. unary minus or load effective address)

5) are there enough address registers to handle the current subtree?

Rather than present the full algorithm, a subset for a simpler language and more general architecture was presented. To get an idea of the complexity introduced by the bipartate register set, think about how question 5 impacts questions 2 and 3 in the logic of figure 24. Not only must the code determine if there are enough registers, it must determine if there are enough of each kind, effectively doubling the logic required. One solution to this confusion is to use the data registers for ALL calculations including address arithmetic. Address registers would be used only when needed to generate a new value (eg. when the '^' operator was encountered or when an address load operation was required). This technique would "waste" the address register and cause an extra data register move. The current technique allows the treecode routine to assume that if any out of scope

addressing is to be performed, the head of the address register list contains the effective address operand.

In figure 24 the list routines "pure_head" and "pure_tail" correspond to the lisp functions "CAR" and "CDR" respectively. "Listlength" is a routine which counts the members in the list passed in as a parameter. The register descriptors which make up the register list contain pointers to lists of temporaries and leaves in the AST. This should facilitate register optimization over basic blocks as described by Aho [AHO 86] although the facility is not used in the current implementation.

The "get_storage" routine gets a storage temporary from the list of temporaries in the block tree structure. If none of the temporaries in the list is marked as "FREE" a new temporary is allocated and added to the list. At the time the temporary is added to the list, the total temporary storage requirement of the current block is bumped by the temporary size (4 bytes). This temporary storage size along with the parameter space size and register save area requirements are used to build the procedure epilogue code. A subsequent discussion of procedure code generation goes into greater detail.

The other major form of free form trees is the BOOLEAN expression tree. These trees were labeled with a jump target in a previous pass so that a postorder traversal could generate jump code. As soon as a boolean expression is identified by the statement processing logic, the BOOLEAN version of treecode is called to handle the jump generation. This code examines each interior node of the tree. If the node is an arithmetic operator, it calls the arithmetic version of treecode and generates code to test the value of the data register "d0", using the jump targets passed in. If the node is an operand a load and branch is generated. If the node is a BOOLEAN, the routine is called recursively.

```
int
bool_treecode ( tree, fjump, tjump, reglist )
treenode * tree;
char * fjump, * tjump;
char * reglist;
{
  switch ( tree->treetype ) {
    case LEAF:
      generate ( LOAD, pure_head ( reglist ),
                 LEAF ( tree ) );
      generate ( BRE, tree->falsejump );
      generate ( BR, tree->truejump );
      break;

    case TREE:
      if ( isboolean ( NODE ( tree ).nodetype ) )
        {
        bool_treecode ( tree, fjump, tjump, reglist );
        }
      else
        {
        treecode ( tree, reglist );
        generate ( BRE, fjump );
        generate ( BR, tjump );
        }

   default:
     {
     /* HORRIBLE MALFORMED AST ERROR */
     }
   } /* end of SWITCH over tree types */
} /* end of int FUNCTION bool_treecode */
```

figure 25: 'C'-like pseudocode for boolean expressions.

One may see from the above that 8 bytes must be added to every treenode if jump code is to be used and the jump address are outside of the union part of the treenode structure. Better differentiation of types in the data definition header files would remove some of this unnecessary space overhead. The prelabeling pass is space inefficient. Direct generation of jump code conserves the space taken by the compiler while taking a toll on optimization opportunities and runtime space requirements.

The great question at this stage is: "How much work is to be done the type checking phase and how much is to be done by the code generation phase?" Two aspects of this question are of particular importance:

1) Is the intelligence associated with address generation built into the pass which constructs addressing trees from designator lists or is it embodied by the final code generation phase?
2) Should jump targets for BOOLEAN expressions be resolved in a separate pass or in the code generation phase?

Production compilers for modern virtual memory, large address space processors will be required to generate compact, efficient code. It seems wise to trade off compile time with run time; the latter being favored. Forcing the final structure to be present at the earliest possible moment allows intermediate optimization phases to run more efficiently. Another consideration is the overall complexity of the compiler. Early versions which attempted to generate code from incomplete designator lists presented nearly intractable debugging tasks. By building a complete addressing tree, each stage now seems simpler and more comprehensible. This aspect of the current implementation is not quite complete but there appear to be no show stoppers on the horizon.

Labeling nodes with jump targets presents a problem only because of the clumsy implementation of tree nodes. Many more tree types should be used along with pointers to blocks of auxiliary information (rather than holding all variants in one large structure). With more discrete types, the fixed space overhead of jump labels would not propagate to nodes which have no requirement for jump labels.

The final remaining important area of code generation is the procedure. The activation record of every procedure is broken up into two distinct

areas, the parameter area and the local storage area. In the parameter area are the passed parameters, the return address, and the static link. Below the static link are:

1) previous frame pointer
2) saved registers
3) local variables
4) temporary storage

Every thing above the previous frame pointer (dynamic link) is the responsibility of the caller. The dynamic link and everything below it are handled by the callee.

Figure 26 below is a schematic representation the stack frame just after a procedure call. With the exception of the static link, this stack frame looks exactly like a standard UNIX 'C' language stack frame. If the static link is considered a parameter (as it is when 'C' programs call Modula 2 procedures [LUTZ 88]) then the two flavors of stack frame are indistinguishable. Indeed, except for the need to create a new static link pointer, the code for a Modula 2 prologue and for a UNIX 'C' prologue are identical. Addressing in the local activation record is relative to the current frame pointer. Each of the regions of the local stack are referenced by offsets defined by assembler pseudo ops in the procedure epilogue. As can be seen in figure 26, these are denoted by %S for the register area, %F for the local variable area and %T for the temporary area. Members of the temporary and local variable area are addressed as local positive offsets (immediate indices) to the area base. For instance, if two long word variables were the first variables to be allocated on the stack and they could be addressed as:

```
-T%0(fp)
-T%4(fp)
```

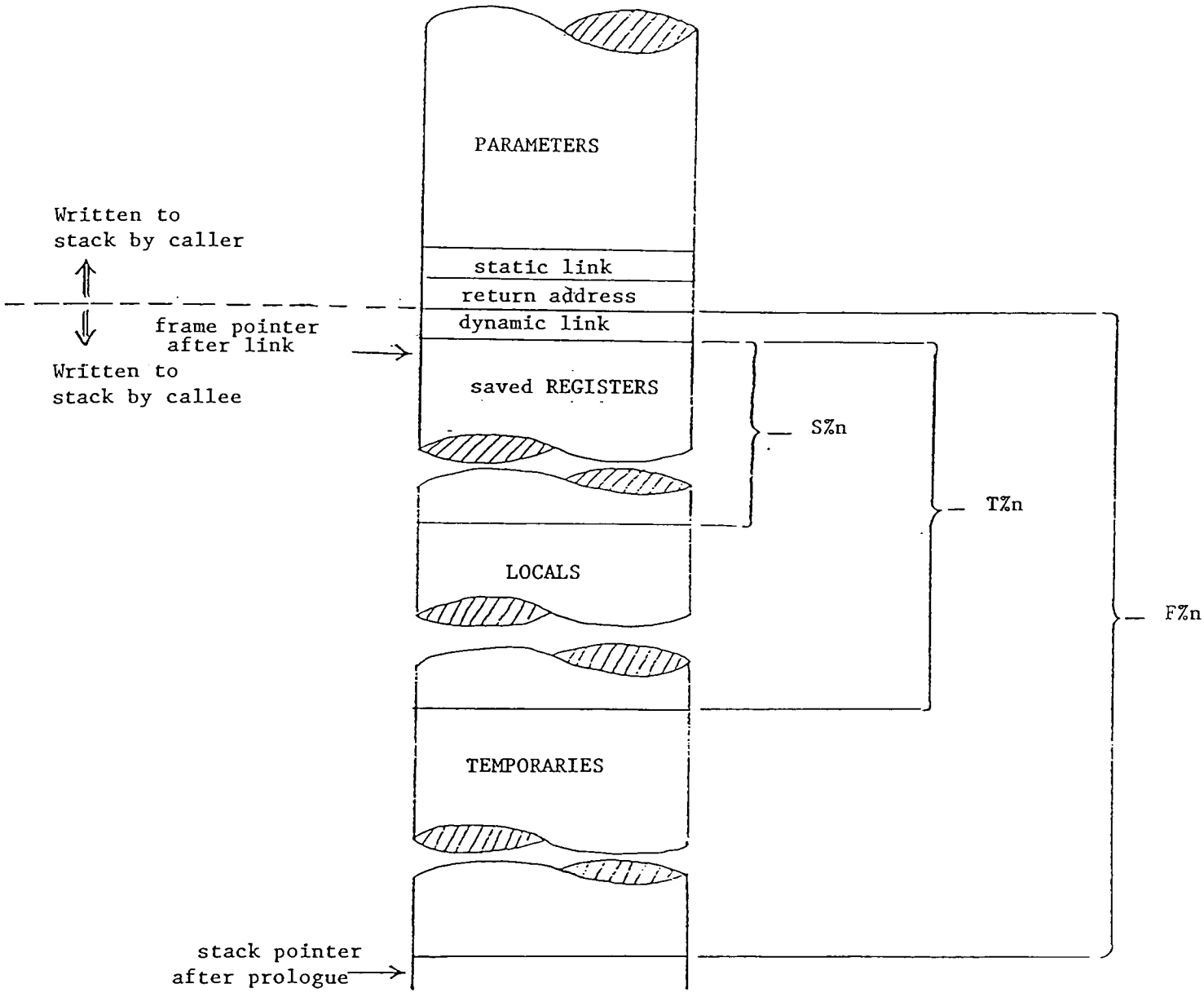

Figure 26: A Modula 2 Procedure Stack Frame

The information needed to accomplish this addressing is available at compile time. The required information is:

1) registers used (not just count, but identity of used registers).
2) temporary space used
3) space allocated for local variables
4) unique "procedure number"
5) procedure entry label

The sequence of operations by which code for a procedure are generated are roughly these:

1) emit the entry point using the entry label.
2) emit the save of the old frame pointer
3) emit the creation of the new frame pointer
4) use "$\%M_{proc\_num}$" to save the registers used by this routine. Note that "$\%M_{proc\_num}$" is merely a symbol and its value is not known at the time that the register save is emitted.
5) generate translated code for the procedure
6) emit the termination label for the procedure
7) emit the return from subroutine code
8) use information gathered in the generation process about register usage to initialize the "$\%M_{proc\_num}$" symbol (use the "set" pseudoop).
9) use the information gathered about temporary storage usage to generate the "$\%T_{proc\_num}$" symbol.
10) use the compiler generated information about local variable space usage to generate the "$\%S_{proc\_num}$" symbol.

The calling and called routines share responsibility for setting up for a procedure call and restoring the environment when the call terminates. The responsibilities of the caller are:

1) push the parameters onto the stack in an order which is the reverse of their appearance in the source call (first parameter pushed last).
2) calculate and push the static link (pointer to the stack frame of the environment which statically encloses the called procedure).
3) jump to subroutine
4) recover the stack

The called routine performs the following duties:

1) establish the dynamic link (set up new frame pointer using the "link" instruction).
2) save the registers to be used by the subroutine (MC68xxx "movm" instruction).
3) copy the static link into the static link register (a5).
4) execute subroutine code
5) restore registers (again using the "movm" instruction).
6) restore dynamic link ("unlk" instruction).
7) return from subroutine.

The calculation of the static link in the calling routine is based on the nesting level of the calling procedure minus the nesting level of the called procedure. If

$$diff_{level} = calling_{level} - called_{level}$$

then he compiler generates code for the 3 possible conditions:

$diff_{level} < 0$	look back $-diff_{level}$ static links and push that address
$diff_{level} = 0$	push current static link
$diff_{level} = 1$	push current frame pointer
$diff_{level} > 1$	HORRIBLE ILLEGAL CALL ERROR

Because procedure variables must always point to procedures declared at level 0 they will always point have a $diff_{level} <= 0$.

The calculation of this static link is performed by an assembly language routine which is part of the runtime package. A possible optimization is the omission of this calculation if the called routine makes no uplevel references and calls no subroutines.

The MC68xxx architecture greatly facilitates the generation of code for subroutine linkage and return. The *link*, *unlk*, and *movm* instructions make this aspect of code generation a delightful process.

The major interesting components of code generation are the unique procedural method for translating corresponding structure trees, the general method of translating free form BOOLEAN and arithmetic trees, and the generation of code for procedures bodies and linkage. The MC68xxx ISA is well suited to handling procedure calls but lacks sufficient generality to handle address expression trees efficiently. An experienced MC68xxx assembler programmer might find ways to improve the efficiency of the tree based code generation for expressions.

9. **Separate Compilation Facilities** -- The ability to create distinct load modules and compile them separately for subsequent linking into a complete program is at the heart of the philosophy behind Modula 2. Object code is produced from the Program Module and ancillary Implementation Modules. The interface to an Implementation Module is defined by the Definition Module. A Definition Module contains no executable code and represents a subview of a symbol table.

   It is possible, of course, for Definition Modules to define the (static) storage of all variables declared in the Definition. An Implementation

Module must "import" the Definition Module which defines its interface before any explicit imports. This implied importation can have one of two effects. Space may be allocated for any variables defined in the Definition Module or the allocation of space may be bypassed and only a symbol reference may be generated. If the latter is true then all variables declared in the Definition Module must be addressed as externs in ALL modules including the corresponding Implementation Module.

If no space is allocated at the time the Definition Module is compiled, leaving that activity to the compilation of the corresponding Implementation Module, then the most that is accomplished by the compilation of the Definition Module is a syntactic check and the creation of a linearized symbol table file. There may be little if any semantic difference between compiling a Definition Module and treating it as an include file for any module which imports it in much the same way as the 'C' preprocessor treats header or ".h" include files. One practical difference is the ability to use the linearized symbol table output from a Definition Module compilation as part of a product, leaving the source unnecessary (a dubious advantage considering that the Definition Module is as much for the programmer as the compiler). Separate compilation is discussed by several authors [FOST 86], [NEWC 87], [BRON 85]. No intermodule checking is performed by the current compiler. Intermodule checking should ideally check the dates of source against the output files to see if recompilation is required and, perhaps, automatically invoke the recompilation.

Whatever the ultimate tradeoffs involved with the technique used for interfacing modules, the approach used here is to provide a separate compilation step for Definition Modules which allocates no space and

produces an output file of Rochester Linear Form (RLF). When a module imports another module, an RLF file is what is being imported.

In the latest version of Wirth's language specification all symbols defined (as opposed to declared) in the DefinitionModule are exported. A Definition Module may import any symbols it needs to complete the definitions it creates. The current implementation is a little fuzzy about what is legal in terms of import in Definition Modules. This is because of a design problem with the FROM clause. A correct realization of Modula 2 would export only those imported symbols which are required for definitions.

After a Definition Module is parsed, the next pass walks over the hash table and "linearizes" the symbol table into RLF. The separate compilation process takes advantage of the fact that a definition module may have only one scope. This is important because a walk of a hash table with multiple scopes would "see" only those scopes which were active at the time of the walk. Even with this advantage, the hash table must be walked several times to get all of the symbol types into the RLF file with all references resolved. Only symbols with no outstanding dependencies may be written to the RLF file.

The order in which major tasks occur in Definition Module compilation are:

1) Parse - create the symbol table
2) write the RLF header file which consists of:
   a) count of identifiers in the file
   b) count of type descriptors
   c) count of attributes
3) walk the name hash table, reading names and writing them to the file (null terminated)

4) walk main hash table and write those entries which are type descriptors

5) walk main hash table and write those entries which are attribute blocks

Steps 3, 4, and 5 use the counts of the respective data types to allocate arrays of character pointers. Every time an item is written to the RLF file the index position is bumped and the address of the item written is placed in the new array position. When a reference to a block is found in the block being written, that reference is searched in the appropriate array and the index into the reference array is substituted for the actual reference. Similar arrays are built on the receiving end (import) and the reverse process is used to resolve references to subordinate blocks. This technique requires that subordinate blocks ALWAYS be written before blocks which reference them. Any time an attribute or type descriptor block refers to other blocks, those blocks must be written and marked as written (so that they will not be rewritten when encountered in the hash table).

Import is essentially the reverse of the export process. The header of the RLF file is read and the appropriate arrays are constructed. As each new entry in the file is read, it is entered into the symbol table and the address of the new block is entered into an array. When a block is found which refers to other blocks, the address of the referenced block (which is now an integer index) is used to index the array and retrieve the actual address of the referenced block which is then used to usurp the index in the referencing block. For attribute blocks, the name must be searched in the symbol table to ensure that no duplicate entries exist.

As stated earlier Implementation Modules are paired with corresponding Definition Modules which define the interface. The Implementation Module

must implicitly import the corresponding Definition Module. In addition, the address of all symbols representing procedures or storage which appear in the Definition Module must be defined as "global" in the code which the Implementation Module emits. All other modules which import this Definition Module must declare the symbols "extern".

Throughout the construction of this compiler, one recurrent theme has presented itself. Whether in the construction and traversal of lists and trees, or, as here, in the accomplishment of separate compilation, it has seemed evident that object oriented techniques would make the compiler much simpler, more portable and easier to maintain. In particular, one is left to wonder if the concept of *passivation/activation* [COX 86] might not go a long way toward automatically providing for export/import of symbols. Because these exported symbols have an internal representation in the implementation language ('C'), their structures (represented as objects and classes) could be transparently passivated on export and activated on import. Cox suggests other possibilities by giving a thumbnail sketch of a symbol mechanism defined as a class with attendant methods in Objective C.

10. A Digression on the Linked List Utilities

A list as defined here is a list head possibly linked to any number of list elements. A list head is a structure of 3 pointers; a head pointer, a tail pointer, and a current pointer. Typically the current pointer is a pointer to the last extant list element upon which an operation was performed. If the last operation was a **unlink**, the current pointer may point to the next or previous list element. The list is doubly linked and grounded at head and tail. This latter fact is important only for the search and listlength routines. Trivial modifications to each routine will easily allow the support of circularly linked lists (more important in operating systems than compilers).

List come in two distinct categories; bound and pure. In a bound list, the list structure is defined as part of the structure containing the data as in:

```
struct foo {
  q_block list;
  int     other_data;
   .
   .
   .
  }
```

Pure lists are merely a structure of 3 pointers; next, previous, and data. The data pointer is a char pointer to whatever data the user wishes to attach to the list. This is a much stronger technique which places no burden on the user data structure although it does add 4 bytes to the size of each list element and one more traversal at runtime. See Sokelman for a good discussion of lists in 'C' [SOKE 85].

The basic operations on list elements are *link, unlink, insert.* The basic operations on list are *link, unlink, search, head, tail*, and *listlength.* Search takes a list head, a comparison routine and success criteria as parameters and returns the block found (possible NULL). Using a generic routine for search dramatically cuts down on coding errors. Lisp programmers will recognize *head* as "CAR" and *tail* as "CDR".

A set of macros is supplied to increase reliability in accessing lists. Included among these are GETHEAD, GETTAIL, GET_NEXT, and GETPREV. When dealing with pointers as one does with lists, procedures are more secure than macros (allows checking for NULL arguments) and the user may write the simple procedural equivalents with little effort.

The list routines provide a clear example of how lists might be better implemented using an object oriented paradigm. When searching a list, the generic list routine performs basic "get next and compare" operations. This

works well for the great majority of searches but not for the few cases which have unusual search criteria (eg. positional criteria). If the primitive list mechanism could be *inherited* by other structures, the need for the elusive completely general purpose searching method would disappear.

Levy proposes a set of linked list primitives for a language which he calls Modula 3 [LEVY 88]. His paper provides an interesting counterpoint to the 'C' language routines presented here.

11. Implementation Notes (make files etc.)

A simple make file is used to create the compiler. Not all dependencies have been added to this file but it is serviceable for the knowledgeable user. The form of the make file was abstracted from Kernighan and Pike [KERN 84] who describe in great detail the development cycle of a simple language using UNIX tools.

Shell scripts for linking the runtime library and the 'C' library (which is required because of the use of 'C' runtime support) would be helpful and should be available soon.

B. Data Bases

The symbol data base has been discussed at length in a previous section. One data base which has not been mentioned is the database of symbols which was used to test the quality of the hashing mechanism. A statistical analysis routine was used to examine all the symbols from several large programming projects. The symbols from all files in all of the projects were run through the analyzer. Hash table entries were treated as histogram bins and the min, max, mean, median and standard deviation of the number of hits in each bin were calculated. The standard deviation algorithm used a

technique which eliminated the overflow problem usually associated with squaring the sums.

Because the analysis was run on a proprietary database, the detailed results are not available. It may be said, however, that the performance of the simple hashing mechanism used here was surprisingly good. By multiplying the character value by its position in the key and summing these values for all characters in the key, the distribution of keys in the table was acceptable over a wide range of table sizes.

V. VERIFICATION AND VALIDATION

The verification of a compiler is a large and difficult task. To the extent that the implementation proceeds from tight, formal specifications with an eye toward verification, the pain of testing may be mitigated. It is always wise to design and test by the process of problem decomposition. Each module verified in unit testing will contribute confidence in the integrated system. Any errors which appear in integration testing using known good data from unit tests suggest a problem in the interface. The 7 major modules or phases below are tested, to the extent reasonable and possible, as stand alone units. This task becomes more difficult as later stages are reached. Using a file based interface mechanism such as IDL would make the entire process more reliable. The ongoing development of the system would normally refine and extend the test bed.

A. **Lexical Analyzer** -- A simple driver which prints the line number, token type, and string of the input is used for unit test of the lexical analyzer. All Modula 2 keywords as well as well-formed identifiers and literals are passed through the test bed. Malformed identifiers and literals are used to test the generation of the error token.

B. Parser -- The completed parser with semantic actions cannot provide an adequate test bed for the ability to parse input sentences. One must isolate the actions of the parser from the effects of the semantic actions. A 'C' utility which strips the actions from the grammar is used to generate a parse only version of the program. A more complete utility would employ yacc to parse the yacc grammar so that include files, switches, typing of the grammar stack and other inessential pieces of the grammar would be removed (a task which is currently performed by hand).

A good test of the parser would be to run the ETH compiler through the parse only compiler to determine it behavior under "ideal" conditions. A suite of malformed constructs constitutes a partial test of error detection capabilities.

An examination of the grammar should allow the programmer to write cases which test every production. Admittedly, this has not been done for the compiler under consideration here, but the way is clear. The test cases which have been written are those which seemed difficult or error prone based on the concerns of the implementor.

C. Symbol Table -- Unit testing of the symbol table mechanism is somewhat difficult due to the amount of detail which must be taken into consideration when generating test input. For instance the nesting level, enclosing scope environment, scope stack, symbol stack, and line number information must all be managed correctly. A driver to handle the many activities of the symbol data base would be nearly as complex as the real environment but would be well worth the effort for a production compiler. Such a driver should be designed to give visibility to all elements of the state vector represented by some appropriate graphical interface. A simplified

mechanism for accessing the symbol routines would allow independent verification of error and facilitate the tracing of data flow in the system.

Testing has taken a more integrated approach here. A nearly complete front end was run against real Modula 2 programs (some with real bugs) and the output (or failure of the first pass) was examined to evaluate the quality of the symbol mechanism. Fortunately, the symbol database mechanism has been the most trouble free of part of the system. It was completely coded (over 1400 lines of 'C' code) before any of it aside from the hashing mechanism was tested. The module assumed what was to be its final form after only 12 hours of debugging.

The only real errors did not show up until the compiler was generating code. At that time it was discovered that local variable offsets in the runtime stack were not being correctly added to the procedure blocks. Another problem of the same nature cropped up when the grammar was rewritten several weeks ago to accommodate collection of closed scopes during the parse. Finding and fixing these bugs was extremely difficult because of the lack of a unit test capability. A symbol table browser would be helpful but is well beyond the limitations of this project.

D. Second Pass -- The system component which currently constitutes the second pass is the type checking and address tree rewrite phase. This code is extremely difficult to unit test because of the lack of a good diagnostic dump of the tree. A tree dump utility does exist (trdiag.c) but it only identifies nodes and their nesting. No information regarding size and type is available. This code has been tested by running very small programs through the compiler and examining the output assembly code. Type check tests will, of course, abort compilation on type mismatches.

IDL as an interface tool would be extremely helpful here. One of the suggestions for further work in section VI is for a tree editor; a program which displays trees graphically in any level of detail and which can create trees as well. This would be a phenomenal learning and diagnostic tool. The Smalltalk class browser is an example of this type of tool and one which might be a good editor prototype for a compiler which was written in a suitable object oriented language. A Smalltalk-like "Inspector" would allow modifying the trees.

E. **Preparation for Code Generation** -- The possibilities for unit test decrease as the latter stages of the compiler are reached. An extant tree is required as input to, and a modified tree is output by the preparation for code gen phase. Without a tree editor which can build and view trees, unit testing is difficult here.

Testing this phase has been difficult and has been accomplished by running a complete compiler against small test cases. The output of the compiler is then examined to try to determine what went wrong or right. This method relies in large part on intuition and knowledge of the internal workings of the compiler. Given the size and complexity of the system, the testing of this phase leaves much to be desired. Again, a file based intermediate form which could be supplied by IDL and a tree browser would go far toward rationalizing this process.

Because the preparation for code generation consists mainly of labeling expression tree nodes, the bulk of testing is limited to free form trees. Confidence in the system may be incrementally built up by using a large set of simple test cases and gradually increasing their complexity. For example, the first tier testing involves factor only expressions with no operators (eg.

"a=b;"). The next level is the testing of expressions with a single binary operator (eg. "a:=b+c;"), and so on.

A good test of the labeling algorithm for arithmetic expressions is:

```
a:=b+c[d^,-6]  [somefunc(c[1,2,3+4*5DIV(6REM7)]MOD 4
```

The production of correct code and a pattern of register usage which reflects the complexity of the expression are currently the only means of verifying the test of this section.

F. **Code Generation** -- Code generation is similar in most respects to the forgoing section with respect to testing. Running test cases of ever increasing complexity through a completed compiler is the manner in which the code generation algorithms were tested. Handling of addressing of designators can be very complex and the version of the compiler which attempts to perform this task is still under test. Exactly what constitutes a good test is not intuitively obvious and surprisingly simple test cases fail while complex examples pass with flying colors. Trying lots of programs written by someone other than the implementor seems to be the best course of action in this environment.

The generation algorithms themselves are recursive, (eg. treecode, bool_treecode, gen_while, gen_for, gen_proc, etc.) and deal with trees and lists. Testing and debugging in a recursive environment is notoriously difficult. The code emission routines are quite simple and, for the most part, only use simple information to create an appropriate "printf". Testing the emit routines is simple and straight forward.

VI. CONCLUSIONS

A. **Paths Not Taken** -- Tree based symbol tables were not used but, for experimental compilers at least, they may provide a more flexible alternative to the mechanism used here.

Recursive descent parsers seem like unlikely candidates for directing the translation of languages such as Modula 2 due to the requirement for forward references. LL parsers do seem, however, to be gaining popularity and seem to be used almost exclusively in Europe. Given the chance to do it all over again, yacc would still be the parser of choice. This decision impacts the choice of implementation language because yacc will not work with any language but 'C'. 'C' seems unsuited to programming in the large and its lack of type security has proved exasperating throughout the project. The advent of precompilers for OOPS languages appear to go a long way toward correcting 'C's problems with respect to large programming projects. The power of these new languages has even caused one neophyte compiler writer to reconsider the ultimate practical value of Modula 2.

B. **Things that Would Be Done Differently** -- The development of this compiler has been characterized by struggling to understand the requirements of the language and the interplay of the necessary code and structures which realize the language. Given the advantage of hindsight it would have been better by far to use Objective C or C++ as the implementation language. It is unlikely that the first time writer of a compiler could take advantage of such tools effectively, but maintenance would have been easier and tool building would have been available at a smaller incremental cost. Looking back over the experience, it is astonishing that compiler writers who build

object oriented languages do not feel that using object oriented programming systems (OOPS) increase the reliability of compilers [JOHN 87].

A linear intermediate form which is produced from the AST now appears to be worth the extra effort involved in its creation. Anklam, Powell and others used such a form in their tree based compilers and it now appears they did so with good reason. Dealing with the raw interface of any given ISA can be a brutal experience. Building a buffering layer between the tree and the machine would be time well spent.

A member of the committee which has overseen this work suggested that the development proceed in a "depth first" fashion. The suggested goal was to generate code for some simple subset of the language to get a feel for an problems that might lie hidden [KITC 87]. This was sage advice which was not completely heeded. Because yacc seemed such a conundrum during the early stages of development, the focus of the effort was to build a completely functioning parser. After all, the remaining effort would be "just 'C' code" and would be trivial in comparison to yacc. This was folly. Address generation and type checking for designators are formidable problems and ones which demand a measure of research and a good deal of "think time". A depth first development effort with an appropriate subset of the language would have resulted in a cleaner compiler and more efficient and more bugfree code. Anyone wishing to create a compiler from scratch should seriously consider taking the time to design an appropriate subset.

Compiler writing demands a sophisticated and mature programming environment. The lesson of this effort has been that the standard tools, while powerful and necessary, do not an environment make. Tools such as y+ [PARK 88], IDL [NEWC 87], visual yacc, and innumerable others suggest that professional language implementors recognize the value of creating an

environment which facilitates their efforts. Writing a compiler without these tools can be a painful experience. The novice does not begin to write compilers which are aesthetically pleasing until he has written at least two which are not [SEVE 88]. One wishes for more complete descriptions of the environments created in the language labs of CMU and Brown University. Perhaps the time spent building this compiler would have been passed more effectively building a tool set for other compiler writers. But the experience of the last several months has seems to have been necessary to gain the perspective just expressed. This sentiment leads into a set of hopes and aspirations for work to follow.

C. Suggestions for Further Work

The work previously described has brought to light countless opportunities for investigation in the areas of compiler construction, programming environments, systems architecture, and education. The following short list is offered without further comment.

1. Complete the compiler with respect to Reals, variant records, and WITH statements.
2. Optimize the compiler(tree and peephole, see Powell[POWE 84a], Frailey[FRAI 79]). Run bottlenecks in the compiler itself through a "super optimizer" (one is available for the MC68000).
3. Clean up code generation to make more portable. Code emission in particular is an unfortunate, *ad hoc* implementation. Rather than UNIX assembler for the MC68xxx, generate an output file which may be translated to assembler for the MC68xxx, the VAX, the 32200, or the 32x32 (this is a surprisingly easy task). Rewrite the generation routines to generate code for the 80x86 series from Intel.

4. Rewrite compiler (including parser semantic actions) in an object oriented language (C++ or Objective C). Make a tree a Class and various tree and node types subclasses (allows "compile yourself message types)
5. Bring intermediate form into full compliance (to the extent meaningful and reasonable) with Ada DIANA Tree structure. Build a graphic tree editor a la Smalltalks class browser which looks at IDL objects. The implementation of IDL should be done in an object orient 'C' like language.
6. Use IDL to define interface between passes and to linearize the symbol tree output of Definition modules.
7. Add the type **Object** to the language definition and write a pre-compiler which supports function call type messaging to convert Modula 2 into a fully object oriented language.[BERG 88], [COX 86], [WIRT 89], [WIRT 89A].
8. Build a teaching system which uses visual yacc (U.of Washington) as the parser.
9. Extend 8 to use the University of Washington visual compiler system.
10. Extend the Modula specification to allow overloading as does Ada (the current implementation will support this extension).
11. Finish the system module and add support for the true multiprocessing. See work describing the firefly project at DEC's WRC for inspiration [THAC 87]. Use this and the work outlined in VI.C.7 above to investigate the relationship between the object-oriented and the process paradigms.

VII. BIBLIOGRAPHY

[AHO 86]. Aho, A.V., Sethi, R., and Ullman, J.D. Compilers Principles, Techniques, and Tools Addison-Wesley, Reading, Massachusetts, 1986.

[ANDE 73] Anderson, T., Eve, J., and Horning, J.J. Efficient LR(1) Parsers Acta Informatica, Vol. 2, No. 1, 1973, 12-39

[ANDE 86] Anderson, T.L. The Scope of Imported Identifiers in Modula 2. In SIGPLAN Notices, Vol. 21, no. 9, September 1986 17-21

[ANKL 77] Anklam, P., Cutler, D., Heinen, R.Jr., and MacLaren Engineering a Compiler Digital Press, Bedford, Massachusetts, 1977.

[AT&T 85] AT&T UNIX PC UNIX System V User's Manual AT&T 1985

[AT&T 85a] AT&T UNIX PC UNIX System V Programmer's Guide AT&T 1985

[ATTE 89] Atteson, K., Lorenz, M., and Dowling W.F. NARPL: A Solution to the Student Compiler Project Problem SIGPLAN Notices Vol. 24, no. 3, March 1989 57-66.

[BACH 86] Bach, M.J. The Design of the UNIX Operating System Prentice-Hall, Englewood Cliffs, N.J. 1986

[BARR 79] Barrett, W.A., and Couch, J.D. Compiler Construction: Theory and Practice Science Research Associates 1979

[BARR 81] Barron, D.W. (Editor) PASCAL - The Language and its Implementation John Wiley & Sons, Chichester, 1981

[BEN- 81] Ben-Ari, M. Cheap Concurrent Programming. In Software Practice and Experience, Vol. 11, No. 12, 1981 1261-1264

[BEN- 82] Ben-Ari, M. Principles of Concurrent Programming Prentice-Hall International, Englewood Cliffs, N.J. 1982

[BIND 85] Binding, C: Cheap Concurrency in C. In SIGPLAN Notices, Vol. 20, no. 9, September 1985 21-26

[BERG 88] Bergen, J., and Greenfield, S. What Does Modula-2 Need to Fully Support Object Oriented Programming? In SIGPLAN Notices, Vol. 23, No. 3, March 1988 73-82

[BRIN 85] Brinch Hansen, P. On PASCAL Compilers. Prentice-Hall, Englewood Cliffs, N.J. 1985

[BRON 85] Bron, C., Dijkstra, E.J. A Note on the Checking of Separately Compiled Modules In SIGPLAN Notices Vol. 20, no. 8 August 1985 60-63

[BROW 84] Brown, C.A., and Purdom, P.W. Jr. A Methodology and Notation for Compiler Front End Design. In Software-Practice and Experience, Vol. 14,no. 4, April 1984 335-346.

[CHAN 84] Chang, C.C. The Study of an Ordered Minimal Perfect Hashing Scheme. In Communications of the ACM, Vol. 27, no. 4, April, 1984 384-387

[CHIR 84] Chirlian, P. Introduction to Modula-2. Matrix Publishers, Inc., Beaverton, Oregon, 1984

[CHRI 86] Christian, K. A Guide to Modula-2. Springer-Verlag, New York, 1986

[CHOW 83] Chow, F.C., and Ganapathi,M. Intermediate Languages in Compiler Construction - A Bibliography. In SIGPLAN Notices, Vol. 18, no. 11, November 1983 21-23.

[COLL 87] Collado, M., Morales, R., and Moreno, J.J. A Modula 2 Implementation of CSP. In SIGPLAN Notices Vol. 22, no. 6 June 1987 25-38

[COX 86] Cox, B.J. Object Oriented Programming. Addison-Wesley, Reading, Massachusetts, 1986

[DLUG 88] Dlugosz, J.M. A Multitasking Kernel for C Programmers. In Computer Language Vol. 5, No. 10. October 1988 49-60

[DOS 88] Dos Reis, A.J. A Note on Ben-Ari's Concurrent Programming System. In Operating Systems Review Vol. 22, No. 3. July 1988 41

[DREC 87] Drechsler, K.H., and Stadel, M.P. The PASCAL-XT Code Generator. In SIGPLAN Notices Vol. 22, no. 8, August 1987 57-78

[FISC 88] Fischer, C.N. and LeBlanc, R.J.,Jr. Crafting a Compiler Benjamin/Cummings, Menlo Park, California 1988

[FLOY 87] Floyd, E.T. Hashing for High-Performance Searching in Dr. Dobb's Journal, February 1987

[FOST 86] Foster, D.G. Separate Compilation in a Modula-2 Compiler. Software-Practice & Experience, Vol. 16, no. 2, February 1986 101-106.

[FRAI 79] Frailey, D.J. An Intermediate Language for Source and Target Independent Code Optimization. IN SIGPLAN Notices, Vol. 14, no. 8, August 1979 188-200.

[GAIT 84] Gait, J. Semaphores Outside the Kernal. In SIGPLAN Notices, Vol. 19, no. 10, October 1984 12-21.

[GERA 87] Gerardy, R. Experimental Comparison of Some Parsing Methods. In SIGPLAN Notices Vol. 22, no. 8, August 1987 79-88

[GISS 86] Gisselquist, R., An Experimental C Compiler for the Cray 2 Computer. SIGPLAN Notices Vol. 21, no. 9, September 1986 32-41

[GRAH 80] Graham, S.L. Table Driven Code Generation. In IEEE Computer Magazine, Vol. 13, no. 8, August 1980 25-34.

[GRAH 79] Graham, S.L., Joy, W.N., and Roubine O. Hashed Symbol Tables for Languages with Explicit Scope Control. In Proceedings of the SIGPLAN'79 Symposium on Compiler Construction. 1979 50-57.

[GROE 86] Groening, K., and Ohsendoth, C. NEMO: A Nicely Modified YACC. In SIGPLAN Notices, Vol. 21, no. 4, April 1986 58-66.

[HART 85] Hartel, P.H., and Starreveld, D. Modula-2 Implementation Overview. In Journal of Pascal, Ada, & Modula-2, Vol. 4, no. 4, July/August 1985.

[HOAR 85] Hoare, C.A.R. Communicating Sequential Processes Prentice-Hall International, Englewood Cliffs, N.J. 1985

[HOLT 83] Holt, R.C., Concurrent Euclid, the UNIX System, and Tunis. Addison Wesley, Reading, Massachusetts, 1983.

[HOPC 79] Hopcroft, J.E., and Ullman, J.D. Introduction to Automata Theory,Languages, and Computation Addison Wesley, Reading, Massachusetts 1979.

[HUNT 85] Hunter, R. Compilers, Their Design and Construction Using Pascal Wiley, Chichester, England 1985.

[JOHN 87] Johnson, R. Workshop on Compiling and Optimizing Object-Oriented Programming Languages. In the Addendum to the Proceedings of the Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87). Special Issue of SIGPLAN Notices, Volume 23, Number 5, May 1988.

[JOHN 75] Johnson, S.C. YACC - yet another compiler compiler Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J.

[KATW 83] van Katwijk, J. A preprocessor for YACC or A poor man's approach to parsing attributed grammars. In SIGPLAN Notices, Vol. 18, No. 10 October 1983 12-15

[KERN 84] Kernighan, B.W., and Pike, R. The UNIX Programming Environment Prentice Hall Software Series Englewood Cliffs, New Jersey 1984

[KITC 87] Kitchen, A. Personal Communication

[KNUT 73] Knuth, D.E. The Art of Computer Programming Vol. 2, Addison Wesley, Reading, Massachusetts 1973

[KNUT 65] Knuth, D.E. On the translation of languages from left to right Information and Control, Vol. 8, No. 6, 607-639

[LAMB 87] Lamb, D.A. IDL: Sharing Intermediate Representations. In ACM Transactions on Programming Languages and Systems, Vol. 9 No. 3, July 1987 297-318

[LEES 87] Leeson, J.J., and Spear, M.L. Type Independent Modules: The Preferred Approach to Generic ADTs in Modula-2. In SIGPLAN Notices Vol. 22, No. 3, March 1987 65-70

[LESK 75] Lesk, M.E., and Schmidt, E. 1975 "Lex- a lexical analyzer generator." In UNIX Programmer's Manual 2, AT&T Bell Laboratories, Murray Hill, N.J.

[LEVE 80] Leverett, B.W., Cattel, R.G.G., Hobbs, S.O., Newcomer, J.M., Reiner, A.H., Schatz, B.R., and Wulf, W.A. An Overview of the Production-Quality Compiler-Compiler Project. In IEEE Computer Magazine, Vol. 13, no. 8, August 1980 38-49.

[LEVY 88] Levy, E., The Linked List Class of Modula-3. SIGPLAN Notices Vol. 23, no. 8, August 1988 93-102

[LO 85] Lo, C. Simple Patches to Modula-2 Text I/O. SIGPLAN Notices, Vol. 20, no. 6, June 1985 20-25.

[LUTZ 77] Lutz, P., and Oldham, D. FOIL - an Intermediate Language for FORTRAN. Technical Report Number 128, State University of New York at Buffalo, 1977.

[LUTZ 88] Lutz, P. Personal Communication.

[MCKE 87] McKeeman, W.M., Trager, S., Cohen, J.L., and Yang, T. C Grammars. Technical Report TR-87-02 Wang Institute of Graduate Studies

[MOFF 84] Moffat, D.V. Some Concerns About Modula-2. In SIGPLAN Notices, Vol. 19, no. 10, October 1984 41-47.

[MOHA 87] Mohay, G. A Simplified Coroutine Structure for Modula-2. In Journal of Pascal, Ada, and Modula-2, Vol. 6, No. 1, January/February 1987 35-42

[MOSS 86] Mossenbock, H. Alex - A Simple and Efficient Scanner Generator. In SIGPLAN Notices, Vol. 21, no. 5, May 1986 69-78.

[MOTO 86] Anon. M68000 8-/16-/32-Bit Microprocessors Programmer's Reference Manual fifth edition Prentice-Hall Englewood Cliffs, N.J. 1986

[MULL 84] Muller, H. Differences Between Modula-2 and Pascal. SIGPLAN Notices, Vol. 19, no. 10, October 1984 32-39.

[NEWC 87] Newcomer, J.M. Efficient Binary I/O of IDL Objects. SIGPLAN Notices, Vol. 22, no. 11, November 1987 35-43

[NICH 88] Nicholl, R.A., A Specification of Modula-2 Process (Coroutine) Management. Journal of Pascal, Ada, & Modula-2, Vol. 7, No. 5, September/October 1988 16-22

[OTTE 84] Ottenstein, K.J., Intermediate Program Representations in Compiler Construction -A Supplemental Bibliography. SIGPLAN Notices, Vol. 19, no. 7, July 1984 25-27.

[PARK 88] Park, J.C.H., y+: A Yacc Preprocessor for Certain Semantic Actions. SIGPLAN Notices, Vol. 23, No. 6, June 1988 97-106.

[PEMB 82] Pemberton, S., and Daniels, M.C. PASCAL Implementation - The P4 System Ellis-Horwood Publishers, Chichester, England

[POWE 84] Powell, M.L., Modula-2: Good News and Bad News (Optimizing Compiler). In Digest of Papers Compcon Spring 1984. 28th IEEE Computer Society International Conference 1984 438-441.

[POWE 84a] Powell, M.L., A Portable Optimizing Compiler for Modula-2. SIGPLAN Notices, Vol. 19, no. 6, June 1984 310-318.

[PRAM 84] Pramanik, S., and Weinberg, B., The Implementation Kit with Monitors. SIGPLAN Notices, Vol. 19, no. 9, September 1984 .

[PYST 88] Pyster, A.B., Compiler Design and Construction Van Nostrand Reinhold New York, New York 1988

[REIS 83] Reis, S., Generation of Compiler Symbol Processing Mechanisms from Specifications. ACM Transactions on Programming Languages and Systems, Vol. 5, No. 2, April 1983 127-163.

[RIBA 88] Ribar, J. Cooperative Multitasking in Modula-2. In Computer Language, Vol. 5, No. 10, October 1988 63-68

[RUSS 83] Russel, S., and Amin, A., An Error-Correcting LL(1) Parser Generator. Proceedings of the 6th Australian Computer Science Conference, February 1983 163-165.

[SCHR 85] Schreiner, A.T., and Friedman, H.G. Jr., Introduction to Compiler Construction with UNIX Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[SEGR 85] Segre, S, and Stanton, M. "Some Concerns About Modula-2" Considered Unwarranted. In SIGPLAN Notices, Vol. 20, no. 5, May 1985 31-35.

[SEVE 88] Severski, M. Personal Communication

[SEWR 84] Sewry, D.A., Modula-2 and the Monitor Concept. SIGPLAN Notices, Vol. 19, no. 11, November 1984 33-41.

[SEWR 84a] Sewry, D.A., Modula-2 Process Facilities. SIGPLAN Notices, Vol. 19, no. 11, November 1984 23-32.

[SOKE 85] Sokelman, G.E., Krekelberg, D.E. Advanced C: Techniques & Applications Que Corporation, Indianapolis, Indiana 1985

[SPEC 83] Spector, D., Lexing and Parsing Modula-2. SIGPLAN Notices, Vol. 18, no. 10, October 1983 25-32.

[SPEC 87] Spector, D., and Turner, P.K. Limitations of Graham-Glanville Style Code Generation. In SIGPLAN Notices Vol. 22, no. 2, February 1987 100-108

[STRO 86] Strom, R. A Comparison of the Object-Oriented and Process Paradigms. In SIGPLAN Notices Vol. 21, no. 10, October 1986 88-97

[TANN 83] Tannenbaum, A.S., Van Staveren, H., Keizer, E.G., and Stevenson, J.W., **A Practical Toolkit for Making Compilers.** Communications of the ACM, Vol. 26, no. 9, September 1983 654-660.

[TANN 84] Tannenbaum, A.S., Keizer, E.G., and Van Staveren, H., **Does Anybody Want to Write Half of a Compiler?** SIGPLAN Notices, Vol. 19, no. 8, August 1984 106-118.

[TANN 87] Tannenbaum, A.S. Operating Systems Design and Implementation Prentice-Hall, Englewood Cliffs, N.J. 1987

[TERR 86] Terry, P.D., **Modula-2 Kernal Supporting Monitors.** In Software-Practice & Experience,May 1986 457-472.

[TERR 86a] Terry, P.D. Programming Language Translation Addison-Wesley, Reading, Mass 1986

[TERR 87] Terry, P.D. An Introduction to Programming with Modula 2 Addison-Wesley, Reading, Mass 1987

[THAC 87] Thacker, C.P., and Stewart, L.C. **Firefly: a Multiprocessor Workstation.** In Proceedings Second International Conference on Architectural Support for Programming Languages and Operating Systems October 1987 164-172

[TORB 87] Torbett, M.A. **More Ambiguities and Insecurities in Modula 2** In SIGPLAN Notices Vol. 22, no. 5, May 1987 11-17

[TREM 82] Tremblay, J.P., and Sorensen, P.G. Compiler Writing McGraw-Hill, New York 1982.

[VAND 89] van Delft, A.J.E., **Comments on Oberon**, SIGPLAN Notices Vol. 24, no. 3, March 1989 23-30.

[TREM 82a] Tremblay, J.P., and Sorensen, P.G. An Implementation Guide to Compiler Writing McGraw-Hill,New York 1982.

[WAIT 84] Wait, W., and Goos, G. Compiler Construction, Springer-Verlag, New York, 1984

[WELS 86] Welsh, J., and Hay, A. A Model Implementation of Standard PASCAL Prentice/Hall International, Englewood Cliffs, New Jersey 1986

[WIRT 77a] Wirth, N., **Modula: a Language for Modular Multiprogramming.** In Software Practice and Experience, Vol. 7, 1977, 3-35

[WIRT 77b] Wirth, N., **The Use of Modula.** In Software Practice and Experience, Vol. 7, 1977, 37-65

[WIRT 77c] Wirth, N., **Design and Implementation of Modula.** In Software Practice and Experience, Vol. 7, 1977, 67-84

[WIRT 85] Wirth, N., **Revisions and Amendments to Modula-2.** In Journal of Pascal, Ada, and Modula-2, Vol. 4, no. 1, January/February 1985 25-28

[WIRT 85a] Wirth, N., Programming in Modula-2. Third, Corrected edition. Springer Verlag, Berlin 1983.

[WIRT 86] Wirth, N., Microprocessor Architectures: A comparison Based on Code Generation by Compiler. Communications of the ACM, Vol. 29, no. 10, October 1986 978-990.

[WIRT 86a] Wirth, N. Algorithms & Data Structures Prentice-Hall, Englewood Cliffs, N.J. 1986

[WIRTH 88] Wirth, N. From Modula to Oberon, Software-Practice and Experience, Vol. 18, no. 7, July 1988 661-670.

[WIRTH 88a] Wirth, N. The Programming Language Oberon, Software-Practice and Experience, Vol. 18, no. 7, July 1988 671-690.

APPENDIX A

```
%{
/* ------------------------------------------------------------------------

                        L E X M O D . L

        A lexical analyzer for Wirth's
        revision 3.1 of Modula 2


VERSION:        BETA 0.1        rfs     for pre-release to RIT Public Domain
                                        Software Library. 1/3/89

------------------------------------------------------------------------ */

#include <stdio.h>
#include "stdtyp.h"
#include "sym_type.h"
#include "trees.h"
#include "name.h"

#ifdef DEBUG /* debugging version */

#define token(x) (int) "x"

#else

#include "y.tab.h"
# define token(x) x

#endif

#include "str_fxns.h"

extern int line_num;
extern int atoi();
%}


%%
    char c;
    int  nest_level;    /* MODULA-2 may nest comments.... */
[\n]                    { showlino (); };
[ \t]           ;       /* strip white space */
"(*"            {
                strip_com();
                }

"+"                                     return token ('+') ;
"-"                                     return token ('-') ;
"*"                                     return token ('*') ;
"/"                                     return token ('/') ;
":="                                    return token (ASSIGN) ;
"="                                     return token ('=') ;
"#"                                     |
"<>"                                    return token (NOTEQL) ;
```



```
"<"                                     return token ('<');
">"                                     return token ('>');
"<="                                    return token (LTE);
">="                                    return token (GTE);
"("                                     return token ('(');
")"                                     return token (')');
"["                                     return token ('[');
"]"                                     return token (']');
"]"[ \t\n]*"["                          return token (CATELEM);
"{"                                     return token ('{');
"}"                                     return token ('}');
"^"                                     return token ('^');
"."                                     return token ('.');
".."                                    return token (RANGE);
AND                                     |
"&"                                     return token (AND);
ARRAY                                   return token (ARRAY);
"@C"                                    return token (AT_SEA);
BEGIN                                   return token (BEGINTOK);
BY                                      return token (BY);
CASE                                    return token (CASE);
CONST                                   return token (CONST);
DEFINITION                              return token (DEFINITION);
DIV                                     return token (DIV);
DO                                      return token (DO);
ELSE                                    return token (ELSE);
ELSIF                                   return token (ELSEIF);
END                                     return token (END);
EXIT                                    return token (EXIT);
EXPORT                                  return token (EXPORT);
FOR                                     return token (FOR);
FORWARD                                 return token (FORWARD);
FROM                                    return token (FROM);
IF                                      return token (IF);
IMPLEMENTATION                          return token (IMPLMN);
IMPORT                                  return token (IMPORT);
IN                                      return token (IN);
LOOP                                    return token (LOOP);
MOD                                     return token (MOD);
MODULE                                  return token (MODULE);
NOT                                     |
"~"                                     return token (NOT);
OF                                      return token (OF);
OR                                      return token (OR);
POINTER                                 return token (POINTER);
PROCEDURE                               return token (PROCEDURE);
QUALIFIED                               return token (QUALIFIED);
RECORD                                  return token (RECORD);
REM                                     return token (REM);
REPEAT                                  return token (REPEAT);
RETURN                                  return token (RETURN);
SET                                     return token (SET);
THEN                                    return token (THEN);
TO                                      return token (TO);
TYPE                                    return token (TYPE);
UNTIL                                   return token (UNTIL);
VAR                                     return token (VAR);
```

```
WHILE                                   return token (WHILE);
WITH                                    return token (WITH);
"@C"                                    return token (AT_SEA);
[a-zA-Z][a-zA-Z0-9]*                    {
                                         showtok();
                                         if ( NULL == ( yylval.y_cp =
                                               install_text ( yytext ) ) )
                                           {
                                           return token ( ERROR );
                                           }
                                         else
                                           {
#ifdef YYDEBUG
printf ( "LEX returns name [%s]\n",yylval.y_cp );
#endif
                                           return token (IDENT);
                                           }
                                        }

[0-7]+[B]                               {intin (yytext, yyleng - 1, atoo, INTEGER
 );

                                        return token (NUMBER);};


[0-7]+[C]                               {intin (yytext, yyleng - 1, atoo, CHAR );

                                        return token (NUMBER);};


[0-9]+                                  {intin ( yytext, yyleng, atoi, INTEGER );

                                        return token (NUMBER);};


[0-9][0-9A-F]*H                         {intin ( yytext,yyleng - 1, atoh, INTEGER
 );

                                         return  token ( NUMBER ); };

[0-9]+[0-9A-F]*[D-F]+[0-9A-F]*          {intin ( yytext, yyleng, atoh, INTEGER );

                                        return token (NUMBER);};

[0-9]+"."[0-9]*("E"[+|-]?[0-9]+)?/[^.]  return token (REAL);

","                                     return token (',');
";"                                     return token (';');
":"                                     return token (':');
"|"                                     return token ('|');
'[^'\n]*'
\"[^"\n]*\"                             {
                                        stringin ( yytext + 1, yyleng - 2 );
                                        return token (STRING);
                                        };
.                                       return token (ERROR);
%%

int
strip_com()
```



```
{
        char c;

        do
        {
        c = input();
        } while ( ! check_end ( c ) );
}

int
check_end ( c )
char c;
{
        char nxt;

        switch (c)
                {
                case '(':       /* COULD BE START OF NESTED COMMENT */
                        if ( ( nxt = input() ) == '*' )
                                strip_com();
                        else
                                unput ( nxt );
                        return ( 0 );

                case '*':       /* COULD BE END OF CURRENT NESTED COMMENT */
                        if ( ( nxt = input() ) == ')' )
                                return ( 1 ); /* we found end of comment */

                default :
                        return(0);
                }

}

int
showlino ()
{
#ifdef YYDEBUG
        printf ( "finishing line number ----> [%d]\n", yylineno-1 );
#endif
        return ( 0 );
} /* end of "showlino" */

int
showtok ()
{
#ifdef YYDEBUG
        printf ( "Identifier is ============> %s\n", yytext );
#endif
        return(0);
} /* end of "showtok" */
```

APPENDIX B

```
%{
/* ======================================================================
THIS FILE :             MODULA.Y

VERSION:        BETA 0.1        rfs     for pre-release to RIT Public Domain
                                        Software Library. 1/3/89

                BETA 0.2        rfs     first full release of front end to
                                        RIT Public Domain Software Library.
                                        4/2/89

====================================================================== */
#include <stdio.h>
#include "stdtyp.h"
#include "listtype.h"
#include "linklist.h"
#include "sym_cfg.h"
#include "sym_type.h"
#include "attribs.h"
#include "trees.h"
#include "sym_tab.h"
#include "chk_type.h"
#include "chk_var.h"
#include "chk_cons.h"
#include "idecl.h"
#include "ra.h"
#include "types.h"
#include "instance.h"
#include "block.h"
#include "imexport.h"
#include "module.h"

static attributes_t * workattr;
static treenode * worktree;
static listhead_t * worklist;
extern treenode * treetop;

extern int yylineno;


%}


%start              CompilationUnit;



%union

%TOKEN          '+'
%TOKEN          '-'
%TOKEN          '*'
%TOKEN          '/'
%TOKEN          ASSIGN      /* := */
%TOKEN          '='
%TOKEN          NOTEQL          /* # <> */
```



```
%TOKEN          '<'
%TOKEN          '>'
%TOKEN          <y_num>         LTE             /* <= */
%TOKEN          <y_num>         GTE             /* >= */
%TOKEN          '('
%TOKEN          ')'
%TOKEN          '['
%TOKEN          ']'
%TOKEN          CATELEM         /* ][ catenated list elements */
%TOKEN          ''
%TOKEN          '^'
%TOKEN          '.'
%TOKEN          RANGE           /* .. */
%TOKEN          AND         /* AND   &                 */
%TOKEN          ARRAY       /* ARRAY                   */
%TOKEN          BEGINTOK    /* BEGIN                   */
%TOKEN          BY          /* BY                      */
%TOKEN          CASE        /* CASE                    */
%TOKEN          CONST       /* CONST                   */
%TOKEN          DEFINITION  /* DEFINITION      */
%TOKEN          DIV         /* DIV                     */
%TOKEN          DO          /* DO                      */
%TOKEN          ELSE        /* ELSE                    */
%TOKEN          ELSEIF      /* ELSIF                   */
%TOKEN          END         /* END                     */
%TOKEN          EXIT        /* EXIT                    */
%TOKEN          EXPORT      /* EXPORT                  */
%TOKEN          FOR         /* FOR                     */
%TOKEN          FORWARD   /* FORWARD                 */
%TOKEN          FROM        /* FROM                    */
%TOKEN          IF          /* IF                      */
%TOKEN          IMPLMN      /* IMPLEMENTATION          */
%TOKEN          IMPORT      /* IMPORT                  */
%TOKEN          IN          /* IN                      */
%TOKEN          LOOP        /* LOOP                    */
%TOKEN          MOD             /* MOD                      */
%TOKEN          MODULE      /* MODULE                  */
%TOKEN          <y_num> NOT         /* NOT                      */
%TOKEN          OF          /* OF                      */
%TOKEN          OR          /* OR                      */
%TOKEN          POINTER     /* POINTER                 */
%TOKEN          PROCEDURE       /* PROCEDURE               */
%TOKEN          QUALIFIED       /* QUALIFIED               */
%TOKEN          RECORD      /* RECORD                  */
%TOKEN          REM             /* REM                 */
%TOKEN          REPEAT      /* REPEAT                  */
%TOKEN          RETURN      /* RETURN                  */
%TOKEN          SET         /* SET                     */
%TOKEN          THEN        /* THEN                    */
%TOKEN          TO          /* TO                      */
%TOKEN          TYPE        /* TYPE                    */
%TOKEN          UNTIL       /* UNTIL                   */
%TOKEN          VAR         /* VAR                     */
%TOKEN          WHILE       /* WHILE                   */
%TOKEN          WITH        /* WITH                    */
%TOKEN          AT_SEA          /* @C                  */
%TOKEN          <y_cp>          IDENT           /* [A-ZA-Z][A-ZA-Z0-9]*  */
```

```
%TOKEN          <y_attr>        NUMBER          /* [0-9][0-9]*           */
%TOKEN          <y_attr>        REAL            /* [0-9]+"."[0-9]*("E"[+|-]?[0-
9]+)?/[^.] */
%TOKEN          ','
%TOKEN          ';'
%TOKEN          ':'
%TOKEN          '|'
%TOKEN          <y_attr>        STRING          /* '[^'\N]*'             |
                                                \"[^"\N]*\"             */
%TOKEN          ERROR       /* .                                       */

%TOKEN          FXN             /*  not really token, just convenient place
                                to define this CONST which needs to be
                                different from all other tokens.  The
                                following tokens are also in this category
                                */
%TOKEN          TYPEASSIGN
%TOKEN          CONSTASSIGN
%TOKEN          PROCCALL
%TOKEN          PROGRAMODULE
%TOKEN          BLOCK
%TOKEN          STATEMENTSEQ
%TOKEN          STATEMENT
%TOKEN          DECLS
%TOKEN          CODE
%TOKEN          STORAGE
%TOKEN          LOCALS
%TOKEN          PARAMS
%TOKEN          CASE
%TOKEN          VARDEC
%TOKEN          MANICONST
%TOKEN          LITERAL
%TOKEN          DESIGNODE
%TOKEN          REFNODE
%TOKEN          ARRAY_LMT
%TOKEN          UMINUS
%TOKEN          UPLUS
%TOKEN          SETUNION
%TOKEN          BOOLEAN
%TOKEN          INTEGER
%TOKEN          CARDINAL
%TOKEN          WORD
%TOKEN          ADDRESS
%TOKEN          LONGINT
%TOKEN          BITSET
%TOKEN          CHAR
%TOKEN          SHORT
%TOKEN          ENUMERATION
%TOKEN          SUBRANGE
%TOKEN          OPAQUE
%TOKEN          FIELD
%TOKEN          ENUMCONST
%TOKEN          ELSEIFLIST

%TOKEN          TEMPORARIES

%TOKEN          LOAO
```



```
%TOKEN          STORE
%TOKEN          UNDEFINED

%type           <y_cp>          idequ, opident


%type           <y_tree>        actualparams, assignment, block, blockdecls,
                                case, caselist, casestatement, CodeModule,
                                nonqualitem,
                                designator, qualident,
                                desident, expression, signedexpr, factor,
                                ftunit,
                                opqualid,
                                casevar, opelsefield,
                                pass2expr, pass2Term, pass2TermList, pass2range,
                                pass2labels, simPass2Exp,
                                pass2Fact, priority,
                                procident, opt_returntype,
                                statement, ifstatement,
                                proccall, casestatement, withstatement,
                                forstatement, whilestatement, loopstatement,
                                repeatstatement, forstep,
                                opcaselse, opelsif, elsestmnt, elseiflist,
                                elsifstmt,
                                procbody, set, sign,
                                simplexpression, sign, term

%type           <y_list>        aparamlist, blockbody,
                                caselablist,
                                desigcomplist, deslist,
                                explist, export, fieldlist, fieldlistseq,
                                fpsect, fpsectlist, formalparams,
                                fpmore,
                                identlist, simpletypelist,
                                l_idlist,
                                formaltypelist, varftlist, opvarftlist,
                                explist,
                                statementseq,
                                importlist, import,
                                modimport,
                                pass2lablist, procprologue,
                                qualimore,
                                nonqualist, ra_elementlist,
                                scopeblock, variant,
                                variantlist

%type           <y_attr>        arraytype, caselabels, codeheader, comnumber,
                                constexpr, consFact, consTerm, consTermList,
                                constrange,
                                enumeration, formaltype,
                                modheader, modident,
                                proceduretype, pointertype,
                                recordtype, pmodule,
                                subrangetype,
                                scopemodbody, settype,
                                simpConstExp
```

```
%type           <y_type>        simpletype, type

%type           <y_num>         addop, DefmodHeader,
                                DefinitionModule, mulop,
                                optatsea, opvartok, relation

%right ASSIGN
%left '|'
%left '<' '>' GLT LTE GTE IN
%left '=' NOTEQL
%left '+' '-' OR
%left '*' '/' AND MOD REM DIV
%left '.'
%right '^'

%%


CompilationUnit :
                        startcompile encloser
                        ;

startcompile    :
                        ;

encloser        :
                        DefinitionModule

                |       CodeModule

                        ;


pmodule         :
                        MODULE IDENT  priority ';'

                        ;

codeheader      :
                        pmodule

                |       IMPLMN pmodule

                        ;

CodeModule      :
                        codeheader progimport block IDENT '.'

                                ;

progimport      :
                        pimportlist

                |
                        ;

pimportlist     :
                        pimportlist pimport
```



```
                |       pimport
                        ;

pimport         :
                        dequalimport importdef
                |       IMPORT defimplist
                        ;

dequalimport    :
                        FROM IDENT IMPORT identlist

                        ;

defimplist      :
                        defitem
                |       defimplist ',' defitem
                        ;

defitem         :
                        defidaction importdef

                        ;

importdef       :
                        ';' DefinitionModule
                        ;

defidaction     :
                        IDENT

                        ;


priority        :
                        '[' constexpr ']'

                |

                        ;



block           :
                        blockdecls blockbody END

                        ;

blockbody       :
                        beginaction statementseq

                |

                        ;

beginaction     :
                        BEGINTOK
                        ;
```

```
blockdecls      :
                  declist
                |
                  ;

declist         :
                  declist declaration
                | declaration
                  ;

designator      :
                  qualident desigcomplist nonqualist
                | qualident desigcomplist
                | qualident nonqualist
                | qualident
                  ;

desigcomplist   :
                  desigcomplist nonqualist deslist
                | nonqualist deslist
                  ;

deslist         :
                  deslist desident
                | desident
                  ;

nonqualist      :
                  nonqualist nonqualitem
                | nonqualitem
                  ;

nonqualitem     :
                  '[' ra_elementlist ']'
                | '^'
                  ;

ra_elementlist  :
                  ra_elementlist CATELEM explist
                | explist
                  ;

desident        :
                  '.' IDENT
                  ;

qualident       :
                  IDENT qualimore
                | IDENT
                  ;

qualimore       :
                  qualimore desident
                | desident
                  ;

signedexpr      :
                  sign simplexpression
                  ;

expression      :
                  signedexpr
                | signedexpr relation signedexpr
                  ;

relation        :
                  '='
                | NOTEQL
                | '<'
                | LTE
                | '>'
                | GTE
                | IN
                  ;

simplexpression :
                  term
                | simplexpression addop term
                  ;

term            :
                  factor
                | term mulop factor
```

```
                ;

factor
        :       comnumber

        |       STRING
        |       designator
        |       designator actualparams

        |       set
        |       '(' expression ')'
        |       NOT factor
                ;

sign    :
                '+'
        |       '-'
        |
                ;

comnumber       :
                NUMBER
        |       REAL

                ;


actualparams    :
                '(' aparamlist ')'

                ;

aparamlist      :
                explist
        |
                ;

explist         :
                explist ',' expression

        |       expression

                ;


declaration     :
                consdef
        |       typedeclaration
        |       vardef
        |       procdecl
        |       scopemodule
                ;

typedeclaration :
                typedecstart optypedecl
                ;
```



```
typedecstart    :
                TYPE
                ;

optypedecl      :
                typedeclist
        |
                ;

typedeclist     :
                typedeclist typedecl
        |       typedecl
                ;

typedecl        :
                idequ type ';'
                        ;

scopemodule     :
                scopemodbody IDENT ';'

                ;


scopeblock      :
                blockdecls blockbody END

                ;


scopemodbody    :
                modheader scopeblock
                ;


modident        :
                MODULE IDENT
                        ;

modheader       :
                modident priority ';' modimport export

                        ;

modimport       :
                importlist
        |
                ;

importlist      :
                importlist import

        |       import

                ;
```

```
import          :
                  FROM IDENT IMPORT identlist ';'

                | IMPORT identlist ';'

                ;

export          :
                  EXPORT identlist ';'

                | EXPORT QUALIFIED identlist ';'

                |

                ;

procdecl        :
                  procprologue procbody IDENT ';'

                |
                  procprologue FORWARD ';'
                ;

procprologue    :
                  procident formalparams opt_returntype ';'

                ;

opt_returntype  :
                  ':' qualident

                |

                ;

procbody        :
                  block

                ;

procident       :
                  PROCEDURE IDENT

                ;

formalparams    :
                  '(' fpsectlist ')'

                |
```



```
                ;

fpsectlist      :
                  fpmore

                |

                ;

fpmore          :
                  fpmore ';' fpsect

                | fpsect
                ;

fpsect          :
                  opvartok identlist ':' formaltype

                ;

opvartok        :
                  VAR

                |

                ;

varbegin        :
                  VAR
                ;

vardef          :
                  varbegin vardeclist
                ;

vardeclist      :
                  vardecmore

                |

                ;

vardecmore      :
                  vardecmore vardecl
                | vardecl
                ;

vardecl         :
                  l_idlist type ';'

                ;

l_idlist        :
                  identlist ':'

                ;

consdefstart    :
                  CONST
```

```
                ;

consdef         :
                consdefstart optconsdeclist
                ;

optconsdeclist  :
                consdecllist
                |
                ;

consdecllist    :
                consdecllist consdecl
                |       consdecl
                ;

consdecl        :
                idequ constexpr ';'
                ;

idequ           :
                IDENT '='
                ;

constexpr       :
                simpConstExp
                |       simpConstExp relation simpConstExp
                ;

simpConstExp    :
                sign consTermList
                ;

consTermList    :
                consTerm
                |       consTermList addop consTerm
                ;

consTerm        :
                consFact
                |       consTerm mulop consFact
                ;

consFact        :
                qualident
                |       comnumber
                |       STRING
                |       set
                |       '(' constexpr ')'
```



```
                |       NOT consFact
                ;

addop           :
                '+'
                |       '-'
                |       OR
                ;

mulop           :
                '*'
                |       '/'
                |       DIV
                |       MOD
                |       AND
                |       REM
                ;

set             :
                opqualid elementlist
                ;

opqualid        :
                qualident
                |
                ;

elementlist     :
                ''
                ;

elemlist        :
                elemlist ',' element
                |       element
                ;

element         :
                constexpr
                |       constexpr RANGE constexpr
                ;

type            :
                simpletype
                |       arraytype
                |       recordtype
                |       settype
                |       pointertype
                |       proceduretype
                ;

simpletype      :
                qualident
                |       enumeration
                |       subrangetype
```

```
                ;

enumeration     :
                '(' identlist ')'

                ;

identlist       :
                identlist ',' IDENT

                |       IDENT

                ;

subrangetype    :
                opqualid '[' constexpr RANGE constexpr ']'

                ;

arraytype       :
                ARRAY simpletypelist OF type

                ;

simpletypelist  :
                simpletypelist ',' simpletype

                |       simpletype

                ;

recordtype      :
                RECORD fieldlistseq  END

                ;

fieldlistseq    :
                fieldlistseq ';' fieldlist

                |       fieldlist
                ;

fieldlist       :
                l_idlist type

                |       casevar

                        ;

casevar         :       CASE opident ':' qualident OF variantlist
                                opelsefield END

                ;

opident         :
```



```
                IDENT

                |
                ;

opelsefield     :
                ELSE fieldlistseq

                |
                ;

variantlist     :
                variantlist '|' variant

                |       variant
                ;

variant         :
                caselablist ':' fieldlistseq

                |
                ;

caselablist     :
                caselablist ',' caselabels

                |       caselabels

                ;

caselabels      :
                constexpr constrange
                ;

constrange      :
                RANGE constexpr

                |
                ;

pass2lablist    :
                pass2labels

                |       pass2lablist ',' pass2labels

                ;

pass2labels     :
                pass2expr pass2range

                ;

pass2range      :
```

```
                RANGE pass2expr

                ;

pass2expr       :

                simPass2Exp
                |       simPass2Exp relation simPass2Exp
                ;

simPass2Exp     :

                sign pass2TermList
                ;

pass2TermList   :

                pass2Term
                |       pass2TermList addop pass2Term
                ;

pass2Term       :

                pass2Fact

                |       pass2Term mulop pass2Fact

                ;

pass2Fact       :

                qualident
                |       comnumber

                |       STRING

                |       set
                |       '(' pass2expr ')'

                |       NOT pass2Fact

                ;

settype         :

                SET OF simpletype

                ;

pointertype     :

                POINTER TO type

                ;

proceduretype   :

                PROCEDURE
                |       PROCEDURE formaltypelist

                ;

formaltypelist  :

                '(' opvarftlist ')'
                |       '(' opvarftlist ')' ':' qualident
```



```
                ;

opvarftlist     :

                varftlist
                |

                ;

varftlist       :

                varftlist ',' ftunit

                |       ftunit

                ;

ftunit          :

                opvartok formaltype

                ;

formaltype      :

                qualident
                |       ARRAY OF qualident

                ;

statementseq    :

                statementseq ';' statement

                |       statement

                ;

statement       :

                assignment
                |       proccall

                |       ifstatement
                |       casestatement
                |       whilestatement
                |       repeatstatement
                |       forstatement
                |       loopstatement
                |       withstatement
                |       EXIT

                |       RETURN

                |       RETURN expression

                |

                ;
```

```
withstatement   :
                WITH designator DO statementseq END
                ;
loopstatement   :
                LOOP statementseq END
                ;
forstatement    :
                FOR IDENT ASSIGN expression TO expression forstep DO stat
ementseq END
                ;
forstep         :
                BY pass2expr
                |
                ;
repeatstatement :
                REPEAT statementseq UNTIL expression
                ;
whilestatement  :
                WHILE expression DO statementseq END
                ;
casestatement   :
                CASE expression OF caselist opcaselse END
                ;
opcaselse       :
                ELSE statementseq
                |
                ;
caselist        :
                caselist '|' case
                |       case
                ;
case            :
                pass2lablist ':' statementseq
                |
                ;
ifstatement     :
                IF expression THEN statementseq opelsif elsestmnt END
```



```
                ;
opelsif :
                elseiflist
                |
                ;
elsifstmt       :
                ELSEIF expression THEN statementseq
                ;
elseiflist      :
                elseiflist elsifstmt
                |       elsifstmt
                ;
elsestmnt       :
                ELSE statementseq
                |
                ;
proccall        :
                designator
                |       designator actualparams
                ;
assignment      :
                designator ASSIGN expression
                ;
```

```
DefinitionModule:
                DefmodHeader defbody END IDENT  .'
                ;
defbody         :
                progimport optdefinlist
                ;

optdefinlist    :
                definlist
                |
                ;

DefmodHeader    :
                DEFINITION MODULE IDENT optatsea ';'
                ;
optatsea        :
                AT_SEA
                |
                ;

definlist       :
                definlist definition
                |       definition
                ;
definition      :
                consdef
                |       typedef
                |       vardef
                |       PROCEDURE IDENT formalparams ';'
                ;
typedef         :
                TYPE optypedeflist
                ;
optypedeflist   :
                typedeflist
                |
                ;

typedeflist     :
                typedeflist typeassign
                |       typeassign
                ;
```



```
typeassign      :
                IDENT ';'
                |       typedecl ';'
                ;
%%
```

APPENDIX C

APPENDIX C SYMBOL TABLE DESIGN DOCUMENT

I. Introduction

A symbol table for a language supporting closed scopes may roughly be broken down into 2 major constituents. These constituents are: 1) a set of code and a commensurate "volatile" data structures which handle the scoping or visibility aspects of the parsing activities, and 2) a set of code and "persistent" data structures which manage type, size, storage location information, etc. The design and construction of the first phase of a Modula-2 symbol handling system differs from the classical approach in the need to manage closed scopes. In addition, a Modula symbol table handler is greatly complicated by the need to handle mutual recursion in open scopes and the possible use of previously undeclared variables which will at some future (lexical) time be exported by a local module. The second phase of symbol handling related to types, sizes, etc. should, for the most part, yield to classical methods

This document will concentrate on the scope handling aspect of the symbol table module because it is here that the most effort and new (not widely available in the literature) design must be applied. The paper by Graham, Joy, and Robine is the basis for this design. Included here are a data diagram and word descriptions of the routines which are required to handle the scope transitions and associated symbol table maintenance.

II. Scope Handling - during the first pass of the compiler, types are resolved and symbol entries are bound to their usages in the abstract syntax tree. The points in the parse of particular interest here are as follows:

1. Module Entry
2. Module Exit
3. Procedure Entry
4. Procedure Exit
5. Import Declaration
6. Export Declaration (this implicitly includes ALL declartions within a Definitions Module)
7. Symbol Declaration
8. Symbol Use

YACC will provide the framework from which the semantic actions needed to process the above events will be called.

A. Data Structures - the following is a brief discussion of data structures required to handle scoping of Modula-2. See figure 4 for a schematic diagram of symbol table structure interactions.

The symbol table data structures are patterned after the classical model of a hash table pointing into a stack symbol table. The symbol stack is segmented by the expected "block table" which delineates "stack frames" corresponding to scopes (both open and closed). Several additions are made to this structure to accomodate the special needs of Modula. The block table has 2 new fields, **1) export_count** and **2) enclosing_scope.** The stack symbol table blocks have added 4 new fields, **1) visiLimit, 2) readOnly, 3) idClass,** and **4) undecl.**

The uses of these fields are:

1. **Block Table**

   a. **export count** - this field is used to keep track of the number of original exports not previously exported through nesting. It is used at scope exit time to adjust the symbol stack frame pointer in the block table. Exported symbols must "live" in the environment above the one in which they were exported. By adjusting the stack frame the previous environment may "gobble" these exported symbols from the environment which is about to die.

   b. **enclosing_scope** - This field identifies the immediately enclosing CLOSED scope. It is required to allow the mixing of closed and open scopes in the same symbol table mechanism. If the visibility limit of a given symbol is greater than or equal to the enclosing scope, the symbol is "alive" in the current environment.

2. **Symbol Stack Block**

   a. **visiLimit** - the visibility limit is the maximum depth of nesting in which a given symbol is visible. This value is altered on closed scope exit and at import time.

   b. **readOnly** - In Modula, exported variables must be exported read only. The read only field indicates the nesting level at which a variable was LAST imported readonly. Note that in nested modules (where access to a symbol is not through an export), variables imported are not imported read only. When a symbol is installed in the symbol table, the value of this field is set to the "pervasive" (all encompassing) level.

c. **idClass** - When a symbol is declared "Export" for the first time (in a nesting sequence) the id class corresponding to that symbol is set to "isExport". When the scope in which the symbol was first exported is exited, the symbol is set to "isVar", "isConst", or whatever attribute applies. This signals the parser to not reject the entries when they are finally declared.

d. **undecl** - In lexically adjacent scopes, variables and procedure names (symbols in general) may be used before they are declared. The usage of an undeclared symbol will trigger its creation just as in the case of exportation, but will not cause the creation of a persistent symbol block. It would be possible to use the idClass field in conjunction with export count or to use the pointer to persistent block (test for NULL) but to eschew confusion, a separate field is allocated.

Note that the functions of b, c and d have been largely superceded by equivalent fields in the persistent attribute block. These 3 variables are more useful on the stack when used with a single pass compiler. They are kept here for possible future performance enhancements.

B. **Semantic Actions Associated with Scope Control** - below is a brief (and incomplete) word description of the special processing associated with scope control in Modula using the data structures described above. The list below is organized by the points in the parse at which the important actions take place (e.g. Import declaration, Module exit, etc.)

1. **Symbol Import**

   a. lookup symbol

b. if the symbol is not found

b.1. create the UDECL symbol

c. move to front of hash chain for this bucket

d. set visiLimit to the current nesting level

NOTE: Because imported symbols may be first encountered in the IMPORT statement (forward references) it is critical that the IMPORT statement **precede** the EXPORT statement. The closed scope entry semantic action must be performed after the IMPORT statement. This requires that the IMPORT semantic action set the visibility limit to current_scope + 1 (the next scope about to be openned by the open scope entry semantic action). Graham [GRAH 79] et al mention only that undeclared imports can be handled but do not provide details of the mechanism. The specification for Modula 2 fits nicely with this constraint on the order of the IMPORT/EXPORT and one gains new respect for the designer of the language after implementing this detail.

2. **Symbol Export** (both explicit and as part of definition module symbol list)

a. lookup symbol

b. if not found install as "isExport"

c. else if idClass != "isExport" /* if previously declared */

c.1. error exit

d. bump export_count

3. **Procedure Entry**

   a. bump scope top

   b. set pointer to stack frame start in new block to current stack top

   c. copy "enclosing_scope" value from previous frame to our frame

Note: The purpose fo c. above is to make sure that a surrounding module has not exported the symbol and thereby caused the special case of an extant but undefined EXPORT variable. See figure C.1

```
MODULE Outer;
  EXPORT foo;
  MODULE Inner;
    EXPORT foo;
    VAR foo;
```

Figure C.1: Multiple Export example.

The first (legal) export of "foo" will install an undefined symbol in the symbol table. The second export of "foo" (also legal) must check the attibute record to ensure it is undefined and therefore legal. By using the "undecl" flag in the symbol stack (2.d) a link traversal could be avoided at the expense of an extra variable.

4. **Module Entry**

   a. bump scope top

   b. set pointer to stack frame start in new block to current stack top

   c. set "enclosing scope" to scope top

5. **Procedure Exit**

   a. release all vars in current frame (note problem in forward decls)

   b. set current stack top to frame start of current frame

   c. decrement scope top

6. **Module Exit** - this is the most expensive transition in the symbol table manipulation.

   a. release local symbols not exported (from stack top to current frame start + export count)

   b. adjust visibility limit on imported symbols by searching the hash chains for symbols with visiLimit = scope top.

   c. check all exported symbols (those remaining in the frame) to see that they have been declared (attribute of idClass is NOT "isExport")

   d. set exported vars to read only (set readonly = TRUE). Note that this is tricky in that, in a 2 pass system, what will REALLY happen is that a "linking" node will be placed in the syntax tree which indicates that all subtrees are readonly (in this path). Thus separate paths will have separate linking nodes which allow subtrees to be accessed in different ways.

      It may be possible to determine errors of assignment using readonly variables as lvals in the first pass but it is not clear that this would be less complicated than inserting a readonly node in the tree structure.

   e. If anything remains in the "UDECL" bag, flag as "used but undefined errors"

   f. decrement scope top

   g. Note that at this point, the symbol stack top will be pointing to the first free entry above the exported symbols and nothing remains to be done. The exported symbols now rest safely within

the frame of the environment surrounding the module which has just died.

III. Undeclared variables which will be exported by some local module.

A. Because symbols may bind to thcir occurencc(s) in the abstract syntax tree before the symbol is declared, the proposed "semiclassical" symbol table mechanism works well in this case. Each time an undeclared symbol is encountered in the parse, the symbol handling mechanism will fill in as much of the persistent data structure as possible. This will allow some consistency checking even though the declaration is to follow. The second pass will make the final check to see if usage is consistent.

NOTES:

1. Pervasive identifiers (CARDINAL, etc.) :
   a. #define PERVASIVE_LEVEL MAX_SCOPE_DEPTH
   b. check level = Pervasive Level & ignore symbol if true on import.
   c. scope entry -- if level > max scope, fatal error.
   d. scope exit -- do NOT reassign visilimit for pervasive level symbols!
   e. general - this technique for pervasive identifiers implies symbols not in symbol table trigger FULL search of conflict chains. This is because one doesn't know if one is looking at a pervasive (which may be at end of chain) or new symbol which doesn't exist! Pervasives don't get sorted to front of chains. This implies that techinque speeds up only searches for extant symbols. Extant symbols should constitute the majority of symbols in non-pathological programs.

APPENDIX D

```
#define SYMTYPE 0 /* so system knows this include file is present */

#ifndef LISTTYPE
#include "listtype.h"
#endif



/* ========================  STRUCTURES  ============================== */


typedef struct lineblock_s {
        struct que_block links;
        int lineno;
        } lineblock_t;



typedef struct linelist_s {
        listhead_t linehead;
        int lineno;
        } linelist_t;



struct dimlist_s {      /* dimension may be expressed as simpletype:
                        qualident;
                        enumeration;
                        subrange;       */
  listhead_t    head;
  short         lowerb;
  short         upperb;
  };


typedef struct sym_tab_block_s {
        int     mod_lexval;
        int     scopeClass;
        short   forward;
        int     visiLimit;
        int     unDecl;
        int     readonly;       /* level at which sym becomes readonly. init to p
ervasive */

        struct hash_block_s *hash_ref; /* link back to conflict block to ease rem
oval */


        char    * m_sym;                /* cast to char because treenode is udef
*/
        } sym_tab_block_t;


typedef struct hash_block_s {
        struct  que_block conflist;
```



```
        short   table_ndx;
        char    * name_ptr;
        } hash_block_t, *hash_block_ptr;


typedef struct hash_bucket_s {
        listhead_t confl_head;
        } hash_bucket_t;

/* ==============>>>  THIS STRUCT FOR NAME TABLE OPEN CHAIN  <<<======= */

typedef struct name_block_s {
        qblk_t  list;
        char    * name_ptr;
        } name_block_t;



typedef struct scope_block_s {
        int export_count;               /* number of objects exported by this
                                           level    */
        unsigned long current_offset;   /* displacement into storage base for
                                           next symbol */
        unsigned long param_offset;     /* ditto for parameters */
        int sym_tab_ndx;                /* pointer to beginning of this block
                                           in symbol table */
        int encl_close_scope;           /* the enclosing closed scope (all
                                           scopes not proc) */
        int priority;                   /* priority set (or inherited) by
                                           current scope */
        listhead_t dequal_list;         /* list of attributes which have been
                                           dequalified from modules .. look
                                           at enclosing closed scope for these
                                           symbols */

        listhead_t bag;
        } scope_block_t;
```

APPENDIX E

```
/* ========================================================================

FILE NAME :     trees.h

PURPOSE :       to define structures, constants, etc. which are used in
                the tree structure of the abstract syntax tree.

HISTORY :       11/23/87        rfs     original version



/* ================================================================== */

#ifndef TREEDEF
#define TREEDEF

#ifndef SYMTYPE
#include "sym_type.h"
#endif

# ifndef ATTRIBUTES
# include "attribs.h"
# endif

# ifndef REGDEF
# include "regdata.h"
# endif

/* ######################### CONSTANT DEFS ############################## */


#define TREE 0
#define LEAF ( TREE + 1 )
#define DESIG ( LEAF + 1 )
#define TEMP ( DESIG + 1 )

#define LEFT 0
#define RIGHT LEFT + 1



typedef struct interior_s {
  short         nodetype;
  struct treenode_t * symptr;   /* ptr to id if extant */
  short         Commutative;
  char          * spare_ptr;            /* a ptr for generic use */
  } interior_t;

typedef struct desig_s {
  int                   desigtype;              /* name, array or dereference */
  struct treenode_t     * symptr;
  int                   use_level;
  int                   form;
  int                   class;
```



```
  typedesc_t            * basetype;
  int                   readonly;
  } desig_t;

typedef union treeu {
  attributes_t  * leaf;
  interior_t    node;
  desig_t       desig;
  storage_t     store;
  } treebody;

typedef struct treenode_t {
  qblk_t   list;        /* forward & backward ptrs.... */
 /* ===>> ONLY NEED THE NEXT 2 FOR TREE TYPE TREENODES (FIX NEXT TIME) << === */
  struct   treenode_t *leftleg;
  struct   treenode_t *rightleg;

  struct   treenode_t * uplink; /* reserved for threading the tree */
  short    treetype; /* tree or leaf */
  short    DataRegCount;        /* used in code gen */
  short    AddrRegCount;        /* used in code gen */
  short    branch;      /* LEFT or RIGHT */
  treebody treedata;
  } treenode;



/* ########################  MACRO DEFS ######################### */


#define ATTRLEAF(foo) (((treenode *)foo)->treedata.leaf)

#define CONSTLEAF(t) ( ATTRLEAF (t)->idclass.constval )
#define VARLEAF(t) ( ATTRLEAF (t)->idclass.varaddress )
#define MODLEAF(t) ( ATTRLEAF (t)->idclass.package )
#define PROCLEAF(t) ( ATTRLEAF ( t )->idclass.procedure )

#define NODE(foo) (((treenode *)foo)->treedata.node)

#define DESIGNATE(foo) (((treenode *)foo)->treedata.desig)

#define REGSTORE(foo)  (((treenode *)foo)->treedata.store)
#define REGDATA(foo)  ( REGSTORE ( foo ).storedata.regdata)
#define TEMPSTORE(foo) ( REGSTORE ( foo ).storedata.tempdata )


#define BLOCK_DCL(foo) (((treenode *)foo)->leftleg)
   /* ______  macros for navigation from decls to leaf _______ */

#define DCL_PROC(foo) (((treenode *)foo)->leftleg)

#define DCL_STMT(foo) (((treenode *)foo)->rightleg)

#define DCL_TEMP(foo) (DCL_STMT(foo)->leftleg)

/* ########################  PROC DECLS ######################### */
```

```
short
init_tree ();

treenode *
mknode ();

treenode *
mkleaf();

treenode *
make_uopnode ();

treenode *
rlink ();

treenode *
llink ();

short
depth_first();

short
preorder ();

treenode *
mkdesig ();

#endif
```

APPENDIX F

```
/* ========================================================================

FILE NAME :     attribs.h

HISTORY :       06/??/87        rfs     original version
                                        abstracted from Fischer & LeBlanc

                                        see also Waite & Goos

                07/26/88        rfs     adding proc & package stuff for code
                                        gen

/* ======================================================================== */

#define ATTRIBUTES 0

/* ++++++++++++++++++++  CONST DECLS ++++++++++++++++++ */

# define STATIC 0
# define AUTOMATIC STATIC+1
# define EXTERNAL AUTOMATIC+1


typedef struct array_s {
  struct typedesc_s     * indextype;
  int                   lwrbnd;
  int                   uprbnd;         /* should these be attrs? */
  struct typedesc_s     * elementref;
  } array_t;

typedef struct variantdesc_s {
  int                   choiceval;
  struct attributes_s   * tag;
  struct attributes_s   * fieldlist;
  struct variantdesc_s  * innervar;
  struct variantdesc_s  * nextvar;
  } variantdesc_t;

typedef struct record_s {
  struct attributes_s * fieldlist;
  struct variantdesc_s * variantlist;
  } record_t;

typedef struct subtype_s {
  struct typedesc_s * basetype;
  int   lwrbnd, uprbnd;
  } subtype_t;

typedef struct settype_s {
  struct typedesc_s     * basetype;
  } settype_t;

typedef struct scopedata_s {
```



```
  int           nesting_level;  /* val in environ BEFORE open scope enter */
  char          * startlab;
  char          * endlab;
  int           procnum;
  char          * varbytes;
  char          * tempbytes;
  short         regmask;
  } scopedata_t;

typedef struct export_s {
  char                  * moduleptr;
  unsigned long         export_offset;
  struct attributes_s   * this_export;
  struct attributes_s   * next_export;
  } export_t;

typedef struct param_t {
  int                   paramlevel;
  unsigned long         paramoffset;
  int                   mode;
  struct attributes_s   * nextparam;
  } param_t;


typedef struct procedure_s {
  scopedata_t   scopedata;
  unsigned long stackneeded;
  int           priority;
  listhead_t    * paramlist;
  char          * returntype;
  int           bodydeclared;
  } procedure_t;

typedef struct proctype_s {     /* this is for procedure types */
  listhead_t    * paramlist;
  char          * returntype;
  } proctype_t;                 /* this is type NOT attribute as is above */

typedef struct opaque_s {
  struct typedesc_s * typeref;
  struct attributes_s * attref;
  } opaque_t;

typedef struct enumeration_s {
  struct attributes_s * firstconst;
  int                   uprbnd;
  } enumeration_t;

typedef struct enumconst_s { /* CONSTANT LIST */
  short enumval;
  struct attributes_s * nextconst;
  } enumconst_t;

typedef struct errortype_s {
  char  * errorname;
  } errortype_t;
```

```
typedef struct field_s {
  char * field_offset;
  struct attributes_s * next_field;
  } field_t;

typedef struct address_s {
  int varlevel;         /* nesting level */
  char * varoffset;
  short indirect;
  short readonly;
  short storage_type;
  short allocate;       /* if TRUE, allocate storage for this (statics only?)
  char  * known_as;
  listhead_t reglist;   /* list of registers containing this variable or its addr
. */
  } address_t;


/* ------------------>> NOTE: <<--------------------
selectors "valuekind" in valuetype_t union constval below are:

        CHAR
        INTEGER
        CARDINAL
        LONGINT
        REAL

*/

  typedef union constval_s {
    char byteval;
    short shortval;
    int intval;
    unsigned int cardval;
    long longval;
    float floatval;
    char * stringval; /* ? */
    /* others */
    } constval_u;

typedef struct valuetype_s {
  short valuekind;
  constval_u constval;
  } valuetype_t;

typedef struct pkgname_s {
  listhead_t * export_syms;
  } pkgname_t;

typedef struct progmod_s {
  listhead_t    * static_list;  /* list of attributes requiring static alloc */
  } progmod_t;

typedef struct localmod_s {
  int           import_count;
  int           export_count;
  listhead_t    qualxprt;
```



```
  } localmod_t;

/* ------------------>> NOTE: <<--------------------
selectors "class" in attributes_t union idclass_t below are:

        CONSTTOK
        ENUMCONST
        VARTOK
        MODULETOK

        TYPETOK         produces NULL class record

        PROCTOK

        FIELD
        PROGRAMODULE
        UNDEFINED

*/

typedef union idclass_s {
  valuetype_t   constval;
  enumconst_t   enumconst;
  /* export_t   exported; not needed, just pure list attached to pkg */
  field_t       fieldef;
  pkgname_t     package;
  param_t       parameter;
  procedure_t   procedure;
  progmod_t     program;
  address_t     varaddress;
  } idclass_t;

typedef struct attributes_s {
  char                  * name;
  char                  * label;
  struct typedesc_s     * idtype;
  listhead_t            * register_list;/* list of registers refering or
                                          containg this node */
  listhead_t            withlist;       /* list of resolution candidates */
  idclass_t             idclass;
  short                 class;          /* selector of id class */
  short                 pervasive;      /* needed for import export */
  short                 fixed;
  short                 forward;
  } attributes_t;

/* ------------------>> NOTE: <<--------------------
selectors "form" in typedesc union typeform below are:

        CHAR
        INTEGER
        CARDINAL
        LONGINT
        BOOLEAN
        REAL

        ENUMERATION
```

```
        BITSET
        ENUMERATION
        ARRAYTOK
        RECORDTOK
        SUBTYPE
        POINTERTOK
        OPAQUE
        ERRORTOK

*/

typedef struct typedesc_s {
  char * size;
  short form;
  short fixed;
  union {
    /* CHAR, INTEGER, CARDINAL, BOOLEAN, REAL, BITSET   all null */
    enumeration_t enumeration;          /* ENUMERATION */
    array_t             array;          /* ARRAY */
    record_t            record;         /* RECORD */
    subtype_t           subtype;        /* SUBTYPE */
    struct typedesc_s   * pointer;      /* POINTER */
    opaque_t            opaque;         /* OPAQUE */
    proctype_t          proctype;
    settype_t           set;
    errortype_t         errortype;

    /* others later */
    } typeform; /* end of UNION typeform */
  } typedesc_t;


/* ++++++++++++++++++ == >> MACROS FOR ATTRIBUTES << == ++++++++++++++++++ */

# define  CONSTATTR(t) (((attributes_t * )t)->idclass.constval)
# define  ADDRATTR(t) (((attributes_t * )t)->idclass.varaddress)
# define  PROCATTR(t) (((attributes_t * )t)->idclass.procedure)
# define  EXPATTR(t) (((attributes_t * )t)->idclass.exported)
# define  PKGATTR(t) (((attributes_t * )t)->idclass.package)
# define  ENUMEMBER(t) (((attributes_t *)t)->idclass.enumconst)
# define  FIELDATTR(t)  (((attributes_t *)t)->idclass.fieldef)

# define  TYPEFORM(t) (((typedesc_t *)t)->typeform)
# define  ENUMLIST(t) (((typedesc_t *)t)->typeform.enumeration.firstconst)
# define  ENUMTYPE(t) (((typedesc_t *)t)->typeform.enumeration)
# define  SUBTYPE(t)  (((typedesc_t *)t)->typeform.subtype)
# define  RATYPE(t)  (((typedesc_t *)t)->typeform.array)
# define  RECTYPE(t)  (((typedesc_t *)t)->typeform.record)
# define  PTRTYPE(t)  (TYPEFORM(t).pointer)
# define  SETYPE(t)  (TYPEFORM(t).set);
# define  PROCTYPE(t)  (TYPEFORM(t).proctype);

# define  CONSTVAL(t) (CONSTATTR(t).constval)
```

APPENDIX G

'C' Language Interface

Modula 2 differs from 'C' in that it supports nested scopes. This is important if one wishes to call 'C' language routines from Modula 2 or vice versa. A language which supports nested scopes must use a static link at run time for uplevel references. Because 'C' does not allow nested scopes there exists a fundamental difference in the runtime environment of the two languages. If the calling convention of the Modula 2 implementation is similar in all other respects to the 'C' calling convention, then a simple "trick" may be used to link the two languages.

In 'C' parameters are pushed in the reverse order in which they appear in the function parameter list. If, in Modula 2, the static link is the last item pushed onto the stack, then it *could* correspond to the first parameter in a 'C' language routine [LUTZ 88]. Using this convention, any 'C' functions *called* by Modula 2 must have a dummy first parameter which is the size of an address (char * would do nicely). Any 'C' functions which *call* Modula 2 must also have a dummy first parameter. Note that in either case, the dummy variable is meaningless. Even though all 'C' routines live at the top nesting level, they cannot "know" the static link of that level. Modula 2 modules may only export routines which live at the same nesting level as 'C' functions. Thus 'C' may only call Modula 2 routines which live at the same static level as the calling routine. "Uplevel" references have no meaning to the called Modula 2 routine because there is no enclosing static level which supports a stack. All variables in the implied static enclosing environment are named global and static references. One assumes that any static link would not be used even if it were available.

All return values are passed in the first data register ("d0" in MC68000 UNIX assembler). This is the standard convention for 'C' and Modula 2 in this environment.

Examples of 'C' functions calling Modula 2 routines and Modula 2 routines calling 'C' functions are given below.

'C' Calling Modula 2

Calling Function is 'C':

```
cmain( )
{
  char * slink;
  in   outdata;
  /* other data... */

  outdata = modproc ( slink, other_args... );
}
```

The Modula 2 function used above is:

```
modproc ( other_args... ):INTEGER;
```

Modula 2 Calling 'C':

Calling Routine is Modula 2:

```
PROCEDURE modproc ( somevar : SomeType, othervars :
OtherTypes )
  BEGIN
    cproc ( somevar ); (* if "somevar" is array must be VAR *)
  END modproc;
```

Definition module required so that Modula 2 may call this 'C' function:

```
DEFINITION MODULE c_application;
  PROCEDURE cproc ( somevar : SomeType );
END c_application.
```

'C' Function Called by Modula 2:

```
typedef struct sometype_s {
  /* definition of SomeType */
  } sometype;

void
cproc( slink, somevar )
char * slink; /* throw away this var or use for scratch */
sometype somevar;
{
   .
   .
   .
} /* end of void FUNCTION cproc */
```