Modula-2

Language Description
And
Compiler

Stephen J. Allan
©2010

CS 5300
Fall 2010
Modula-2

Modula-2 is a programming language developed by Niklaus Wirth. The description that follows describes a subset of the language that is implemented in this project. A full introduction to the language can be found in *Programming in Modula-2* by Wirth, published by Springer-Verlag.

Section 0 contains the description of the lexical conventions used in this chapter and a description of the tokens of Modula-2. Semantic descriptions of the statements are given in Section 0. Section 0 contains the grammar for the subset of Modula-2 to be implemented. Finally, Section 0 contains miscellaneous information needed about the language.

**Lexical Conventions**
The symbols of the Modula-2 vocabulary are divided into the following classes: identifiers, numbers, strings, characters, operators and delimiters, and comments.

**Notation**
In the BNF presentation of the syntax, = can be read as “is defined as.” | indicates an alternative while square brackets ([ ]) denote optional material. Braces ({ }) indicate that the enclosed syntactic unit is repeated zero or more times. Names appearing totally in upper-case (e.g., `BEGIN`) represent reserved words that must appear. Symbols appearing between quotes (e.g., “;”) represent symbols that must appear in the syntax without the quotes.

**Identifiers**
Identifiers are sequences of letters and digits. The first character must be a letter.

\[
\text{Ident} = \text{letter} \{ \text{letter} | \text{digit} \}
\]

Upper and lower case letters are considered as distinct. Thus `hello` and `Hello` are separate identifiers.
Numbers
Numbers are integers denoted by sequences of digits. Numbers must not include any spaces.

If followed by the letter B, integers are taken as octal numbers, or as hexadecimal numbers if followed by the letter H.

```
Number   = integer
integer   = digit {digit}
            |   octalDigit {octalDigit} "B"
            |   digit {hexDigit} "H"
octalDigit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7"
digit      = octalDigit | "8" | "9"
hexDigit   = digit | "A" | "B" | "C" | "D" | "E" | "F"
```

Strings
Strings are sequences of any characters enclosed in quote marks. In order that the closing quote is recognized unambiguously, the string itself evidently cannot contain a quote mark. To allow strings with quote marks, a string may be enclosed within apostrophes instead of quote marks. In this case, however, the string must not contain apostrophes. A string cannot contain the new line character (decimal 10).

```
String = "" { character } "" | "" { character } ""
```

Characters
Character constants look like strings but contain only one character between the quotes. In order to be able to denote non-printable characters, their octal ordinal number followed by C can be used. For example, 14C is a value of type CHAR denoting the control character form feed with the ordinal number 14B.

```
Character = "" character "" | "" character ""
```
Operators and Delimiters

Operators and Delimiters are either special characters or reserved words. These latter are written in upper case and cannot be used as identifiers.

The operators and delimiters composed of special characters are:

+  addition, set union
-  subtraction, set difference
*  multiplication, set intersection
/  division, symmetric set difference
:=  assignment
&  logical AND
~  logical NOT
=  equal
# <>  unequal (two different symbols)
<  less than
>  greater than
<=  less than or equal
>=  greater than or equal
( )  parentheses
[ ]  index brackets
{}  set braces
(* *)  comment brackets
^  dereference operator
, ; : .. |  punctuation symbols
Operators are defined by the following classes:

- **UnOperator** = "+" | "-" | NOT | "~"
- **MulOperator** = "*" | "/" | DIV | MOD | AND | "&"
- **AddOperator** = "+" | "-" | OR
- **Relation** = ";=" | ";#" | ";<=" | ";<" | ";=" | ";<" | ";="}

The operators AND and & both represent the *logical and* operation.

The UnOperator's have the highest precedence while Relation's have the lowest. Parentheses can be used to override the precedence. Any operators appearing in the same expression and having the same precedence are evaluated left to right.

The reserved words are enumerated in the following list.

<table>
<thead>
<tr>
<th>AND</th>
<th>ELSE</th>
<th>IMPORT</th>
<th>PROCEDURE</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARRAY</td>
<td>ELSIF</td>
<td>IN</td>
<td>QUALIFIED</td>
<td>UNTIL</td>
</tr>
<tr>
<td>BEGIN</td>
<td>END</td>
<td>LOOP</td>
<td>READ</td>
<td>VAR</td>
</tr>
<tr>
<td>BY</td>
<td>EXIT</td>
<td>MOD</td>
<td>RECORD</td>
<td>WHILE</td>
</tr>
<tr>
<td>CASE</td>
<td>EXPORT</td>
<td>MODULE</td>
<td>REPEAT</td>
<td>WITH</td>
</tr>
<tr>
<td>CONST</td>
<td>FOR</td>
<td>NOT</td>
<td>RETURN</td>
<td>WRITE</td>
</tr>
<tr>
<td>DEFINITION</td>
<td>FROM</td>
<td>OF</td>
<td>SET</td>
<td>WRITELN</td>
</tr>
<tr>
<td>DIV</td>
<td>IF</td>
<td>OR</td>
<td>THEN</td>
<td></td>
</tr>
<tr>
<td>DO</td>
<td>IMPLEMENTATION</td>
<td>POINTER</td>
<td>TO</td>
<td></td>
</tr>
</tbody>
</table>

**Comments**

A comment may be inserted between any two symbols. They are arbitrary sequences of characters enclosed in the comment brackets (* and *). Comments are skipped by the compiler and serve as additional information to the human reader. They may also serve to signal instructions (options) to the compiler.
**Semantics**

A brief description of the semantics for the different statements of Modula-2 is given below.

**Assignment Statement**
The assignment statement is the most elementary statement. Its semantics are:

1. Evaluate the left-hand-side.
2. Evaluate the right-hand-side.
3. Replace the value of the left-hand-side with the value of the right-hand-side.

**Control Statement**
There are two categories of statements discussed in this section: repetitive and conditional.

**Repetitive Statements**
The WHILE statement consists of an *expression* and a *sequence of statements*. The corresponding actions are:

1. Evaluate the condition that takes the form of an expression yielding the value TRUE or FALSE.
2. If the value is TRUE, execute the statement sequence and then repeat the previous step; if the value is FALSE, terminate the loop.

The above rules imply that the body of the WHILE statement may not be executed.

The REPEAT statement consists of a *sequence of statements* followed by an *expression*. The corresponding actions are:

1. Execute the statement sequence.
2. Evaluate the condition that takes the form of an expression yielding the value TRUE or FALSE. If the value is TRUE, the loop is terminated; if the value is FALSE, repeat the previous step.

The FOR statement is similar to the Pascal’s, with the exception of the optional BY *ConstExpression* clause. If present, this gives the amount the index variable is to be incremented (or decremented) by every time through the loop. If this clause is not present, one is assumed to be the default increment.
Conditional Statements
The IF statement is again similar to Pascal’s with the exception of the ELSIF clause (see Section 0). This clause consists of an expression (Boolean) followed by a sequence of statements. This allows the easy nesting of multiple conditions. The semantics are:

1. Evaluate the condition that takes the form of an expression yielding the value TRUE or FALSE.
2. If the value is TRUE, the THEN part is executed.
3. If the value is FALSE, the following ELSIF or ELSE is executed. If there is no following clause, the statement terminates.

Input/Output Statements
Input and output are defined in the language on both standard input/output and on files. We only concern ourselves with standard input/output.

The READ statement has only a single argument. The type of the argument indicates the type of value that is to be read and assigned to the value of the argument. A string that is read is defined to contain no white space. Only variables of types CHAR, BOOLEAN, INTEGER, and CARDINAL can be read from standard input.

The WRITE statement has only a single argument. The type of the argument indicates the type of the expression to be written. The only expression types that can be written are CHAR, BOOLEAN, INTEGER, and CARDINAL. The WRITELN statement simply outputs a new line character.

Data Types
Only four elementary data types are used: INTEGER, CARDINAL, BOOLEAN, and CHAR. The legal values and operators that can operate on them are similar to Pascal. The CARDINAL data type is similar to INTEGER except no negative numbers are allowed. Thus you have twice as many CARDINAL numbers as you have positive INTEGERS.

The structured data types, ARRAYS and RECORDS, are similar in declaration and use to Pascal.
Scope
The scope of an identifier is similar to Pascal.

Procedures
Procedures are similar in declaration, scope, and use as Pascal with the following exception. Procedures may have a RETURN statement that immediately terminates the execution of the procedure and returns control to the calling location. If the procedure has no RETURN statement, control is returned when the final END is reached.

Functions
Functions are similar in declaration, scope, and use to Pascal, with the following exception. Functions have a RETURN statement used to return the value (of an elementary data type) from the function. The type of the expression used in the RETURN statement must match the type of the function as it was declared. All functions must have at least one RETURN statement.

Grammar
The subset of Modula-2 grammar we use is given below. Those productions starting with “*” are optional and do not need to be implemented when you generate code. It is expected that those who want an “A” will implement all of these optional features. These optional features are implemented in the lexical analyzer, parser, and symbol table by all. The optional features are not to be done until the project is working fully and correctly.

CompilationUnit = ProgramModule
ProgramModule = MODULE Ident ";" Block Ident "."
Block = {Declaration} BEGIN StatementSequence END
ConstantDeclaration = Ident "=" ConstExpression
TypeDeclaration = Ident "=" Type
VariableDeclaration = IdentList ":=" Type
ProcedureDeclaration = ProcedureHeading ";" Block Ident
ConstExpression = SimpleConstExpr [Relation SimpleConstExpr]
SimpleConstExpr = [UnOperator] ConstTerm {AddOperator ConstTerm}
ConstTerm = ConstFactor {MulOperator ConstFactor}
ConstFactor = Ident
| Number
| String
| Character
| "(" ConstExpression ")"
| NOT ConstFactor
Type = SimpleType
| ArrayType
| RecordType
SimpleType = Ident
ArrayType = ARRAY SubrangeType OF Type
* RecordType = RECORD FieldListSequence [";" ] END
SubrangeType = "[" ConstExpression "." ConstExpression "]"
* FieldListSequence = FieldList [";" FieldList]
* FieldList = IdentList ":" Type
ProcedureHeading = PROCEDURE Ident FormalParameters
FormalParameters = "(" [FPSection [";" FPSection]] ")" [":" SimpleType]
FPSection = [VAR] IdentList ":" SimpleType
IdentList = Ident [";" Ident]
Declaration = CONST {ConstantDeclaration [";" ]}
| TYPE {TypeDeclaration [";" ]}
| VAR {VariableDeclaration [";" ]}
| ProcedureDeclaration [";" ]
StatementSequence = Statement [";" Statement]
Statement = Assignment
Assignments = Designator "::" Expression
ProcedureCall = Ident "(" ActualParameters ")"
IfStatement = IF Expression THEN StatementSequence
   {ELSIF Expression THEN StatementSequence}
   [ELSE StatementSequence] END
WhileStatement = WHILE Expression DO StatementSequence END
RepeatStatement = REPEAT StatementSequence UNTIL Expression
ForStatement = FOR Ident "::" Expression TO Expression
   [BY ConstExpression] DO
      StatementSequence END
ReadStatement = READ "(" Designator ")"
WriteStatement = WRITE "(" Expression ")"
   | WRITELN
NullStatement =
Designator = Ident {"," Ident | "[" ExpList "]"}
ExpList = Expression {"," Expression}
Expression = SimpleExpression [Relation SimpleExpression]
SimpleExpression = [UnOperator] Term {AddOperator Term}
Term = Factor {MulOperator Factor}
Factor = Number
| String
| Character
| Ident "(" ActualParameters ")"
| Designator
| "(" Expression ")"
| NOT Factor
ActualParameters = [ExpList]

**Miscellaneous**

**Keywords**
The elementary types, INTEGER, CARDINAL, BOOLEAN, and CHAR are not reserved words but are keywords. In addition, the Boolean values TRUE and FALSE are keywords. The difference between reserved words and keywords is that reserved words may not be redefined, but keywords may be redefined.

**Standard Procedures**
Standard procedures/functions are predefined but may be redefined by the user. The standard procedures/functions we deal with are:

CHR(x) function – the character with ordinal number x.
ORD(x) function – ordinal number of x in the set of values defined by type T of x.
DEC(x) procedure – x := x - 1.
INC(x) procedure – x := x + 1.
Features not Implemented
The features that are not implemented by this grammar and those that may be somewhat different are: procedure types, open arrays, variant records, real, enumerated, and bitset types, pointer type, with statement, modules (except main modules), priorities, import, export, input/output statements are different, and many standard procedures/functions.

Example
The following is a sample program written using Modula-2. It implements the towers of Hanoi problem. It is included to give the reader a feeling for the language. Study it carefully.

(* This program simulates the tower of Hanoi problem. *)

MODULE HANOI;

CONST
  mindisk = 1;
  maxdisk = 26;
  terminate = -1;

TYPE
towertype = RECORD
  tower : ARRAY [ mindisk .. maxdisk] OF CHAR;
  top : INTEGER;
END; (* RECORD *)

VAR
  source,dest,aux : towertype;
  disk : INTEGER;
  counter : INTEGER;
PROCEDURE initialize();
BEGIN (* initialize *)
    source.top := mindisk - 1;
    dest.top := mindisk - 1;
    aux.top := mindisk - 1;
END initialize;

PROCEDURE setsource(): INTEGER;
VAR
    num : INTEGER;
    disk : CHAR;
BEGIN (* setsource *)
    READ(num);
    IF ( num <= terminate ) THEN
        WRITE("PROCESSING COMPLETED GOOD-BYE");
        WRITELN;
        RETURN (num);
    END; (* IF *)
    FOR disk := 'A' TO CHR( ORD("A") + num - 1) DO
        INC(source.top);
        source.tower[ source.top ] := disk;
    END; (* FOR *)
END setsource;
END; (* FOR *)
RETURN(num);
END setsourse;

(* MOVE *)

PROCEDURE move( VAR source, dest : towertype );
BEGIN (* move *)
    INC(dest.top);
    dest.tower[ dest.top ] := source.tower[ source.top ];
    DEC(source.top);
END move;

(* PRINT *)

PROCEDURE print();
VAR
    index : INTEGER;
BEGIN (* print *)
    WRITE("source       -> ");
    FOR index := mindisk TO source.top DO
        WRITE(source.tower[index]);
        WRITE(" ");
    END; (* FOR *)
    WRITELN;
    WRITE("auxiliary    -> ");
FOR index := mindisk TO aux.top DO
    WRITE(aux.tower[index]);
    WRITE(" ");
END; (* FOR *)
WRITELN;
WRITE("destination -> ");
FOR index := mindisk TO dest.top DO
    WRITE( dest.tower[index]);
    WRITE(" ");
END; (* FOR *)
WRITELN;
WRITELN;
END print;

(* TOWERS *)

PROCEDURE TOWERS( disk : INTEGER; VAR source,dest,aux : towertype );
BEGIN (* TOWERS *)
    IF disk = 1 THEN
        move(source,dest);
        print();
    ELSE
        TOWERS( disk - 1, source, aux, dest);
        move(source,dest);
        print();
        TOWERS( disk - 1, aux, dest, source);
    END IF;
END TOWERS;
BEGIN (* HANOI *)
  counter := 0;
  REPEAT
    initialize();
    disk := setsource();
    IF (disk <> terminate)
      THEN
        INC(counter);
        WRITELN;
        WRITE("   TOWERS OF HANOI TRACE : test number -> ");
        WRITE(counter);
        WRITELN;
        print();
        TOWERS( disk, source, dest, aux);
      END; (* IF *)
  UNTIL (disk <= terminate);
END HANOI.

Introduction to the Compiler

Miscellaneous Issues

Options to the Compiler
The main program must allow for at least three options:

- -a file: file is the name of the output assembly file. In the absence of this option, the default name is MIPS.s.
- -s: print the local symbol table on exit from every scope.
- -h: how to use the compiler.

To enable these actions, include the desired option(s) on the command line. Arguments from the command line are generally made available to the main function in C and C++ using the variables named argv and argc. See your C or C++ documentation to see how these variables are actually declared and used.

Separate Modules
It is convenient to dedicate modules (C or C++ files) to contain the routines associated with a particular function or phase of the compiler. For example, utility.c might contain utility routines that are needed by many other modules. symboltable.c might contain the routines associated with the symbol table. Along with these modules, there should be a corresponding .h file that contains function prototypes for the routines in the .c file. These .h files must be included in modules that access functions defined in other modules. The type of the parameters should be listed in parentheses separated by commas. The formal parameter names are optional. See Figure 0.1 for an example.

```
TYPE *typecreate (int, TY_KIND, ID *, TYPE *);
ID *entertype (ID *, TYPE *, ID_KIND);
```

Figure 0.1 Sample extern declaration of function

Global Variables
Your compiler needs to maintain certain global variables. One, with the name currscope, is an integer variable that contains the current lexical scoping depth. Others global variables are introduced as needed.

It is helpful, for organizational purposes, to dedicate a module, say globals.c that simply contains the definitions of all global variables used by the compiler. A corresponding file, globals.h, contains all the variables defined using an extern declaration. It is the file globals.h that is included in all the other modules that reference global variables. The module globals.c has to be compiled like all other modules.

Strictly speaking, this is not the only technique that can be used for the declaration of global variables. Global variables may be defined in any module as long as each global variable is defined in only one module, and declared as extern in the other modules that access it. Either way, make sure that global variables are defined and initialized in exactly one module.

Types in the Compiler
It is also useful to maintain a single file, called types.h, that contains all of the type definitions needed by the compiler. This file can then be included in other modules that make use of these types (which is all of the other modules). As we proceed further, type declarations are given that should be included in this module.
# if UNIX, uncomment the following eight lines
CC = gcc
SUFF =
OBJ = o
CFLAGS = -c -g
LFLAGS = -ll -lc -lg
YACCSEP = .
LEXSEP = .
LNAME = -o m2
YACC = bison
LEX = flex

OBJJS = main.$(OBJ) m2$(YACCSEP)tab.$(OBJ) lex$(LEXSEP)yy.$(OBJ) \
    globals.$(OBJ) symboltable.$(OBJ) utility.$(OBJ)
SRCS = main.c m2.tab.c lex.yy.c globals.c symboltable.c utility.c

m2$(SUFF): $(OBJJS)
    $(CC) $(LNAME) $(LFLAGS) $(OBJJS)
m2$(YACCSEP)tab.c: m2.y
    $(YACC) -vd m2.y
lex$(LEXSEP)yy.c: m2.l types.h globals.h utility.h
    $(LEX) m2.l

clean:
    rm -f *.$(OBJ) core m2.output m2$(YACCSEP)tab.h m2$(YACCSEP)tab.c\
    lex$(LEXSEP)yy.c

main.$(OBJ): main.c
globals.$(OBJ): globals.c types.h
symboltable.$(OBJ): symboltable.c types.h globals.h utility.h
utility.$(OBJ): utility.c types.h globals.h symboltable.h
m2$(YACCSEP)tab.$(OBJ): m2$(YACCSEP)tab.c types.h globals.h symboltable.h utility.h
lex$(LEXSEP)yy.$(OBJ): lex$(LEXSEP)yy.c m2$(YACCSEP)tab.h
Maintaining Modules

Figure 0.2 An example makefile

Modules can be maintained, compiled, and linked using the make facility available on most systems. Figure 0.2 shows a sample makefile (for a Linux system) for the compiler. Use this as an example to create your own makefile. You are required to use the make facility or the .Net project for your compiler.

In the makefile, lines beginning with # indicate comments. Comments are terminated by the end of line. Lines having an equal sign are defining variables. Line 12 shows the definition of OBJS, a variable that represents all the object modules of the compiler. Notice that the line terminates with a back slash. This indicates the line continues onto the next line with the lines being concatenated together. Lines beginning in the first column (e.g., line 16) show a name, a colon, and the names of the modules on which that name depends. The line(s) following (beginning with a tab character, e.g., line 17) show the action(s) to be performed if any of the dependent modules have been modified. In the example, m2 (the name of the compiler), is shown to depend on the object files in the variable OBJS. Macro-like variables are referenced as shown, $(OBJS). If one of the files in the line has been modified since m2 was created, the line(s) following is (are) executed. Thus if any of OBJS have been modified since m2 was created, a new version of m2 is created by relinking all the object modules. For more information on make, see the documentation.

The Semantic Stack and Semantic Actions

The compiler must maintain a semantic stack which, in tandem with the parser stack, helps the compiler manipulate and determine the meaning of the program it compiles. The entries in the semantic stack represent the values or meanings of all the grammar symbols. In contrast, the parser stack only maintains enough information to decide the form, or syntax, of the compiled program.

Exactly what constitutes values for the entries in the semantic stack depends heavily on the semantics of the language being compiled, and the objectives of the translator (e.g., true compiler, preprocessor, interpreter, etc.). It is hard to provide a general treatment of the issues in this course. We concentrate on the specific task of a one-pass compiler for Modula-2 that generates machine code for a MIPS machine, hoping that the experience gained extrapolates to related problems.

Manipulation of the semantic stack is triggered by the parser during reduce moves. Typically, the values of the symbols on the right-hand-side of the production are popped off the semantic stack, a value for the symbol on the left-hand-side is determined from these values and the semantic rules of the language, and that new value is pushed back on the semantic stack. All symbols in the grammar have values associated with them in the semantic stack even though some of these values may be NULL (indicating no information is needed about this symbol). Not all reductions require semantic actions. One of the most important decisions in the design of the compiler is to establish early what kind of value each symbol has and what productions require semantic actions associated with their reductions. The remaining chapters in this manual help you understand how to resolve these issues.
Yacc (or Bison) allows you to specify the semantic action associated with the symbol on the left-hand-side of the production. These actions may return values and may utilize the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired.

An action is specified by writing arbitrary C or C++ statements enclosed in curly braces ({}). For example, consider Figure 0.3.

```
Expression:   Expression ADDSY Expression
            {$$ = binary_op($1, AddOp, $3);}  
```

**Figure 0.3 Yacc production and semantic action**

To return a value for the symbol `Expression` on the left-hand-side of the production, a value is assigned to the variable $$$. To refer to values of symbols on the right-hand-side of the production, the variables $1$, $2$, $3$, ..., are used and these refer to the values returned by the components of the right side of the production reading from left to right. In Figure 0.3, the `Expression` on the left-hand-side is assigned the value returned by the call to the function `binary_op`. This function has three arguments, the value of the first expression on the right-hand-side ($1$), the enumerated type indicating the particular operator (AddOp), and the value of the second expression on the right-hand-side ($3$). If no action is specified after a particular production, the default action $$ = $1$; is performed.

Values on the semantic stack do not have to be of a single type. In Yacc, you may define the type of the value on the semantic stack. This is done using a %union statement as shown in Figure 0.4. This is part of the Yacc input. This example shows that the elements of the semantic stack can be any one of the six types shown. Each of these types has to be previously defined and would be included in the file `types.h`.

```
%union
{
    int   int_val;
    char *name_ptr;
    CONS *cons_ptr;
    ID   *id_ptr;
    TYPE *ty_ptr;
    EXPR *exp_ptr;
}
```

**Figure 0.4 Example %union statement**
Yacc must know which type is associated with each symbol in the grammar. This is done using the %type statement as shown in Figure 0.5. This shows that the symbols IdentList, Ident, and ProcedureName all are pointers to ID records when placed on the semantic stack. **Beware:** once you start using the %type statement, you are forced to type many symbols even though you may not be using them. You are required to use %type statement in your project.

```
%type <id_ptr> IdentList Ident ProcedureName
```

Figure 0.5 Example %type statement

A complete discussion of Yacc can be found in the documentation.

**Initial Values for the Semantic Stack**

As the semantic stack is only manipulated via reductions, the determination of the values of certain terminal symbols becomes somewhat unclear. Remember that there are certain terminal symbols in the grammar that have a specific instance, or value, that determines their semantic properties. For Modula-2, these are symbols such as an identifier (IDENTSY) and constants such as an integer constant (INTCONSTSY). For an identifier, the name of the identifier needs to be known in addition to the token IDENTSY. For an integer constant, the actual value of the constant needs to be known in addition to the token INTCONSTSY. In order for the parser to obtain these additional values, the lexical analyzer (Lex or Flex) has to communicate this to the parser (Yacc). This is done in Lex by assigning a value to the variable yylval. This value is placed on the semantic stack when the lexical analyzer returns a token to the parser. For example, Figure 0.6 shows a few lines of Lex input.

```
{letter}({lord})*  {vecho(); yylval.name_ptr = strsave(yytext);
                return(IDENTSY);}  
{digit}+  {vecho(); yylval.int_val = intconvert(yytext, 0,
                strlen(yytext) - 1, 10); return(INTCONSTSY);}  
```

Figure 0.6 Example using yylval

This example shows the lexical analyzer returning not only a token (via the return statement), but also assigning a value to yylval. Note the field name used in referencing yylval are the same field names defined in the %union statement (see Figure 0.4).

For further information on this, see the Yacc information as stated above and information on Lex in the documentation.

**Precedence and Associativity**

Yacc allows the use of an ambiguous grammar for expressions and then the use of %left, %right, and %nonassoc to express associativity with each representing what you would expect. Precedence is shown by the order in which these statements are ordered, with the statements shown first representing the lowest precedence. For example, consider Figure 0.7. The
relational operators have the lowest precedence while multiply and divide have the highest. You are **required** to use an ambiguous grammar for expressions and constant expressions and these features of `Yacc` in your project.

```
%nonassoc EQUALSY NESY GTSY GESY LTSY LESY
%left PLUSSY MINUSSY
%left STARSY SLASHSY
```

*Figure 0.7 Showing precedence in Yacc*
The Symbol Table

The proper handling of type related information by a compiler is of vital importance, especially for strongly typed languages. This chapter describes the kind of information that must be maintained, and shows, in a rather detailed manner, how to utilize such information. It is important to develop a clear feeling about this material, as type handling is involved in many aspects of the compilation. For example, management of type related information must occur in all declarations (constants, types, variables, procedures, and functions), in expressions of all kinds, and in many of the different statements.

Semantic Structures for Types

A data structure to properly handle types is shown in Figure 0.1. This figure contains the declaration for the C structure type_info that is sufficient for describing all useful aspects of types in Modula-2. This information would be placed in the file types.h.
typedef enum type_kind TY_KIND;
enum type_kind { Integer, Char, Boolean, String, SubRange, Array, Record, UndefinedType};

typedef struct type_info TYPE;

struct type_info
{
 int ty_size;
 TY_KIND ty_kind;
 TYPE *ty_next;
 union
 {
  struct
   {
    TYPE *RangeType;
    int min;
    int max;
   } ty_subrange;
  struct
   {
    TYPE *ElementType;
    TYPE *IndexType;
   } ty_array;
  struct
   {
    ID *FirstField;
   } ty_record;
  }
   ty_form;
}; /* type_info */

Figure 0.1  Data structure for type information
Let us describe briefly the meaning of the fields of type_info in Figure 0.1. Note that in the declarations, the name TYPE is used as the name of the data structure for type_info.

- **ty_size**: The number of bytes needed to allocate storage to objects of this type.
- **ty_kind**: Describes indirectly the property of the type itself, by the enumeration type TY_KIND. The value of TY_KIND describes the possible kinds of types and is declared prior to type_info.
- **ty_next**: An auxiliary pointer field used to link types together as needed.
- **ty_form**: A union containing entries for those types that need more information.
  - **ty_subrange**: Contains additional information for a subrange type. This includes the type of the subrange and the minimum and the maximum values the subrange includes. This field is used to maintain information about the upper and lower bounds of an array.
  - **ty_array**: Contains additional information about arrays. This includes the type of the elements and the type of the index for the array.
  - **ty_record**: Contains additional information about records. This includes a pointer to the list of fields in the record.

It should be apparent from the description above that the type related information is intertwined, as type structures point to other type structures, etc. Given the semantics of Modula-2, one can rely on the fact that the graph of the data structures can never contain cycles. This would have not been the case, however, had we allowed pointer types (e.g., type a = record ... ; b:↑a; ... end;). On the other hand, it is often the case that there are many actual links (pointers) to the same type structure, so care must be exercised to determine when a given type is no longer needed.

This particular organization of type information enables the compiler to treat types efficiently with minimal effort. In particular, strong type checking can be accomplished trivially, as will be seen soon. Type information is present in both the symbol table and the semantic stack, interlinked in a number of ways. How this information is utilized by the compiler is now described.

**Creating a Type**

It is useful to define and implement certain utility functions that can simplify the effort needed to implement type related parts of the compiler. A good place to put these functions is the file symboltable.c. Let us first describe a function named typecreate shown in Figure 0.2.

As you can see, typecreate creates and returns an instance of TYPE (with space acquired dynamically via malloc), with the values of the fields ty_size and ty_kind always set from the arguments and the field ty_next initialized to NULL. Depending on the kind of type created, the formal parameters id_list and type are used for initial values. You should always be careful to initialize all fields so you do not have surprises later.

The compiler needs to declare the global variables shown in Figure 0.3. These variables provide easy access to the structures for the predefined primitive types. An initialization function, typeinit, that initializes these global variables with their values is shown in Figure 0.4. It should be called once from the main program, just before the parser is called. The code for
typeinit should be included in the file symboltable.c. Notice the use of named constants (e.g., INTSIZE) for the sizes of the various types. The definition of these constants is shown in Figure 0.5 and should be included in the file types.h.

```c
TYPE *typecreate (int size, TY_KIND kind, ID *id_list, TYPE *type)
{
    TYPE *t;

    t = (TYPE *)malloc(sizeof(TYPE));
    t->ty_size = size;
    t->ty_kind = kind;
    t->ty_next = NULL;
    switch (kind)
    {
    case SubRange:
        t->ty_form.ty_subrange.RangeType = type;
        break;
    case Array:
        t->ty_form.ty_array.ElementType = type;
        break;
    case Record:
        t->ty_form.ty_record.FirstField = id_list;
        break;
    default:
        break;
    } /* switch */
    return(t);
} /* typecreate */
```
Figure 0.2 Function `typecreate`

```c
```

Figure 0.3 TYPE related global variables

```c
void typeinit (void)
{
  int_type = typecreate(INTSIZE, Integer, NULL, NULL);
  card_type = typecreate(CARDSIZE, Cardinal, NULL, NULL);
  bool_type = typecreate BOOLSIZE, Boolean, NULL, NULL);
  char_type = typecreate(CHARSIZE, Char, NULL, NULL);
}
```

Figure 0.4 Function `typeinit`
Symbol Table Organization
The symbol table serves as one of the two main repositories of semantic information in the compiler (the other one being the semantic stack), and contains information describing the properties (attributes) of the various entities that are named via identifiers. What entities can be named via identifiers depends on the specific language being compiled. Typically, they include variables, procedures, functions, types, labels, (named) constants, formal parameters for procedure/functions, etc.

For block structured languages with static name-binding (such as ALGOL-60, Pascal, and Modula-2), the symbol table organization must accommodate the scoping rules of the language. This means that the symbol table must allow identifiers that are already in use in an enclosing scope to be reused, without conflict. It must also ensure that, when the symbol table is searched for an identifier, its most recent definition is always found. Finally, the symbol table must be able to accommodate the removal of all identifiers that are associated with a given scope when that scope is closed.

Something worth mentioning at the outset is that the attributes of the various identifiers are not relevant in the organization of the symbol table insofar as searching, insertion, and deletion of entries are concerned. It is only the variable name and the scope of that name that is important.

The main issue is the scoping rules of the particular language, namely at what point in a program is a new scope entered, and at what point is a scope closed.

In the following, a simple, yet practical and efficient approach for the organization of the symbol table of a one-pass compiler for block structured languages is presented. The data structure used for the symbol table of each scope is a binary search tree. A stack of symbol tables is maintained with an entry in the stack pointing to the root of the tree for each currently active scope.

The simplicity of the scheme is due to the fact that scopes do not overlap, but are wholly nested within each other. Using this observation, a counter of the nesting depth of scopes is sufficient to determine its properties. We follow the convention that depth 0 represents the scope of the predefined identifiers for the language (e.g., INTEGER, TRUE, etc.), depth 1 denotes the identifiers declared at the global (main) level, and depth 2 denotes identifiers declared within procedures or functions. Note that even though Modula-2 allows the user to define procedures/functions within another procedure/function, this feature is not to be implemented.
The compiler maintains a scope counter, `currscope`, with initial value zero, that is incremented by one each time a new scope is opened (i.e., when the compiler begins processing the main program or a new procedure/function) and is decremented by one when a scope closes (after the compiler completes processing the main program or a procedure/function). To illustrate this situation, consider the Pascal-like fragment shown in Figure 0.6. The markers indicate at which points in the program the scope counter changes value. It increases by one after a new procedure or function name is seen, and decreases by one when the corresponding end is seen. At any given time, all new entries in the symbol table are made with the depth indicated by the current value of `currscope`. Whenever `currscope` is incremented, a new (empty) symbol table is pushed onto the stack. Whenever `currscope` is about to get decremented, the symbol table for that scope must be popped from the stack. Code for the declaration of the stack of symbol tables and `currscope` is shown in Figure 0.7 and should be included in the file `globals.c`. The constant `SCOPEDEPTH` indicates how deeply nested scope can get. A value of three is sufficient for this project.

```
1->  program X;
    type a = array [1..10] of boolean;
    var i, j : integer;
2->  procedure p1 (a, b : integer);
    var i, k : boolean;
    begin
      ......
1->    end; { p1 }
2->  procedure p2 (x, y : boolean);
    var j, b : boolean;
3->    function f (v1, v2 : char) : integer;
        var b : char;
    begin
      ......
2->      end; { f }
    begin
      ......
1->    end; { p2 }
    begin
      ......
0->  end. { X }
```

Figure 0.6 Pascal-like code fragment

```
#define SCOPEDEPTH 3
ID *scope [SCOPEDEPTH];
```

Figure 0.7 Declaration of symbol table stack
It is important to realize the implications of the fact that a new scope is opened after the name of the procedure/function is seen. This means that the name itself is defined (declared) in the previous scope. For example, the names \( p_1 \) and \( p_2 \) in the program frame of Figure 0.6 are defined at scope 1. This means that the names themselves persist after the procedures/functions are fully processed, whereas their local identifiers (e.g., \( j \), \( b \), etc.) are deleted. This allows the names of procedures/functions to be called by the block that encloses them.

**The C Code for the Symbol Table Management**

The routines implementing the management of the symbol table are outlined below. The data structure for nodes in the symbol table is shown in Figure 0.8. Note that nodes in the symbol table are referred to as type ID.

```c
typedef enum identifier_kind ID_KIND;
enum identifier_kind {Constant, Type, Variable, RParameter, VParameter,
                    Field, Procedure, Function};

typedef struct id_info ID;

struct id_info
{ char *id_name;
  int id_addr;
}

Figure 0.8 Data structure for a symbol table entry
```

Let us briefly describe each of the fields contained in the structure `id_info`.

- `id_name`: Pointer to the name of the symbol.
- `id_addr`: Run-time address of the symbol.
- `id_level`: Lexical level at which the symbol is declared.
- `id_type`: Pointer to the type of the symbol.
- `id_left`: Pointer to the left subtree of the binary search tree.
- `id_right`: Pointer to the right subtree of the binary search tree.
- `id_kind`: Describes the use of the symbol. The various uses are those shown in the enumerated type `ID_KIND` declared prior to `ID`.
- `id_next`: Pointer used to link symbols together. This field is used to link formal parameters and record fields.
- `id_value`: Used to store a constant value.

```c
ID *newid (char *name)
{
  ID *new_id;
```
Figure 0.9 Function `newid`

```c
ID *search (char *name, ID *table)
{
    int temp;

    if (table == NULL)
        return(NULL);

    temp = strcmp(name, table->id_name);
    if (temp == 0)
```
Accessing the Symbol Table

Functions used to create and access nodes in the symbol table are described below:

- **newid**: A function used to create a new instance of a node of the symbol table. It is called with one argument, a pointer to the character string representing the name of the symbol and returns a pointer to a new ID record fully initialized. This function is shown in Figure 0.9.
- **search**: A function used to look up a name in a symbol table and is shown in Figure 0.10. It takes two arguments: (a) a pointer to the name to search for, and (b) a pointer to the symbol table to be searched. It returns a pointer to the node found in the symbol table. If a node with the same name is not found, the NULL pointer is returned. Two functions, that call search, are useful: (a) **localsearch**, searches the local symbol table for a particular name, and (b) **globalsearch**, searches all the symbol tables in the stack for a particular name. If desired, these could also be combined into a single function with a Boolean flag indicating whether a local or global search is required.
- **entername**: This function enters a name into the symbol table. It has one argument: a pointer to the ID record to be entered into the symbol table and returns a pointer to the entry made. entername assumes that there is no other entry with the same name at the same depth and issues an error if another one is found. The code for entername is similar in structure to the code in search and is not repeated. Make sure you take care of the case of entering a name in an empty symbol table.
- **exitscope**: It is called to close the current scope, i.e., to delete from the stack of symbol tables the symbol table associated with the current scope.

**Initialization of the Symbol Table**

The symbol table must be initialized with the entries for the predefined types, the Boolean constants, and all predefined procedures and functions. These should all be defined at scope 0. This initialization must be performed once, after the initialization of the types has been performed, but before the parser is invoked.

The following function is useful in doing this initialization as well as useful during the compilation process. This is:

- **entertype**: This function has three arguments: (a) a pointer to a symbol table node, (b) a pointer to a type for the entry, and (c) the kind of variable. The function places this information in the proper fields for the entry.

```
entertype(entername(newid(strdup("INTEGER"))), int_type, Type);
```
Using the functions just described, initialization of the symbol table can be done using the function called \texttt{symtabinit}. Examples of the code from this function are shown in Figure 0.11 - Figure 0.12. Figure 0.11 shows an example of type initialization. Figure 0.12 shows example code for placing \texttt{TRUE} into the symbol table.

That way, your symbol table is fully initialized. The initialization function should also set \texttt{currscope} = 1, after the symbol table has been initialized. The compiler is then ready to take off.
**Declarations**

The previous chapter discussed the symbol table and the data structures used to implement it. This chapter shows how these data structures are used, in conjunction with the grammar, to create the symbol table for a program as it is parsed. This chapter deals with all types of declarations: simple declarations, user-defined types (including records and arrays), and the declaration of procedures and functions.

**Simple Declarations**

This section deals with the declaration of variables and shows how simple types are handled.

**Declaration of Variables**

The production that describes the declaration of variables is shown in Figure 0.1. IdentList denotes a comma-separated list of identifiers, and Type denotes any valid type specification. Before we get into details about the semantic actions for this production, we have to decide the semantic meaning for the two nonterminal symbols.

```plaintext
Figure 0.1 Production for VariableElement

<table>
<thead>
<tr>
<th>VariableElement</th>
<th>: IdentList COLONSY Type SEMICOLONSY</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{vardecl($1, $3);}</td>
</tr>
</tbody>
</table>
```

We assume that the nonterminal symbol Type always has a corresponding value on the semantic stack that is a pointer to a TYPE record. For this to hold true, we must ensure that the productions that define Type have corresponding semantic actions that properly determine a value for the Type record. Do not get confused with Type, a nonterminal symbol in the grammar, and TYPE, a data structure. The relevant productions that define Type are shown in Figure 0.2. Now, since each production for Type has a single (nonterminal) symbol on its right-hand-side, we do not have to provide explicit semantic actions for them, if we arrange that each such nonterminal also has a semantic value that is a pointer to a TYPE record.

The function typename has as a parameter a pointer to an ID record (returned from qualident) and returns a pointer to a TYPE record obtained from the ID record. This is the most commonly occurring type, and it includes the predefined types INTEGER, BOOLEAN, etc. The treatment of the remaining types is covered in subsequent sections.

```plaintext
Type : SimpleType

| ArrayType
| RecordType
;```
Type Identifiers

The most elementary and basic case is when Type is denoted by an identifier, based on the production in Figure 0.2. For the discussion of the semantic actions associated with this production, assume the declarations specified in Figure 0.3. The type declaration for CONS, as used in the figure, is shown in Figure 0.6. Note that the function qualident has a single parameter, a pointer to a character string, and returns a pointer to an ID record. The character pointer is returned from the lexical analyzer, along with the token IDENTSY, when an identifier is recognized (see Section 0). The actions for the function qualident are as follows:

- Let name point to the name of the identifier. It points to the specific instance of IDENTSY. Set st_ptr = globalsearch(name).
- If an entry for name is not found (i.e., st_ptr == NULL), issue a diagnostic to that effect.
- Otherwise, release the space for IDENTSY (i.e., free(name)), and return(st_ptr).

The semantic actions just described are all your compiler needs to do in order to handle named types. Notice that the net effect of this action is that a pointer to some TYPE structure is pushed onto the semantic stack (by assigning it to $$). Most often, this is a pointer to the TYPE record representing the identifier.

Identifier Lists

A common nonterminal symbol in the grammar is IdentList whose production is shown in Figure 0.4. The semantic value for IdentList is a pointer to a linked list of ID records. The value returned by the function identlist is exactly what the name implies. Let us define the actions of the function identlist. There are two parameters to identlist: (a) a pointer to an identifier name (name), and (b) a pointer to the head of a linked list of ID records (list). The function returns a pointer to an ID record, the head of the linked list.
More precisely, the actions are:

- Create a new ID node (i.e., \( st\_ptr = \text{newid} (\text{name}) \)).
- Make \( st\_ptr \) the new head of the linked list (i.e., \( st\_ptr->id\_next = \text{list} \)).
- Return the newly created node (i.e., \( \text{return} (st\_ptr) \)).

**Variable Declarations**

Remember again that the pertinent production for variable declarations is shown in Figure 0.1. A way to maintain the current size of the data segment is needed so run-time addresses can be maintained. The manner in which this is done is left to you. Let us assume that the variable \( \text{ldatasize} \) contains this information. Assume the type declarations shown previously in Figure 0.3. The semantic actions, as defined by \( \text{vardecl} \), for this production are very simple. There are two parameters to \( \text{vardecl} \): (a) a pointer to a list of ID records, and (b) a pointer to a TYPE record \( (ty\_ptr) \). For every element in the list, \( st\_ptr \), do the following:

- Set the type of \( st\_ptr \) to be \( ty\_ptr \) (\( st\_ptr->id\_type = ty\_ptr \)).
- Enter \( st\_ptr \) into the symbol table. Issue an error message if the identifier is already present in the symbol table.
- Give \( st\_ptr \) its run-time address. If the variable is local, this is done by decrementing \( \text{ldatasize} \) by the size of the type \( (ty\_ptr->ty\_size) \) and assigning \( st\_ptr->id\_addr = \text{ldatasize} \). If the variable is global, assign \( \text{gdatasize} \) to \( st\_ptr->id\_addr \) and increment \( \text{gdatasize} \) by the size of the type.

The semantic actions shown above suffice to handle the declaration of all variables.

<table>
<thead>
<tr>
<th>ConstElement : IDENTSY EQSY ConstExpression SEMICOLONSY</th>
</tr>
</thead>
<tbody>
<tr>
<td>{constdecl($1, $3);}</td>
</tr>
</tbody>
</table>

**Figure 0.5 Production for ConstElement**

```c
typedef struct cons_info CONS;

struct cons_info
{
    TYPE *cons_type;
};
```

**Figure 0.6 Structure for constants**

**Definition of Named Constants**

While on the subject of declarations, we can quickly take care of the declaration of named constants. Named constants in Modula-2 are introduced via the production shown in Figure 0.5.
To represent the semantic value of a constant on the semantic stack, a new structure needs to be defined. This structure is shown in Figure 0.6 and should be placed in the file types.h. Notice the structure contains the pointer to the type of the constant and the constant value.

The function `consdecl` has two parameters: (a) a pointer to a character string that represents the identifier name (`name`), and (b) a pointer to a CONS structure (`cons_ptr`). The function returns no value. Its semantic actions are as follows:

- Search the local symbol table for `name`.
- If `name` is already defined, issue an appropriate error message.
- Create a new ID node. Assume `st_ptr` points to that node.
- Enter `st_ptr` into the symbol table. Put the type in the symbol table (i.e., `st_ptr->id_type = cons_ptr->cons_type`).
- Indicate the variable is a constant (i.e., `st_ptr->id_kind = Constant`).
- Put the constant value in `st_ptr` (`st_ptr->id_value = cons_ptr->cons_value`).
- `free(cons_ptr)`.

**Constant Expressions**

Constant expressions are used in the declarations of constants as well as in the declaration of subranges. This section discusses how constant expressions are handled by the compiler.

Consider Figure 0.7 that shows the productions for `ConstExpression`. `ConstExpression` returns a pointer to a CONS record. The function `evalcons` takes three arguments, two operands (pointers to CONS records) and an operator (an enumerated type) and returns a pointer to a CONS record that represents the evaluation of the expression. The functions `makeintcons` and `makecharcons` take the appropriate constant and returns a pointer to the correct CONS record.

The function `getcons` takes a pointer to an ID record, checks that it is truly a constant, and returns a pointer to a CONS record that represents that constant. The details for all of these functions are straightforward and are left to you.

**Strings**

The function `makestrcons` needs a little more explanation. Once a string constant has been recognized, it is necessary to keep track of it so the strings can be placed in the correct location once code is generated. This is done by maintaining a linked list of all strings recognized.

**String Data Type**

The data structure used to store the linked list of strings is shown in Figure 0.8. Two variables, `str_head` and `str_end`, are used to point to the start of the string linked list and the end of that list. These variables should be placed in the file `globals.c`. Both of these variables are initialized to NULL when the compiler starts. Every time a string is found in a constant expression, an entry is added to the end of this linked list.

The code for `makestrcons` is shown in Figure 0.9.
ConstituteExpression : ConstituteExpression ORSY ConstituteExpression
{$$ = evalcons($1, OrOp, $3);}
| ConstituteExpression ANDSY ConstituteExpression
{$$ = evalcons($1, AndOp, $3);}
| NOTSY ConstituteExpression
{$$ = evalcons($2, NotOp, NULL);}
| ConstituteExpression EQSY ConstituteExpression
{$$ = evalcons($1, EqOp, $3);}
| ConstituteExpression NESY ConstituteExpression
{$$ = evalcons($1, NeOp, $3);}
| ConstituteExpression LESY ConstituteExpression
{$$ = evalcons($1, LeOp, $3);}
| ConstituteExpression GESY ConstituteExpression
{$$ = evalcons($1, GeOp, $3);}
| ConstituteExpression LTSY ConstituteExpression
{$$ = evalcons($1, LtOp, $3);}
| ConstituteExpression GTSY ConstituteExpression
{$$ = evalcons($1, GtOp, $3);}
| ConstituteExpression ADDSY ConstituteExpression
{$$ = evalcons($1, AddOp, $3);}
| ConstituteExpression SUBSY ConstituteExpression
{$$ = evalcons($1, SubOp, $3);}

Figure 0.7 Productions for ConstituteExpression

typedef struct str_list STR;

struct str_list
{
    char *str_ptr;
};
User Defined Types

This section shows how the compiler handles user-defined types, namely records and arrays.

Definition of New Type Names

New type names are introduced in Modula-2 via the production shown in Figure 0.10.

The function `typedecl` has two parameters: (a) a pointer to an identifier name (`name`), and (b) a pointer to a TYPE record (`ty_ptr`). The function returns no value. The corresponding semantic actions are as follows:

- Search local scope for a previous definition of `name`.
- If `name` is already defined in the current scope, issue a diagnostic to that effect.

```c
CONS *makestrcons (char *str)
{
    CONS *cons_temp;
    STR *str_temp;

    cons_temp = (CONS *)malloc(sizeof(CONS));
    cons_temp->cons_type = str_type;
    cons_temp->cons_value = sdatasize;
    sdatasize += strlen(str) + 1;
    str_temp = (STR *)malloc(sizeof(STR));
}
```

**Figure 0.8 String linked list data structure**

**Figure 0.9 Code for makestring**

**Figure 0.10 Production for TypeElement**
• Otherwise, create a new ID node and enter it into the symbol table. Assume st_ptr points to the node in the symbol table.
• Enter the type into the symbol table (i.e., st_ptr->id_type = ty_ptr).
• Indicate this variable is a type (i.e., st_ptr->id_kind = Type).

Definition of Subrange Types
Subrange types are defined by the production shown in Figure 0.11 and are used in Modula-2 only in defining the upper and lower bounds of an array. The function subrangetype has two parameters that are pointers to CONS structures, call these min and max respectively. It returns a pointer to a TYPE record. The semantic actions performed by this function are straightforward:
  • Create a new type ty_ptr = typecreate(min->cons_type->ty_size, SubRange, NULL, min->cons_type)
  • Place the minimum and maximum values in ty_ptr according to the values in min and max.
  • return(ty_ptr).

<table>
<thead>
<tr>
<th>SubrangeType</th>
<th>: LEFTBRACESY ConstExpression DOTDOTSY ConstExpression</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RIGHTBRACESY</td>
</tr>
<tr>
<td>$_$ = subrangetype($2 $4);</td>
<td></td>
</tr>
</tbody>
</table>

Figure 0.11 Production for SubrangeType

Definition of Records

<table>
<thead>
<tr>
<th>RecordType</th>
<th>: RECORDSY FieldListSequence ENDSY</th>
</tr>
</thead>
<tbody>
<tr>
<td>$_$ = recordtype($2);</td>
<td></td>
</tr>
</tbody>
</table>

Figure 0.12 Production for RecordType

A record type is defined via the production shown in Figure 0.12. We assume that the semantic value corresponding to FieldListSequence in the semantic stack is a pointer to a list of ID records. The function recordtype has one parameter a pointer to a list of ID records (list), and returns a pointer to a TYPE record. The semantic actions for recordtype are as follows:
  • Set offset = 0 (offset is a local variable).
  • For each element (st_ptr) in list do the following:
    o Set field id_kind to Field.
    o Set field id_addr to offset.
    o Update offset (i.e., offset += st_ptr->id_type->ty_size).
  • Create a new type for this record
    ty_ptr = typecreate(offset, Record, field_list, NULL)
  • return(ty_ptr).
If there are name conflicts, an appropriate diagnostic must be issued.

FieldListSequence : FieldList

{$$ = fieldlistsequence($1, NULL);}

| FieldListSequence SEMICOLONY FieldList
Figure 0.13 Production for FieldListSequence

One now has to make sure that the nonterminal FieldListSequence is evaluated properly. The productions for FieldListSequence are shown in Figure 0.13. It is assumed that the nonterminal FieldList’s semantic value is a pointer to a list of ID records. The function fieldlistsequence has two parameters both of which are pointers to lists of ID records (list1 and list2) and returns a pointer to a list of ID records. The semantic actions are as follows:

- Find the end of list1 and hook list2 onto it.
- return(list);

The production for FieldList is shown in Figure 0.14. The function givetype has two parameters: (a) a pointer to a list of ID records, and (b) a pointer to a TYPE record. For each identifier in the list, the proper type must be given. The pointer to the list of ID records is returned. Fill in the details.

FieldList : IdentList COLONSY Type

{$$ = givetype($1, $3)}

Figure 0.14 Production for FieldList

Definition of Array Types

ArrayType : ARRAYSY SubrangeType OFSY Type

{$$ = arraytype($2, $4);} 

Figure 0.15 Production for ArrayType

The definition of array types is also quite easy. The only production for array types is shown in Figure 0.15 Production for ArrayType

The nonterminal symbol SubrangeType refers to the type of the subscript and returns a pointer to a TYPE record (as shown in Section 0). The function arraytype has two parameters: (a) a pointer to a TYPE record (ty_temp), and (b) a pointer to a TYPE record (ty_ptr) and returns a pointer to a TYPE record. The semantic actions for the above production are as follows:

- Create a new type for the array. ty_temp1 = typecreate(arraysize(ty_temp, ty_ptr), Array, NULL, ty_ptr). The function arraysize finds the size of an array given two parameters, pointers to the type for the index and to the type for the elements. The size is determined
as follows. The size of the array is the size of the elements multiplied by the upper bound of the subrange minus the lower bound of the subrange plus one.

- The type of the index is placed in the newly created `TYPE` record. `ty_temp1->ty_form.ty_array.IndexType = ty_temp`
- `return(ty_temp1)`.

Remember, the declaration of multidimensional arrays is an array of arrays.

**Declaration of Procedures and Functions**

This section describes the semantic actions for the definition of procedures and functions. Before we begin a detailed presentation, it will be instructive to describe the overall approach in intuitive terms. For the sake of brevity, when we mention procedures we really mean both procedures and functions, unless indicated otherwise.

Information for a given procedure is entered in the symbol table when the procedure is declared. The definition of a procedure proceeds in the following manner.

First, the procedure header is processed. The header contains the procedure name and the description of its formal parameters, where each parameter description includes its name and type, and whether it is a `reference` or a `value` parameter. For functions only, the header also specifies the type of the value returned by the function. The information provided by the header is all that is really needed by the compiler in order to know precisely how to generate code for calling that procedure. This observation is very important for understanding the handling of forward declarations of procedures by the compiler. An entry is made in the symbol table and that entry contains whatever the compiler needs to know in order to process a call to that procedure.

After that, the compiler must process the declaration section for this procedure. The bulk of this is handled by routines already described (and, one hopes, already implemented). However, before proceeding with the translation of local declarations, the compiler must first increment the value of `currscope` (in effect opening a new scope), and then it must make an entry in the symbol table for each formal parameter as if it were a local variable. It can then proceed with the processing of the local declarations.

Because the handling of local variables is exactly the same as has been previously discussed, no further discussion is presented here. We next look at the declaration of the procedure name and the formal parameters.

**Declaration of the Procedure Header**

```
ProcedureHeading : PROCUREURESY ProcedureName FormalParameters
    {\$ = procedureheading($2, $3);}
```

*Figure 0.16 Production for ProcedureHeading*
The production for the declaration of the procedure header is shown in Figure 0.16. It is assumed that the nonterminal symbol ProcedureName has as a semantic value a pointer to an ID record (st_ptr) and that FormalParameters has as a semantic value a pointer to a list of ID records (list). These are passed to the function procedureheading that returns a pointer to an ID record representing the procedure in the symbol table. The semantic actions that occur in procedureheading are:

- For every item in the list (call the item st_temp), do the following:
  - Decrement the value of pdatasize by:
    - Size of the formal parameter if it is a VParameter (value parameter).
    - Size of a pointer if it is a RParameter (reference parameter).
  - Assign an address to this formal parameter (i.e., st_temp->id_addr = pdatasize).
- Set st_ptr->id_next = list. Place a pointer in the symbol table entry for the formal parameter list.
- return(st_ptr).

pdatasize starts off as the total amount of storage needed by the formal parameters. Its value is set in the routines dealing with the formal parameters (Section 0).

### Declaration of the Procedure Head

![Figure 0.17 Production for ProcedureName](image)

We now need the semantic actions associated with ProcedureName. The production defining ProcedureName is shown in Figure 0.17. The function procedurename has a single parameter, a pointer to the procedure name (name), and returns a pointer to an ID record. The actions it performs are:

- Check that name is not in the symbol table. If it is, issue an error message.
- st_ptr = newid(name).
- Enter the ID record into the symbol table.
- Set the field id_kind to be Procedure.
- Increment currscope.
- Set pdatasize to four.
- return(st_ptr).

### Declaration of the Function Head
The production for the declaration of the function header is shown in Figure 0.18. The function heading has three parameters: (a) a pointer to an ID record (representing the function name), (b) a pointer to a list of ID records (representing the formal parameters), and (c) a pointer to a TYPE record (representing the type of the value returned by the function). The semantic actions for FunctionHeading are similar to ProcedureHeading with the following additions:

- Set the field id_kind to Function (rather than Procedure, as it had already been set).
- Set st_ptr->id_type = st_temp (assume st_temp is the name of the third parameter).
- Check that st_temp is a scalar type. If not, issue an error.

### Declaration of the Formal Parameters

Formal parameters are a bit more complex. The production for formal parameters is shown in Figure 0.19. The production is easily handled; there is nothing to do but put on the stack the correct semantic value.

```
FunctionHeading : FUNCTIONSY ProcedureName FormalParameters

COLONSY Type

$$ = functionheading($2 $3 $5);$$

Figure 0.18 Production for FunctionHeading

```

```
FormalParameters : LEFTPARENSY OptFPSectionList RIGHTPARENSY

{$$ = $2;}

Figure 0.19 Production for FormalParameters

```

```
OptFPSectionList : FPSectionList

| /* empty */

{$$ = NULL;}

;

FPSectionList : FPSection

{$$ = $1;}

| FPSectionList SEMICOLONY FPSection

{$$ = fnsectionlist($1, $3);}  

```

The Symbol Table
The Symbol Table

Figure 0.20  Production for the remained for formal parameters

The productions for finishing the rest of the formal parameters are as shown in Figure 0.20. The actions for OptFPSectionList are obvious. The actions for FPSectionList connect the two lists produced by FPSectionList and FPSection. The actions for FPSection are also straightforward. Passed as parameters to fpsection are (a) an enumerated type denoting whether the parameter is passed by reference or value, (b) a pointer to a list of ID records (list), and (c) a pointer to a TYPE record for these parameters. The function returns the pointer to a list of ID records (list). fpsection assigns the type to every element in the list, indicates the kind of parameter this element represents (VParameter or RParameter), enters all elements of the list into the symbol table, and does any necessary address calculation.

The calculation of addresses for formal parameters is a little confusing. This confusion is fully explained in a subsequent chapter. For now, let the following discussion suffice. The address of the formal parameters needs to be assigned in the reverse order from their declaration. Thus, the function fpsection needs to calculate the total space required for all the parameters. This can be done using a global variable, named pdatasize, that is defined in the file globals.c. This global variable has the current size of the formal parameters processed thus far and should be initialized to four. In the function, the size of each VParameter is added to pdatasize, and the size of a pointer is added for each RParameter. See if you can figure out why. The details are left to the reader.

This essentially completes the description of the semantic actions for procedure declarations.
**Issues on Code Generation**

The run-time structure of Modula-2 programs is described in this chapter. The material presented here should be viewed in conjunction with the MIPS machine that is the target for your compiler. For more information about the MIPS assembly language and to get a MIPS simulator (spim) see [www.cs.wisc.edu/~larus/spim.html](http://www.cs.wisc.edu/~larus/spim.html).

**The Run-Time Structures of a Modula-2 Program**

![Figure 0.1 Overall organization of memory](image)

The conceptual run-time organization of memory, as viewed by the compiler, has the structure shown in Figure 0.1. There, you see a map of the program’s memory image at run time, separated into five main segments:

- **S1**: The first segment contains system information.
- **S2**: The second segment contains the instructions for the various procedures, functions, and also for the main program.
- **S3**: The third segment contains the initialized data for the program. For Modula-2, these data only reflect the values of the string constants encountered in the program. More generally, in this segment one would also find the storage for the variables that are initialized to something other than 0 (e.g., remember the declarations `int x=10, y=20;` in C).
- **S4**: The fourth segment contains the uninitialized data for the main program (i.e., global data that are initialized to 0). In our case it consists of the space allocated for the global variables.
- **S5**: Finally, the fifth segment is the memory set aside for the run-time stack of the program. This segment contains the local data (local variables, parameters, etc.) for the activations of the procedures/functions in the program. Also, it may be used to get temporary space during execution for the evaluation of expressions.

---

1 The run-time organization presented here is tailored specifically to the implementation of a subset of Modula-2. For a more general treatment of the subject, read sections 10.1, 10.2, and 10.4 in A&U, chapter 9 in F&L, or section 6.2 in Holub.
The run-time stack, residing in the fifth segment, is necessary because procedures/functions in Modula-2 can be recursive. This makes static allocation of storage for their variables inappropriate. To understand the operation of the stack, consider the following.

Because we are not going to allow nested definitions of procedures/functions in our implementation of Modula-2, we are not going to worry about referencing nonlocal variables. A variable is nonlocal if it is not local and it is not global. During the execution of a Modula-2 program, only global variables and the local variables of the executing procedure/function are accessible. The data for procedures/functions that have not yet completed execution are suspended. To illustrate this situation, let us assume that a given Modula-2 program has three procedures A, B, and C. Let us further suppose that at some point during execution of that program the main program has called A, which has called B, which has called C, which has called A (indirect recursion), which has called B (also indirect recursion), which is executing. At this point, there is only one procedure active, namely the most recent activation of B. There are also two inactive (or suspended) activations of A, and one suspended activation for each of B and C. Each activation of a procedure has in the run-time stack (maintained in segment S5) an instance of the space for its local data, what is traditionally known as an activation record. At this point in execution, a snapshot of segment S5 has the form shown in Figure 0.2. For reasons that become apparent later, it is convenient to have the stack growing from high addresses to low addresses in memory.

![Figure 0.2 Sample case of the run-time stack](image)

At any given time during execution, only global data (residing in segments S3 and S4), and that data in the top activation record are accessible. The remaining activation records are inaccessible until the activations of the procedures/functions that they have invoked complete execution and return.

Because of this run-time organization of memory (necessitated by the recursive nature of procedures), the precise addresses of data local to procedures/functions are not known by the compiler during compilation. However, they can always be referenced via a known offset from the variable address of the top activation record. This is possible because the activation records for procedures/functions have a format determinable by the compiler during compilation. That format is explained below specifically for Modula-2.
Register Usage
This section briefly outlines the usage of the global and local registers for the MIPS machine. The usage of registers in this section is only a suggestion and other schemes may be used if you wish.

Global Registers
It is assumed throughout that four specific registers, or names, of our machine are dedicated to hold addresses related to the organization of memory. These registers and names are shown in Figure 0.3. Specifically, the label $SA$ is the start of the initialized data area. Register 28 ($ga$) points to the start of the global data area. Register 29 ($sp$) always points to the top of the stack; more precisely the value of $sp$ is the address of the last used portion of segment S5. Register 30 ($fp$) always points to a specific point inside the top activation record. These registers allow us to reference data as an offset from the appropriate register.

<table>
<thead>
<tr>
<th>Register</th>
<th>Name</th>
<th>Contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>SA</td>
<td>$SA$</td>
<td>Address of initialized data area</td>
</tr>
<tr>
<td>28</td>
<td>$gp$</td>
<td>Global data area</td>
</tr>
<tr>
<td>29</td>
<td>$sp$</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>30</td>
<td>$fp$</td>
<td>Frame pointer</td>
</tr>
</tbody>
</table>

Figure 0.3 Dedicated global registers

Local Registers
There are 18 registers (registers 8-25) that are allocated by the compiler. The local registers may be allocated in many ways. We choose a very simple-minded method that is adequate for this project, but would not be sufficient for a real compiler because of its inefficient usage of the registers. We treat the 18 local registers as a set of available registers. The routines to manipulate this set are described below.

- **getreg**: Find an available register, remove it from the set, and return the number of the register allocated. If all the registers are in use, issue an appropriate message.
- **retreg**: Return a register (that is a parameter) that was in use and place it back in the register set indicating it is now not in use.
- **clearreg**: Place all registers back in the register set showing all registers are now not in use.

Registers are cleared (via clearreg) at the end of each Modula-2 statement (e.g., assignment, read, write, etc.). In this way, fewer registers are used because they are not maintained from statement to statement. This also causes inefficient program execution because values are loaded into registers and then saved at the end of the statement even though they might be used again in the next statement.

Activation Records for Procedures and Functions
The information in the activation record has the format shown in Figure 0.4.
If you look closely, you realize that the precise format of the activation records, for any given subprogram, is easily computed by the compiler. Specifically, after the header of a subprogram is seen, the number and type for each of the arguments is known. The next two fields (the return address and the old value of the frame pointer) are standard. Therefore, this portion of the activation record, up to and including the old frame pointer field, is determined by the subprogram header alone. The value returned by a function is placed in register $v0 (register 2).

The portion for the local variables is completely determined after the compiler has processed the variable declarations for the subprogram. After that, when code is generated for the instructions in the subprogram, local variables and arguments can be accessed via a constant offset from the address specified by the value of the frame pointer register ($fp).

In the activation record, in general, there is some space allocated for storing temporary values (typically intermediate values during expression evaluation). The size of this portion is not known until after the compiler has seen and processed all the statements for a subprogram. Depending on how expressions are translated by the compiler, this field may have many entries, or very few. As a matter of fact, this field may not be needed at all if temporary space is allocated by the compiler at the top of the stack (via the stack pointer register $sp).

In general, a subprogram must also save, in the activation record, the contents of registers it uses to ensure that evaluation of expressions in the calling subprograms is not affected. Again, the compiler has to wait until it has compiled all the statements of a subprogram before it knows how many, and which, registers have been used. To simplify our implementation, we save all the local registers when a procedure/function is called.

Figure 0.4 Activation records for procedure and functions
The assignment of offsets to variables, global or local, has been previously discussed in 0. All registers are must be saved on the run-time stack and restored when the procedure/function call returns.

For the addresses of the parameters and local variables to be valid at run-time, actual space must be allocated for each activation record in the run-time stack. We see shortly how this can be accomplished with minimal effort. In simple terms, the invocation of each subprogram involves some set up by the calling program and some by the called program to allocate enough room in the stack. The code that creates the activation record is called the prologue. In a similar manner, subprogram return involves some work by the called program, and some by the calling program. The code to clean up the activation record is called the epilogue.

**Initial Code Generated**

Before the compiler starts generating code, an initial code sequence must be generated in order to establish the correct values in the global registers and jump to the proper location to start execution. See Figure 0.5 for the initial code segment. When this code sequence is generated, the labels SA, GA, and main are unknown. These have to be defined in the program. The label SA is generated as the label for the first line of the string constants. The label GA is generated after the last of the strings has been output. The final instruction is a jump to the label b_begin, the address of the main program. This is because there may be many procedures and functions defined before the main program appears.

```
.text
.globl main
main:   la $gp, GA       # Initialize the global pointer
```

*Figure 0.5 Initial MIPS code generated*

Three variables are used to keep track of the sizes of the different data areas. They are described below.

- **gdatasize**: This variable keeps track of the amount of space that is currently needed for the global data area. This variable is updated every time a new global variable is defined. It is also used in assigning run-time addresses to global data. At the end of the program, gdatasize contains the total amount of space needed for the global area.
- **ldatasize**: This variable keeps track of the amount of space that is currently needed for the local data area. This variable ldatasize is updated every time a new local variable is defined. It is also used in assigning run-time addresses to local data. At the end of the declarations in a procedure/function, ldatasize contains the total amount of space needed to be allocated for the current procedure/function.
- **sdatasize**: This variable keeps track of the amount of space that is currently needed for the string data area. This variable is updated every time a new constant string is used. It is also used in assigning run-time addresses to all strings. At the end of the program, sdatasize contains the total amount of space needed for this area.
**Finishing Up**

After the parse is complete there are a few unfinished tasks that need to be completed.

- The contents of the string linked list must then be output. This means an `.asciiz` command is generated for every string. Also, on the first one of these generated, the label SA must be prepended.
- After the last of the strings is output, a label similar to GA must be generated. This defines the global data area.

When this is all completed, you have a program that is ready to be assembled and executed.
Expressions
This chapter discusses all aspects of handling expressions in the Modula-2 grammar. First, the semantic structures for expressions are defined along with routines for maintaining these structures. Then primitive expressions are discussed, both constant and variable. Finally the handling of arithmetic and Boolean expressions is shown.

Semantic Structures for Expressions
Information about expressions must be maintained in the semantic stack during the parsing of expressions. The semantic information for expressions is represented by the C structure shown in Figure 0.1. This code is placed in the file types.h.

```
typedef enum expr_kind EX_KIND;
enum expr_kind {GlobalV, LocalV, LocalA, Cons, RegisterV, RegisterA, FuncRef};

typedef struct expr_info EXPR;

struct expr_info
```

Figure 0.1 Data structures for expressions
It is important to understand each field in this structure, so they are discussed below.

- **ex_type**: This field describes the type of the expression.
- **ex_inv**: This field is used in various ways described below.
- **ex_kind**: This field describes where the value represented by the expression is stored. The different values for this field are described below.
  - **GlobalV**: The value of the expression is stored in the global data area. The offset of the location is stored in the `ex_inv` field. Referencing expressions of this type use register `$gp` and the offset in `ex_inv`.
  - **LocalV**: The value of the expression is stored in the activation record of the currently active procedure or function. The offset is stored in `ex_inv`. Referencing expressions of this type use register `$fp` and the offset in `ex_inv`.
  - **LocalA**: The address for the value of the expression is stored in the activation record of the currently active procedure or function. The offset is again stored in `ex_inv`. To reference an expression of this type, the address (using `$fp` and `ex_inv`) has to be loaded into a register and then the value is loaded using that address.
  - **Cons**: The value of the expression is a constant. If its type is `INTEGER`, `CARDINAL`, `CHAR`, or `BOOLEAN` its value is stored in the `ex_inv` field.
Referencing constant expressions is done by placing the literal in the instruction. Finally, if the expression has type string, the value stored in ex_inv is the offset from the beginning of the string data area of the first character in the string. Referencing expressions of this type use the label SA and the offset in ex_inv.

- RegisterV: The value of the expression is in a register. The register number is stored in the ex_inv field. Expressions are referenced using that register.
- RegisterA: The address of the value of the expression is in a register. The register number is stored in the ex_inv field. The expression is referenced by loading the value using the address in the register.
- FuncRef: The expression is a function reference.

- ex_next: This field is used to link together lists of EXPR records.

A useful question to ask yourself to see if you really understand the above information is why there isn’t a GlobalA kind for an expression. See if you can figure this out and what it would take to make GlobalA necessary.

**Useful Utility Functions for Expressions**

It is very helpful to have a function that creates a new instance of an expression, with its fields set according to values passed as parameters. Specifically, `newexpr` is defined in Figure 0.2.

```
EXPR *newexpr (TYPE *type, EX_KIND kind, int inv)
{
    register EXPR *e;

    e = (EXPR *)malloc(sizeof(EXPR));

    e->ex_type = type;
}
```

*Figure 0.2 Code for newexpr*

Another useful function, `makeexpr`, is used in the for statement. It takes as a parameter a pointer to an ID record and creates an EXPR record. The code is shown in Figure 0.3. The constant GLOBALSCOPE is defined to be one, the level where all global variables are defined.
**Primitive Expressions**

In this section we show how the compiler handles expressions with no operators, involving only constants or variables. The relevant semantic actions are quite simple, and can be handled with very little effort.

**Constant Expressions**

The productions for constants within an expression are shown in Figure 0.4. As can be seen, integer and character constants are handled by simply calling the routine `newexpr` with the proper arguments. Note that the third argument is a value returned from the lexical analyzer (the actual integer or character constant). See Section 0 to review this material.

```c
EXPR *makeexpr (ID *st_ptr)
{
    TYPE *type;
    EX_KIND kind;
    int inv;

    type = st_ptr->id_type;
    inv = st_ptr->id_addr;
    if (st_ptr->id_level == GLOBALSCOPE)
        kind = GlobalV;
    else if (st_ptr->id_level == currscope)
```

![Figure 0.3 Code for makeexpr](image_url1)

![Figure 0.4 Production for constant expression](image_url2)
A little more work is necessary with the string constant because the string must be added to the end of the linked list of strings as discussed in Section 0. The code for `strexpr` is shown in Figure 0.5. This routine creates a new `EXPR` record and places the string on the linked list.

```c
EXPR *strexpr (char *str)
{
    EXPR *ex_temp;
    STR  *str_temp;

    ex_temp = newexpr(str_type, Cons, sdatasize);
    sdatasize += strlen(str) + 1;
    str_temp = (STR *)malloc(sizeof(STR));
    if (str_head == NULL)
```

Figure 0.5 Code for `strexpr`

```
    return ex_temp;
}
```

**Variable Use**

We have shown in the previous section how to handle expressions that have only literal constants in them. We now describe the use of variables in expressions. The production that defines an identifier (we are not considering array and record references at the present time) is shown in Figure 0.6. Note that `Designator` is one of the aliases used for `Expression` in the grammar. The semantic actions for the function `designator` are as follows. `designator` has a single parameter, a pointer to a character string, and returns a pointer to an `EXPR` record.

- Look up the identifier in the symbol table.
- One expects to find the `id_kind` field equal to `Variable`, `Constant`, `RParameter`, or `VParameter`.
- If the name is not found, or if it is found but represents some other kind than those mentioned above, emit an appropriate error message.

```
Figure 0.6 Production defining `Designator`

<table>
<thead>
<tr>
<th>Designator : . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDENTSY</td>
</tr>
<tr>
<td>$=$ designator($1);</td>
</tr>
<tr>
<td>[Fig 0.6 ] Production defining <code>Designator</code></td>
</tr>
</tbody>
</table>

Variable Use

We have shown in the previous section how to handle expressions that have only literal constants in them. We now describe the use of variables in expressions. The production that defines an identifier (we are not considering array and record references at the present time) is shown in Figure 0.6. Note that `Designator` is one of the aliases used for `Expression` in the grammar. The semantic actions for the function `designator` are as follows. `designator` has a single parameter, a pointer to a character string, and returns a pointer to an `EXPR` record.

- Look up the identifier in the symbol table.
- One expects to find the `id_kind` field equal to `Variable`, `Constant`, `RParameter`, or `VParameter`.
- If the name is not found, or if it is found but represents some other kind than those mentioned above, emit an appropriate error message.
• Otherwise, using the information in the symbol table, make an \texttt{EXPR} record and return a pointer to that record. Make sure the proper information is placed in the \texttt{EXPR} record. Make sure you handle the different possibilities for \texttt{IDENTSY} (e.g., constant, variable).

This is all that needs to be done for variables referenced in expressions.

\textbf{Arithmetic Operators}

This section discusses the translation of unary and binary arithmetic operators.

\textbf{Binary Arithmetic Operators}

We describe next the translation of the five binary arithmetic operators, $+$, $-$, $\ast$, $/$, and \%. It turns out that we can easily use a generic function, named \texttt{binop}, to generate code for all five of them, given the specific operator as one of the parameters. The relevant productions from \texttt{Expression} are shown in Figure 0.7. As can be seen, \texttt{binop} has three parameters: (a) a pointer to an \texttt{EXPR} record, (b) an enumerated type representing the operator, and (c) a pointer to an \texttt{EXPR} record. An outline of the code for \texttt{binop} is shown in Figure 0.8.

\begin{figure}[h]
\centering
\begin{verbatim}
Expression : . . .
    | Expression ADDSY Expression
        {$$ = binop($1, AddOp, $3);}
    | Expression SUBSY Expression
        {$$ = binop($1, SubOp, $3);}
    | Expression MULSY Expression
        {$$ = binop($1, MulOp, $3);}
\end{verbatim}
\caption{Relevant productions for binary operators}
\end{figure}

The functions \texttt{getreg} and \texttt{retreg} have been previously discussed.
Unary Minus

The unary minus is incorporated into the language via the grammar production shown in Figure 0.9. Its treatment is similar to the treatment of the binary arithmetic operators shown in Figure 0.8. The details are left to you. Remember that the unary minus is only applicable to expressions of INTEGER type.
Boolean Expressions
This section discusses the code generated for Boolean expressions composed of relational and Boolean operators.

Relational Operators
The handling of the relational operators is handled using the MIPS comparison instruction. The code generated for this construct matches the template shown in Figure 0.10, where ? is the appropriate condition.

Boolean Operators
To complete the treatment of Boolean expressions, we need to describe how to handle the Boolean operators NOT, OR, and AND. The first is a unary operator. The relevant productions are shown in Figure 0.11. Instructions exist in the MIPS machine that directly implement these operators. The code looks similar to that we saw for arithmetic operators (see Figure 0.8).
Simple Statements

This chapter deals with simple statements, statements that do not involve any flow of control. The write statements are the first statements introduced. Then the assignment and read statements are discussed. Finally, the null statement is discussed.

Write Statement

The write statements are particularly simple to handle. In MIPS, printing is done using system calls. The applicable system calls are shown in Figure 0.1. It is described via the productions shown in Figure 0.2. A point worth remembering, because it is encountered again later, is that the nonterminal WriteStatement needs no semantic value.

<table>
<thead>
<tr>
<th>Service</th>
<th>System Call Code</th>
<th>Arguments</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_int</td>
<td>1</td>
<td>$a0 = integer</td>
<td></td>
</tr>
<tr>
<td>print_string</td>
<td>4</td>
<td>$a0 = address of string</td>
<td></td>
</tr>
<tr>
<td>read_int</td>
<td>5</td>
<td>$a0 = buffer</td>
<td>integer in $v0</td>
</tr>
<tr>
<td>read_string</td>
<td>8</td>
<td>$a0 = buffer</td>
<td>$a1 = length</td>
</tr>
<tr>
<td>exit</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 0.1  MIPS system calls

WriteStatement : WRITESY LEFTPARENSY Expression RIGHTPARENSY

{writestatement($3);}
| WRITELNSY

Figure 0.2 Productions for write statement

The function writestatement has a single parameter, a pointer to an EXPR record. It does not return a value. The semantic actions follow:

- Make sure the type of the EXPR record is a valid type for the write statement. If it is not, issue an appropriate diagnostic message and quit.
- Otherwise, place the value of the expression in register $a0 ($a1).
- Generate the proper system call.
- Free any registers used.
- Free the EXPR record.

See Figure 0.3 for an example of printing an integer.
These simple actions are all that are needed for the implementation of the `write` statement.

The function `writeln` has no parameters. The function should generate code to output the carriage return and line feed characters.

**Assignment Statement**

We next turn our attention to the simple assignment statement, so you can actually assign values to variables. Remember that it is described via the production shown in Figure 0.4.

Assignment : Designator `ASSIGNSY` Expression

{assign($1, $3);}

It should be noted that the only things we can have on the left-hand-side of the assignment statement at this point in time is a simple variable. We have not discussed the referencing of records or arrays. This is dealt with in Chapter 9. It should also be noted that type checking has to occur to make sure the types of the left-hand-side and right-hand-side are the same. Also, the left-hand-side has to represent a kind that can be stored into (e.g., it cannot be a constant).

**Read Statement**

The handling of the `read` statement is almost identical to the handling of the `write` statement already described. It can be described via the production in Figure 0.5.
The semantic actions for Designator have already been described in Section 0. The semantic actions for ReadStatement are described below. The function readstatement has a single parameter, a pointer to a list of EXPR records. The semantic actions follow:

- Make sure the value represented by the EXPR record can have an address (e.g., constants do not have addresses). If it does not, issue an appropriate error message.
- Generate the proper system call.
- Move the number read to the proper location.
- Free any registers that are no longer needed.
- Free the EXPR record.

**Null Statement**

The production for the null statement is shown in Figure 0.6. There are no semantic actions necessary for the null statement.

**Control Statements**

This chapter discusses the four statements that deal with flow of control. These are the if statement, the while statement, the repeat statement, and the for statement.

**If Statement**

The handling of the if statement, although by no means difficult, provides the first clear illustration thus far for the need to rewrite, in a nontrivial manner, perfectly valid grammar productions in order to provide a one-pass compiler with semantic *hooks* so it can generate code at the proper time. Rather straightforward examples of such a need have already been encountered, but now things become somewhat more complicated. More such cases arise in the treatment of the various iterative statements, and also in the treatment of procedures and functions.
To understand what needs to be accomplished with the code generated for if statements, we must first completely comprehend its semantics. To illustrate the issue, see the template shown in Figure 0.1. It demonstrates the overall structure of the if statement, accompanied with a skeleton of the code generated.

The template consists of five blocks, marked if, elsif, else one, else two, and end. Each has two columns, the first representing Modula-2 code, and the second the MIPS machine code associated with it. The first and last blocks (if and end) always appear. The second block (elsif) should be interpreted as a repetition of zero or more such blocks, appended next to each other. The third and fourth (else one and else two) should be interspersed as alternatives (i.e., for any given if statement, exactly one of the two is present). It should be clear from the template that the code that must be generated by the compiler for the if statement, over and above the code generated by its various components (Expression and StatementSequence), consists of a series of jumps skipping over parts of code, reflecting the value of the Boolean expressions guiding the flow of control in the if statement. More precisely, we have two different kinds of jumps:

- A conditional jump, effected when the expression in the if and elsif parts is false, skipping over the next statement sequence and going to the next alternative (if any).
- An unconditional jump, effected after a chosen alternative has completed execution, with the target being a label marking the end of the whole if statement.

| +-- | if expression | code for Expression in REG |
|     | if           | beqz REG, next1           |
|     | if then StatementSequence | code for StatementSequence |
|     | if           | j out                     |
| +-- |
| +-- |
|     | elsif Expression | next1: |
|     | elsif        | code for Expression in REG |
|     | elsif        | beqz REG, next2           |
|     | elsif        | then StatementSequence    | code for StatementSequence |
|     | elsif        | j out                     |
| +-- |

**Figure 0.1 Template for if statement**
Notice also that in the if statement, all jumps, conditional or unconditional, are transferring control in a forward fashion. When the j instructions are generated, the locations where the jumps go to are not known, thus you need to generate a new label for the instructions. When the location for the jumps has been determined, these labels have to be printed. Based on these observations, the semantic actions to implement if statements are outlined below.

Figure 0.2 Productions for if statement

The first step is to rewrite the productions that describe the if statement to enable the invocation of semantic actions at the appropriate points. This is accomplished by introducing artificial reductions, without changing the syntax, because semantic actions can only be triggered via reductions. The set of productions shown in Figure 0.2 provides one such choice. The argument to the semantic action in ThenPart should be especially noted. You should understand what this does and an explanation can be found in the Yacc manual. Look it up and figure out why it is necessary.

Let us briefly discuss the purpose of each of the semantic routines found in the productions of Figure 0.2:

- ifwhilehead: This function has a single parameter, a pointer to an EXPR record and returns an integer. It performs the following actions:
  - Check that type of the expression is Boolean. If not, generate an appropriate error message.
  - Generate a new label (label).
  - Generate a conditional jump on equ to label.
  - Free any registers not needed.
  - Free the EXPR record.
  - Return the integer representing label.
• thenpart: This function has a single parameter, an integer, representing a label and returns an integer representing a label. The actions are as follows:
  o Generate a new label (label).
  o Generate an unconditional jump instruction to label.
  o Print the label represented by the parameter.
  o Return the integer representing the new label (label).
• elsifsequence: This function has two integer parameters, (label1, label2), representing labels and returns an integer representing a label. The actions are:
  o Generate an unconditional jump instruction to label1.
  o Print label2.
  o Return the integer representing label1.
• ifstmt: This function has one parameter, an integer representing a label. The semantic action is:
  o Print the label represented by the parameter.

These semantic actions fully suffice for the implementation of the if statement. It is obvious that once the rationale behind the various choices is understood, the semantic actions themselves border on trivial.

**While Statement**
The while statement is also quite easy. Its basic template is shown in Figure 0.3. The productions for the while statement are shown in Figure 0.4.

```
while start:
  Expression code for Expression in REG
    jeqz REG, end
  do StatementSequence code for StatementSequence
```

*Figure 0.3 Template for while statement*
Notice that the semantic actions of if while head were previously described in the if statement (see Section 0). Therefore, we only need to describe while repeat head and while stmt.

- **while repeat head**: This function has no input parameters and returns an integer representing a label.
  - Generate a new label (label).
  - Print label.
  - Return label.

- **while stmt**: This function has two parameters, an integer representing the label of the start of the while statement, and an integer representing the label of the end of the while statement. It returns nothing. The semantic actions are:
  - Generate an unconditional jump to the start of the while statement (given by parameter one).
  - Print the label represented by parameter two.
  - Free all registers not in use.

**Repeat Statement**

With its template shown in Figure 0.5, the repeat statement is also easily implemented, based on the productions shown in Figure 0.6.
The semantic actions for `whilerepeathead` were given in Section 0. The semantic actions for `repeatstmt` are as follows:

- `repeatstmt`: This function has two arguments, an integer representing the label (`label`) of the start of the `repeat` statement and an expression representing the condition. The semantic actions are:
  - Check that the type of the expression is Boolean. If not, generate an appropriate error message.
  - Generate a jump on `eqz` to `label`.

**For Statement**

The handling of the `for` statement is more complex than that of the other loops, but still quite manageable. Figure 0.7 shows the template corresponding to the `for` statement. Depending on whether we have the `to` or the `downto` step, the jump on the condition `gt` or `lt` and the code to increment or decrement is applicable. Remember that the only types allowed for `ID` are integer, cardinal, character, or Boolean. Finally, `I` corresponds to the location for accessing the index of the loop, and `T` represents a temporary location, implicitly declared by the compiler as if it were a normal variable (global or local, depending on `currscope`) to hold the value of the expression denoting the end value of the loop index. `T` should be allocated on the stack.
Again, to provide the semantic actions for the FOR statement, we must rewrite the corresponding productions. An appropriate solution is given in Figure 0.8. The nonterminal symbol ToDownto places on the semantic stack an integer constant indicating the amount to increment the loop index on each iteration. Based on the template in Figure 0.7, and the four functions defined in Figure 0.8, the semantic actions are outlined below.

- **forhead**: This function has two parameters: (a) a pointer to a character string (name) and (b) a pointer to an EXPR record (expr) and returns a pointer to an EXPR record. The semantic actions for the function are as follows:
  - Look up name in the symbol table. Call the entry st_ptr. It is expected that name denotes a variable of the correct type.
  - Generate code to move the value of the expression (expr) to the location of st_ptr.
Return an expression record made from st_ptr.
There are many checks that must be made before all this can take place, in order to ensure
that the construct is valid. Specifically, the type of expr must not be undef_type, st_ptr,
must be found, and represent a variable (with its kind field not equal to Cons), the type
fields of st_ptr and expr must be the same and either integer, cardinal, character, or
Boolean. If this is not the case, issue appropriate diagnostics.

- forexpression: This function has two parameters, (a) an integer representing the label
  (label) of the increment for the statement and (b) a pointer to an EXPR record (expr)
  representing the end value of the loop. It returns an EXPR record representing the
temporary expression. The semantic actions are as follows:
  - If expr->ex_kind != Cons, the compiler must declare an implicit variable of the
    appropriate type. Generate code to move the value represented by expr into this
    new location and free expr. Call this new temporary T.
  - Generate a jump around the increment. This is discussed below.
  - Print label1.
  - Return expr.

- forby: This function has three parameters: (a) a pointer to an EXPR record representing
  the index variable (index), (b) a pointer to an EXPR record representing the temporary
  variable for the bound of the loop (bound), and (c) a pointer to a constant record
  representing the increment/decrement. It returns an integer representing a label. The
  semantic actions follow:
    - Generate instructions to increments/decrements index.
    - Generate label (label).
    - Generate code to compare the index variable (index) and bound.
    - Generate a conditional jump on greater than or less than depending on the sign of
      the parameter.
    - Return label.

- forstmt: This function has two parameters both representing labels (label1 and label2).
The function does not return a value. The semantic actions follow:
  - Generate an unconditional jump to label1.
  - Print label2.
  - Free all records and registers that are not now needed.

This completes the treatment of the loops. It should be evident that their handling is rather
straightforward, with the possible exception of the for statement. The templates provided for you
should be essential in understanding the rationale behind the various choices.
User Defined Types

This chapter deals with the built-in functions of Modula-2 as well as referencing elements in structured data types.

The Built-In Functions chr and ord

The handling of the functions chr and ord is very simple. The function chr converts an expression of type integer, cardinal, Boolean, or character to an expression of type character, with the same value. It is not an error if the value of the number is out of the valid ASCII bounds [0, 127]. The implementation of this operation requires no MIPS instructions if the size of the parameter is the same as a character. If the sizes are different, code must be generated. Type checking must be done on the argument as well as assuring there is only one argument to the function. The type of the resulting expression is changed to char_type.

Similarly, the function ord is a conversion from an expression of type integer, cardinal, Boolean, character to an expression with the same value and type integer. The implementation of this operation requires no MIPS instructions if the size of the parameter is the same as an integer. Otherwise, code must be generated. Type checking must be done on the argument as well as checking that there is only one argument. The resulting expression always is of type integer.

| Expression : ...
| CHRSY LEFTPARENSY Expression RIGHTPARENSY
| ORDSY LEFTPARENSY Expression RIGHTPARENSY
| ...

Figure 0.1 Productions for built-in functions

The determination of the precise sequence of actions needed for the productions Figure 0.1, including type checking, code generation, and putting the correct information onto the semantic stack, are left to the reader.

Referencing Structured Types

This section outlines the way in which structured data types are referenced in Modula-2.

Referencing Record Fields

Generating code to handle references of record fields in expressions is relatively easy to accomplish. Here, and also in the handling of array references described next, the compiler needs to perform arithmetic on the address of the operands, or generate code that performs such address manipulation during execution. There are several schemes that can accomplish our
objectives. We try to present here a compromise between simplicity and efficiency of the generated code.

| Designator : ... |
| Designator DOTSY IDENTSY |
| \{$$ = recordref($1, $3);$$} |

Figure 0.2 Production for record reference

The production that describes the referencing of record fields in expressions is shown in Figure 0.2. The corresponding semantic actions for record references are outlined in Figure 0.3. This is all that is needed to generate code for references to records in expressions. You are reminded, again, that the schemes presented thus far are described in their full generality and need no changes whatsoever as you keep adding more and more features to the compiler.

Array References in Expressions

Array references in expressions are somewhat more complex to handle than record references, but still quite manageable. Remember that array references are specified via the production shown in Figure 0.4. The effect of the semantic actions for this production are roughly as follows, assuming that the construct is type compatible: Let us suppose that the value of Expression is $I$, the low bound of the array is $L$, the size of the component type of the array is $S$, and the address of the LValue in the right-hand-side (representing the beginning of the array) is $A$. Then, the compiler must compute the address $B$ of the new LValue (representing the address of the referenced array element). The value for $B$ must satisfy one of the following (equivalent) formulas:

$$ B = A + (I - L) * S = A + IS - L * S $$

The compiler computes the quantity $B$ via a combination of compile-time computations and generation of code that completes the computation at execution time. In general, the best combination depends on the specific attributes of the quantities $A$, $I$, $L$, and $S$ above. Of particular importance is whether these quantities are known at compile-time or not. In general, the more of these quantities that are known by the compiler, the better quality code it can generate. As a matter of fact, the quantities $S$ and $L$ above are always constant in Modula-2 (and also in Pascal and C, among others) but not in Algol or PL/I. The value of the quantity $I$ is usually not known by the compiler. On the other hand, the quantity $A$ is constant for global variables, and has a constant component for local variables and value parameters. In any event, the semantic actions for arrayref are outlined in Figure 0.5 (covering three pages).
EXPR *recordref(EXPR *e, char *s)
{
  TYPE *typer;
  ID *f; EX_KIND kindr;
  int invr, offset, reg;
...
  /* Check that e represents a record. If not, issue a message and quit. */
  /* Traverse the list of fields pointed to be e->ex_type->ty_form.ty_record */
  /* looking for one that has name s. If such a field is found, set */
  /* f to point to it, and set typer to f->id_type. */
  /* Otherwise, issue an appropriate message and quit. */
  /* Calculate the address of the field from the address of the record */
  kindr = e->ex_kind;
  invr = e->ex_inv;
  offset = f->id_addr;
  if (offset != 0) /* see if you can figure out why */
  {
    switch (kindr)
    {
      case LocalV:
        invr += offset;
        break;
      case GlobalV:
      
    }
  
  
}

Figure 0.3  Outline for record reference

LValue : LValue LBRACESY Expression RBRACESY
{$$ = arrayref($1, $3);}

Figure 0.4  Production for array reference
EXPR *arrayref (EXPR *base, EXPR *index)
{
    TYPE *typer; EX_KIND kindr;
    int invr, reg, size, low;

    ... /* Check that base represents an array. If not issue a message and quit. */
    ... /* Check if the type of the array index of base is the same type as index. */
    ... /* If not issue an appropriate message and quit. */

    typer = base->ex_type->ty_form.ty_array.ElementType;
    size = typer->ty_size;
    low = base->ex_type->ty_form.ty_array.IndexType->ty_form.ty_subrange.min;

    /* Multiply index and low bound by size */
    if (size != 1) /* figure out why */
    {
        low *= size;
        switch (index->ex_kind)
        {
            case Cons:
/* Shift index by the value of low bound */

if (low != 0)
{
    switch (index->ex_kind)
    {
        case Cons:
            index->ex_inv = index->ex_inv - low;
            break;
        default:
            ... /* Generate code to subtract low from index leaving the */
            ... /* result in register reg. */
            index->ex_kind = RegisterV;
            index->ex_inv = reg;
            break;
    } /* switch */
} /* if */
Procedures and Functions

This chapter describes the semantic actions for the invocation of procedures and functions. The material presented here is to be understood in conjunction with the material describing the run-time storage structure of Modula-2 programs, presented in Section 0. Before we begin a detailed presentation, it is instructive to describe the overall approach in intuitive terms. For the sake of brevity, when in this chapter we mention procedures we really mean both procedures and functions, unless indicated otherwise.

Outline

When a procedure is called, a number of hidden actions take place that are concerned with the setting up of the activation record, the transmission of parameters, the creation of linkages for nonlocal referencing, and similar “housekeeping” activities. These actions must take place before the actual code for the statements in the body of the procedure is executed. Commonly, the compiler inserts a block of code to perform these actions, called the *prologue*, at the start of
the executable code block for the procedure. On termination of a procedure, a similar set of housekeeping actions is required to return results and free the storage for the activation record. An epilogue is a set of instructions inserted by the compiler at the end of the executable code block to perform these actions. Thus prologue and epilogue instructions for a procedure usually handle much of the detail involved with procedure call and return operations.

When a procedure is invoked, the compiler must do the following. First, when the procedure name is seen, the compiler must search the symbol table for that name, in order to ascertain what procedure it represents, and retrieve the information that is needed to translate the call. That information is stored on the semantic stack. After that, one after another, the expressions constituting the arguments are evaluated. The compiler must ensure for each one that it conforms to the corresponding formal parameter of the procedure, and must generate code to push its value (or address) on the run-time stack. When all the arguments are seen, the compiler must generate a procedure call (the MIPS jal instruction) to that procedure. The implementation of the return statement is quite straightforward, and simply involves the generation of the epilogue code and the generation of the return instruction. For functions only, the return statement must also move the return value to the appropriate location. With this outline as a guide, we next proceed to describe the semantic actions in some detail.

**Activation Records**

This section describes how the activation record is built at procedure invocation and how the space in the activation record is reclaimed when returning from the procedure. Building the activation record takes place before the procedure is called and also in the prologue code of the procedure. The epilogue code takes care of reclaiming the space of the activation record.

**Building the Activation Record**

Code to build the activation record is not found in a single place in the executable program. In fact, it is found in two different places. Remember that the activation record (for our purposes) consists of the parameters, control information, and local variables. Let us now describe how the activation record is built.

At the time of the procedure call, code is generated to push the values (or addresses) of the actual parameters on the run-time stack. This starts the building of the activation record. When the procedure call is actually performed, prologue code is executed that finishes the building of the activation record. The prologue code should do the following:

- Save the local registers.
- Generate the label for the procedure.
- Push the return address onto the stack.
- Push the frame pointer (register $fp) onto the run-time stack.
- Update the frame pointer to point to the new activation record.
- Allocate space for the local variables. This is done by decrementing the value of the stack pointer (register $sp) by the size of the local data area.
After the prologue is executed, the new activation record is complete and the procedure is ready to execute. The new environment (stack and frame pointers) is now correct.

**Reclaiming the Activation Record**
The activation record is reclaimed by the epilogue code. The epilogue code should do the following:

- Generate label.
- Update the frame pointer to point to the previous activation record. In other words, put the old value of the frame pointer, found in the current activation record, back into register \$sp.
- Replace the value of \$ra with its value on the stack.
- Deallocate the space for the parameters. This is done by incrementing the value of the stack pointer by the size of the parameter area.
- \$j \$ra
- Restore the local registers.

When the return is done, the environment (the stack and frame pointers) have the same values they did before the procedure was called.

**Blocks**
The productions for a block are shown in Figure 0.1. These productions represent the body of a procedure. The only semantic action that takes place is in `beginclause` and that is where the prologue code for a procedure is generated. The other productions take care of any other semantic actions that have to take place.

```
Block : BeginClause StatementSequence ENDSY

; BeginClause : BEGINSY
  {beginclause();}
```

*Figure 0.1 Production for block*

**The Return Statement**
The return statement is denoted in the grammar via the production shown in Figure 0.2. The semantic actions are quite simple to implement, keeping in mind the semantics of the `return` statement. More specifically, the return statement without an expression can appear in procedures or in the main program (where it is treated as a `stop`). Return statements with expressions can appear only in functions, where the value of the expression must be moved into the designated register for the return value. In any event, a MIPS instruction `j` must be generated
by the compiler to the location of the instruction following the call. More specifically, the function returnstmt has a single parameter (a pointer to an EXPR record) and does not return a value. The semantic actions are as follows:

- If the current procedure is a function, generate code to move the value of the return expression into the appropriate register.
- Generate the epilogue code.
- Generate a MIPS return instruction.
- Free any unused registers and records.

```plaintext
ReturnStatement  : RETURNSY OptExpression
                   {returnstmt($2);}
```

*Figure 0.2 Production for return statement*

Implicit in the treatment of the return statement, as described in the semantic actions above, is the fact that the return values for functions cannot be arrays or records. That eliminates the problem of having to worry about block-moves, allows us to return all values in registers, and also simplifies other aspects of the compiler. This restriction, if you recall, has already been enforced in the semantic actions implementing function definitions.

```plaintext
ProcedureCall : IDENTSY LPARENSY ActualParameters RPARENSY
               {proccall($1, $3);}

FunctionCall  : IDENTSY LPARENSY ActualParameters RPARENSY
               ;
```

*Figure 0.3 Productions for procedure and function calls*

**Procedure and Function Calls**

The invocation of procedures and functions is specified via the productions in Figure 0.3. The semantic actions for the two functions must now be described.

- proccall: This function has two parameters: (a) a pointer to a string (name) and (b) a pointer to a list of EXPR records representing the actual parameters (exprlist). It does not return a value. The semantic actions are:
  - Search the symbol table for name (call the ID record str_ptr). If not found, issue an appropriate message.
  - Check that str_ptr represents a procedure. If not, issue an appropriate message.
Call actualparam with a list of actual parameters (exprlist) and a list of formal parameters (st_ptr->id_next).

Generate the jal instruction.

- **funccall:** This function has two parameters: (a) a pointer to a character string (name) and (b) a pointer to a list of EXPR records representing the actual parameters (exprlist). It returns an EXPR record representing the result. The semantic actions are as follows:
  - Search the symbol table for name (call the ID record st_ptr). If not found, issue an appropriate message.
  - Check that st_ptr represents a function. If not, issue an appropriate message.
  - Call actualparam with a list of actual parameters (exprlist) and a list of formal parameters (st_ptr->id_next).
  - Generate the jal instruction.
  - Create and return an EXPR record representing the result of the function call.

- **actualparam:** This function has two parameters: (a) a pointer to a list of EXPR records (actual parameters) and (b) a pointer to a list of ID records (formal parameters). It does not return a value. Its semantic actions are:
  - For each element of the actual parameter list and each element of the formal parameter list do the following (taking one element from each list at a time):
    - Compare the types of each to see that they are the same.
    - If they are not, issue an appropriate message.
    - If the formal parameter is an RParameter, generate code to push the address of the actual parameter on the run-time stack.
    - If the formal parameter is a VParameter, generate code to push the value of the actual parameter on the run-time stack.
    - Free any registers and records no longer needed.

This completes the description of the semantic actions for invocation of procedures and functions.

**Referencing Parameters**

One final issue must be addressed before this chapter is complete. Parameters are referenced in much the same way as are local and global variables already discussed. The only exception is parameters that are passed by reference (RParameter types). In order to get the value of reference parameters, the address first has to be loaded into a register. Then, the value can be loaded using the address in that register. In other words, two load instructions must be generated to load a reference parameter.

The same sort of thing has to take place in order to store a value into a reference parameter. The address must first be loaded into a register. Then the address in that register is used to store to the value. In other words, a load and then a store must be generated to store a reference parameter. Work this out and make sure you understand these ideas.

**Predefined Procedures/Functions**

Predefined procedures and functions are not discussed further. They are relatively easy to implement and the details are left to the reader.