CSCI 447 – Fall 2000
Recursive-Descent Parsing

Professor: Muhammed Mudawwar
Due Date: Monday, November 20, 2000

Objectives:

1- To develop a recursive-descent parser for a given grammar.
2- To generate a syntax tree as an intermediate representation.
3- To handle syntax and some semantic errors.

The M Language

The language that will be used in this assignment is a small language, called the M language (where M stands for the Mudawwar language ☺ … or perhaps the Module language). A compilation unit in the M language is a module that consists of variable declarations and function definitions. Variables declared outside functions have the scope of the entire module and can be used in any function, while those declared inside functions are local variables. A special function, called *main*, specifies the beginning of program execution. An example of a module written in the M-language is shown below:

```
module example
  var x:integer;       -- global var in module example
  function main()
    var y:real;        -- local var in function main
    read(x,y);         -- predefined function for reading input
    write(2*x+y);      -- predefined function for writing output
  end function main;
end module example;
```

The types supported in the M language are **integer**, **real**, **char**, **boolean**, and **array**. There is no facility to define new types. An array type is parameterized by size and element type, where braces { . . . } enclose the parameters of a type. For instance, **array**{10, **char**} is an array of 10 characters and **array**{2, **array**{5, **real**}} is a 2D array with 2×5 real elements. Arrays begin at index 0 and they are stored in row major order.

Functions include variable declarations and statements. A function has an optional result type. If the result type is not specified, the function will be a procedure. However, the keyword **procedure** is not used in this language. Formal parameters have 2 modes: **in** parameters are input parameters to a function, while **out** parameters are output or result parameters. An **in** parameter should be read only. An **out** parameter should be written only. If the *mode* of a formal parameter is not specified, it defaults to **in**. The result of a function is specified using the **result** keyword. A function **result** is equivalent to an **out** parameter. Function overloading is NOT allowed. The following is an example of a function:

```
function f(in x,y: real; out sum:real):real
  sum := x+y;
  result := x*y;
end function f;
```

Grammar Rules:

The following is the grammar of the M language. Terminal symbols (tokens) are in **bold**. Non-terminal symbols appear in *italic*. Optional symbols are enclosed in ⟦ and ⟧. They may or may not appear in a program. Repetitive sequences are enclosed in ⦃ and ⦄. Repetitive sequences may be repeated zero, one, or more times. For example, a *ModuleUnit* begins with **module** keyword, followed by an **id**entifier, followed by a list of variable declarations, ⦃*VarDecl*⦄, followed by a list of function definitions, ⦃*FuncDefn*⦄. It is terminated with the **end** keyword, followed by the

optional **module** keyword, ⟦**module**⟧, followed by an optional **id**entifier, ⟦**id**⟧, followed by semicolon. The optional identifier name appearing at the end of a module must match the identifier name appearing in the module header.

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 → *AndExpr* ⟪ **or** *AndExpr* ⟫
AndExpr → *RelExpr* ⟪ **and** *RelExpr* ⟫
RelExpr → *AddExpr* ⟦ **relop** *AddExpr* ⟧
AddExpr → *MulExpr* ⟪ **addop** *MulExpr* ⟫
MulExpr → *UnaryExpr* ⟪ **mulop** *UnaryExpr* ⟫
UnaryExpr → ⟪ *UnaryOp* ⟫ *Primary*

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

UnaryOp → **addop**
→ **not**

Lex Specification:

The M language is not case sensitive. Identifiers and keywords can appear in lower or uppercase. Do the necessary modifications to your lex specification so that it can now be used with the new grammar. Notice that few keywords have changed. New keywords, such as **module** and **result** are introduced in this grammar. They replace the **program** and **return** keywords that were in the lex specification. You lex scanner should also recognize the left and right curly braces **{** and **}** as tokens.

Syntax Error Handling:

If a syntax error occurs while parsing, an error message should be reported specifying the line number, the lookahead token that caused the error, and the expected token. 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 expression of each global variable, local variable, formal parameter, or function result is NOT required in this assignment because no type checking is necessary at this time.

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.

Static Semantics Checking:

The following static semantic rules should be checked. Error messages should be reported when these rules are violated, but parsing and translation should continue. Error message should specify the line number and describe the error that occurred.

1 An optional identifier appearing at the end of a module (or function definition) must match the name of module (or function).

2 All variables must be declared before they can be used. Functions should be defined before they can be called.

3 Identifiers cannot be declared more than once within the same scope. However, they can be re-declared in different scopes (for example, within two different functions, or as global in a module and local to a function). Function names cannot be overloaded.

There are many other semantic rules that need to be checked. However, they are not part of this assignment.

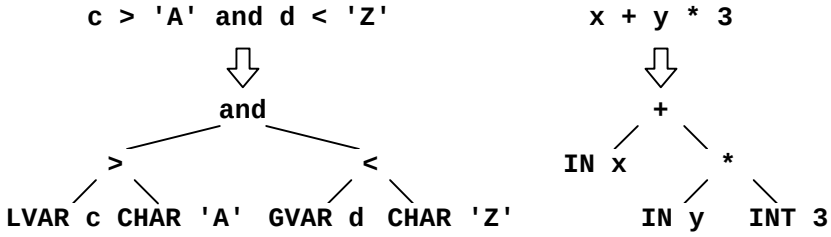
Translation into a Syntax Tree:

A syntax tree is the intermediate representation for the M language. Expressions and statements should be translated into a syntax tree.

1 An operator node stores the kind of operator: +, –, *****, /, **mod**, **and**, **or**, **not**, <, <=, etc. It should have a left and a right pointer, and should also contain the line number at which the operator appears. Unary operators use only the left pointer, while binary operators use both pointers. Type information is not necessary at this stage, but will be added later.

2- Variables and literal constants in a syntax tree are entries in symbol and literal tables.

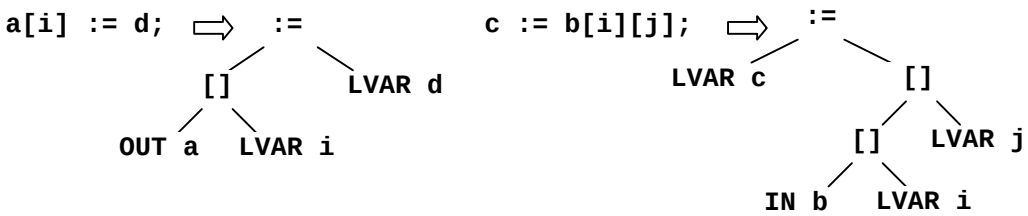
3- Expressions are translated into syntax trees as shown below. The following is the translation of two expressions, where *c* is a local variable, *d* is a global variable, and *x* and *y* are **in** parameters.



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

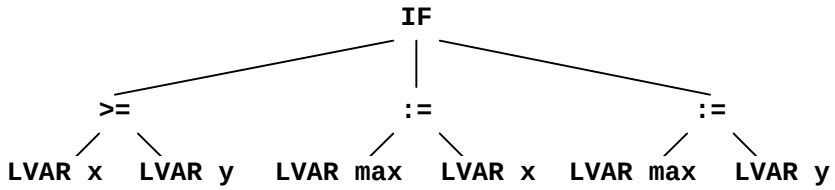
```
+     -     *     /     mod   or    and   not
<     <=    >     >=    =     <>
```

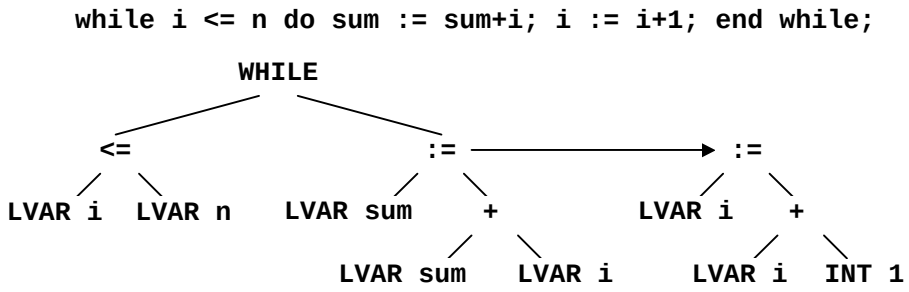
4- For indexing, the **[]** operator is used, The left child points to the object to the left of the **[]** operator, while the right child points to the index expression. The result of the **[]** operator is an address. An assignment statement is translated into an assignment operator node. The left child points to the object on the left hand side of the assignment operator. The right child points to the expression on the right hand side. The following are examples on the indexing and assignment operators:



5 The **if** and **while** statements are translated into **IF** and **WHILE** nodes. The **IF** node uses three pointers, a pointer to a Boolean expression, a pointer to the statement list of then-part, and a pointer to the statement list of else-part. The **WHILE** node uses two pointers, a pointer to a Boolean expression and a pointer to a statement list. Statement lists are implemented as linked lists. A *sibling* pointer links all statement trees together. The following examples show the syntax trees of **if** statement and **while** loop.

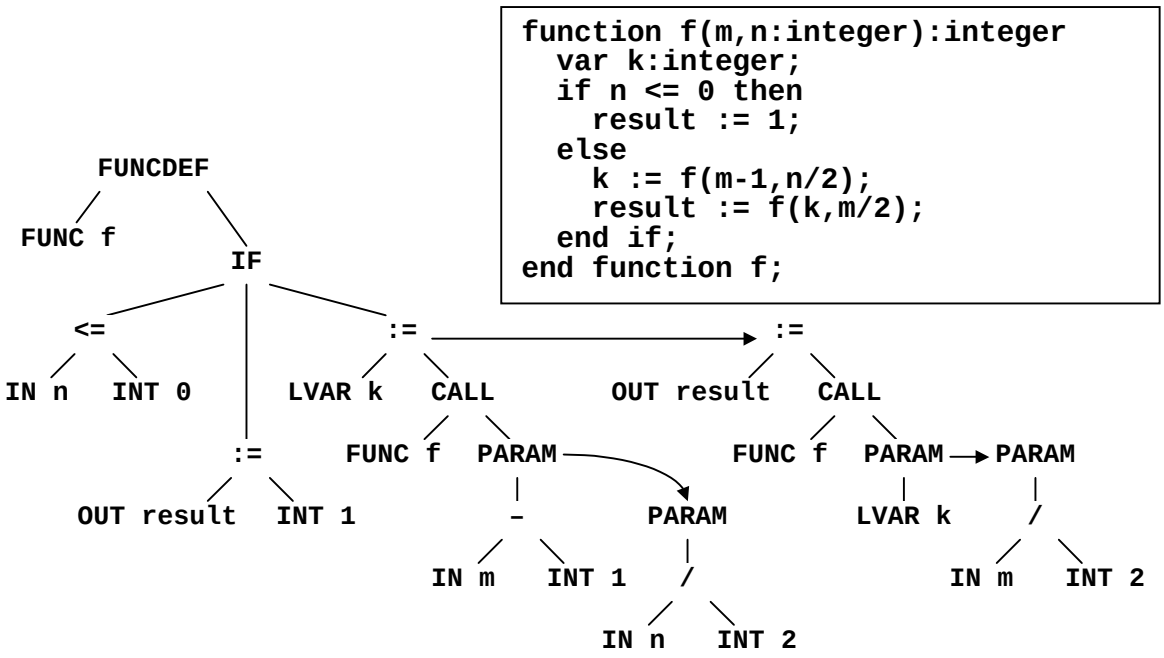
```
if x >= y then max := x; else max := y; end if;
```





6 To translate functions and function calls, we need a **FUNCDEF** node and a **CALL** node. A **FUNCDEF** node has a left pointer pointing to a **FUNC** symbol in the global symbol table, a right pointer pointing to a statement list, and a sibling pointer for linking function definitions in a module. A **CALL** node has a left pointer pointing to a **FUNC** symbol, a right pointer pointing to an actual parameter list, and a sibling pointer in case the function call is a statement. An actual **PARAM** node has a pointer pointing to an expression tree and a sibling pointer for linking **PARAM** nodes.

Consider the following recursive function *f* with one **in** parameter *n*. The **result** of this function is of type **integer**. This function definition is translated into the following syntax tree:



The function **result** is represented as an **OUT** node in the syntax tree. 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.

Symbols may be referenced multiple times in a syntax tree. For example, **FUNC f** appears 3 times in the above tree. **IN n**, **IN m**, **OUT result**, **LVAR k**, **INT 1**, and **INT 2** appear twice. Although the same symbol appears multiple times in the above tree to simplify the drawing, each symbol exists once in the corresponding symbol or literal table. Therefore, a syntax tree is not really a tree but rather a directed acyclic graph (DAG) in the formal sense.

Output the Symbol Tables:

1 For each global variable: output its name and its line number.

2 For each function: output its name, its line number, and the function symbol table containing formal parameters, result, and local variables.

3 For each formal parameter and function result: output its mode, its name, and line number.

4 For each local variable: output its name and its line number.

Output the Syntax Tree:

Output the syntax tree of each function in a parenthesized prefix notation, which corresponds to the preorder traversal. For example, the syntax tree of the function *f* is displayed as shown below (assuming that function *f* starts at line 10). Indentation helps visualizing the structure of the syntax tree. **@***n* indicates a line number and **->** indicates a sibling link.

```
(FUNCDEF @10
  (FUNC f @10)
  (IF @12
    (<= @12
      (IN n @10)
      (INT 0)
    )<=
    (:= @13
      (OUT result @10)
      (INT 1)
    ):=
    (:= @15
      (LVAR k @11)
      (CALL @15
        (FUNC f @10)
        (PARAM @15
          (- @15
            (IN m @10)
            (INT 1)
          )-
        )PARAM
      ->(PARAM @15
          (/ @15
            (IN n @10)
            (INT 2)
          )/
        )PARAM
      )CALL
    ):=
  ->(:= @16
      (OUT result @10)
      (CALL @16
        (FUNC f @10)
        (PARAM @16
          (LVAR k @11)
        )PARAM
      ->(PARAM @16
          (/ @16
            (IN m @10)
            (INT 2)
          )/
        )PARAM
      )CALL
    ):=
  )IF
)FUNCDEF
```

```
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;
```

Report:

A detailed report should be written explaining your design and implementation. Specifically, you need to discuss the parsing functions and what they do (parameters used if any and result returned), the implementation of the syntax tree (tree node structure and functions that allocate tree nodes), the symbol structure, symbol and literal tables, the handling of syntax and semantic errors.

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 Parsing functions, implementation details, and documentation
3 Syntax tree, implementation details, and documentation
4 Symbol structure, symbol and literal tables, implementation details and documentation
5 Handling syntax and semantic errors, error reporting
6 Report Document