

CSCI 447 – Spring 2001

Yacc Assignment

Professor: Muhammed Mudawwar
Due Date: Tuesday, May 22, 2001

Objectives:

- 1- To develop a Yacc specification for a given grammar.
- 2- To translate into 3-address code.
- 3- To generate symbol tables

The M Language

The language that will be used in this assignment is the same M language that was used in the previous assignment

Grammar Rules:

The following is the grammar of the M language.

```
ModuleUnit  →  module id {VarDecl} {FuncDefn} end [module] [id] ;
VarDecl     →  var id { , id } : TypeExpr ;
TypeExpr    →  integer
               →  real
               →  char
               →  boolean
               →  array { intconst , TypeExpr }
FuncDefn    →  function id ( [FormalList] ) [ : TypeExpr ]
               {VarDecl} {Stmt} end [function] [id] ;
FormalList  →  Formal { ; Formal }
Formal      →  [ Mode ] id { , id } : TypeExpr
Mode        →  in
               →  out
Stmt        →  Object := Expr ;
               →  if Expr then {Stmt} end [if] ;
               →  if Expr then {Stmt} else {Stmt} end [if] ;
               →  while Expr do {Stmt} end [while] ;
               →  FuncCall ;
FuncCall    →  id ( )
               →  id ( Expr { , Expr } )
Object      →  id {Suffix}
               →  result {Suffix}
Suffix      →  [ Expr ]
Expr        →  Expr or Expr
Expr        →  Expr and Expr
Expr        →  Expr relop Expr
Expr        →  Expr addop Expr
```

```

Expr      →  Expr mulop Expr
Expr      →  UnaryOp Expr

Expr      →  Object
          →  intconst
          →  realconst
          →  charconst
          →  boolconst
          →  strconst
          →  FuncCall
          →  ( Expr )

UnaryOp    →  addop
          →  not

```

Yacc Specification and Operator Precedence:

Convert the above grammar into a conflict-free Yacc specification. Notice that the grammar rules for *Expression* are ambiguous. Therefore, you should remove this ambiguity by assigning priorities to the operators. The **or** operator has the least priority, then the **and** operator, then the relational operators **relop** (<, <=, >, >=, =, <>), then the **addop** operators (+, -), then the **mulop** operators (*, /, **mod**), then the *unary operators* (+, -, **not**). Most operators are left associative. However, the unary operators are right associative and the relational operators are non-associative.

Syntax Error Handling:

If a syntax error occurs while parsing, a meaningful error message should be reported specifying the line number. Compilation can terminate at the first syntax error. Error recovery is not required. However, you are invited to recover from syntax errors if you have time.

Symbol Tables:

All declared identifiers should be entered in symbol tables. Multiple symbol tables should be used for a given module. One symbol table is dedicated for all global identifiers. Symbol entries are either global variables or functions. A symbol entry for a global variable should store its kind as **GVAR**, its name, and its declaration line number in the source file. A symbol entry for a function should store its kind as **FUNC**, its name, its line number where it is defined, and a pointer to a symbol table for that function. Each function should have its own symbol table. Symbol entries in function symbol tables are formal parameters, local variables, and function results. A symbol entry for a formal parameter stores its kind as **IN** or **OUT**, its name, and its line number. A symbol entry for a function result stores its kind as **OUT**, its name as **result**, and its line number where the result type appears in the source file. A local variable symbol entry stores its kind as **LVAR**, its name, and its line number. Hashing techniques should be used to speed up the insert and lookup functions.

The type attribute of each global variable, local variable, formal parameter, or function result is NOT required in this assignment because no type checking is necessary.

Literal Table:

One literal table should be used for all literal constants in a program, regardless of where they appear. A constant value should appear exactly once in a literal table. Look up the literal table before inserting a new literal constant. Hashing techniques should be used to improve the speed of insert and lookup functions. There are five kinds of entries in the literal table. **INT** is used for integer literals, **REAL** is used for real literals, **BOOL** is used for Boolean literals, **CHAR** is used for character literals, and **STR** is used for string literals. In addition to its kind, the value of a literal constant should also be stored.

Translation into Three-Address Code:

Here are the three-address instructions needed to translate programs written in the M-language:

- 1 Assignment instructions of the form: **x := y op z**, where *op* is a binary arithmetic or a logical operation.
- 2 Assignment instructions of the form: **x := op y**, where *op* is a unary operation
- 3 Copy statements of the form: **x := y**
- 4 Variables and literal constants in three-address instructions are entries in symbol and literal tables.

For example, the assignment statement:

a := not (b > 'A' and c < 3) is translated into:

```
(TEMP 1) := (LVAR b) > (CHAR 'A')
(TEMP 2) := (IN c) < (INT 3)
(TEMP 3) := (TEMP 1) and (TEMP 2)
(TEMP 4) := not (TEMP 3)
(GVAR a) := (TEMP 4)
```

where *a* is global variable, *GVAR*, in the global symbol table, *b* is a local variable, *LVAR*, in the local function symbol table, and *c* is an input parameter, *IN*, in the local symbol table as well. (*TEMP i*) is a compiler-generated temporary. Temporaries are identified by number and are entered in the local function symbol table. Temporaries are numbered starting at 1 in each function.

The following is a list of operators that can appear in a source file:

```
+      -      *      /      mod  or   and  not
<      <=     >      >=     =      <>
```

- 5 Unconditional jumps are of the form: **goto (LABEL L)**. A label (**LABEL L**) can precede instructions.
- 6 Conditional jumps are of the form:

```
if x goto (LABEL L) or
ifnot x goto (LABEL L).
```

The **if** and **while** statements are translated using the conditional and unconditional jumps. For example, the following **if** statement:

```
if x >= y then max := x; else max := y; end if;
```

is translated into:

```
(TEMP 1) := (IN x) >= (IN y)
ifnot (TEMP 1) goto (LABEL 1)
(OUT max) := (IN x)
goto (LABEL 2)
(LABEL 1)
(OUT max) := (IN y)
(LABEL 2)
```

The following **while** statement:

```
while i <= n do sum := sum + i; i:= i+1; end while;
```

is translated into:

```

goto (LABEL 2)
(LABEL 1)
(LVAR sum) := (LVAR sum) + (LVAR i)
(LVAR i) := (LVAR i) + (INT 1)
(LABEL 2)
(TEMP 1) := (LVAR i) <= (IN n)
if (TEMP 1) goto (LABEL 1)

```

- 7 For array indexing, we will need the following 3-address instructions:

```

x := y [] i
x [] i := y

```

The *opcode* of the first instruction is `:=[]`, while the *opcode* of second is `[]:=`.

For example, `b[2*i] := a[j+3] * c` is translated into:

```

(TEMP 1) := (LVAR j) + (INT 3)
(TEMP 2) := (IN a) [] (TEMP 1)
(TEMP 3) := (TEMP 2) * (LVAR c)
(TEMP 4) := (INT 2) * (LVAR i)
(OUT b) [] (TEMP 4) := (TEMP 3)

```

For two-dimensional and multi-dimensional arrays, we need to keep track of their dimension sizes in the symbol table. For example if *a* is a matrix with 10 rows and **20** columns, and *b* is a matrix with 15 rows and **5** columns, and if matrices are stored in row-major order, then `b[j+2][i] := a[i][j]` is translated as:

```

(TEMP 1) := (LVAR i) * (INT 20)
(TEMP 2) := (TEMP 1) + (LVAR j)
(TEMP 3) := (IN a) [] (TEMP 2)
(TEMP 4) := (LVAR j) + (INT 2)
(TEMP 5) := (TEMP 4) * (INT 5)
(TEMP 6) := (TEMP 5) + (LVAR i)
(OUT b) [] (TEMP 6) := (TEMP 3)

```

Two-dimensional and multi-dimensional arrays are not required in this assignment, but you are invited to handle them if you have time.

- 8 To translate functions and function calls, we need a **FUNCTION** label, a **call** instruction, a **param** instruction, and a **return** instruction.

For example, the following function:

```

function f(m,n:integer):integer
  var k:integer;
  if n <= 0 then
    result := 1;
  else
    k := f(m-1,n/2);
    result := f(k,m/2);
  end if;
end function f;

```

is translated into:

```

(FUNCTION f)
(TEMP 1) := (IN n) <= (INT 0)
ifnot (TEMP 1) goto (LABEL 1)
(OUT result) := (INT 1)
goto (LABEL 2)
(LABEL 1)
(TEMP 2) := (IN m) - (INT 1)
(TEMP 3) := (IN n) / (INT 2)
param (TEMP 2)
param (TEMP 3)
(LVAR k) := call (FUNCTION f)
(TEMP 4) := (IN m) / (INT 2)
param (LVAR k)
param (TEMP 4)
(OUT result) := call (FUNCTION f)
(LABEL 2)
return

```

The function **result** is represented as an **OUT** parameter. A function **result** should be entered in a function symbol table as an **OUT** symbol. Since **result** is a reserved word, it will not conflict with other identifiers.

Report:

A detailed report should be written explaining your design and implementation. Specifically, you need to discuss the Yacc specification, grammar attributes, semantic actions, and the data structures used for the implementation of the 3-address code.

To Submit:

- 1 The report document.
- 2 Place all your files (lex files, C/C++ files, executable program, test inputs and outputs) in one directory. Name this directory according to your user name. Tar this directory into a file, uuencode the tar file, and mail it to cs447. To do this, execute the following command sequence:

```

mv your_assignment_directory username

tar cvf username.tar username

uuencode username.tar username.tar > username.uu

mail -s "parser from your full name" cs447 < username.uu

```

Grading

Your grade will be divided into the following components:

- 1 Correctness and output
- 2 Yacc specification, implementation details, and documentation
- 3 Intermediate code, implementation details, and documentation
- 4 Report Document