CSCI 447 – Summer 2002
Recursive-Descent Parsing

Professor: Muhammed Mudawwar
Due Date: Tuesday, July 16, 2002

Objectives:

1- To develop a recursive-descent parser for a given grammar.
2- To generate a syntax tree as an intermediate representation.
3- To generate symbol and literal tables.
4- 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 ☺). A program is a collection of functions. A special function, called *main*, specifies the beginning of program execution. A predefined function *input* is used for reading input, and a function *output* is used for writing output. *Input* and *output* can take an arbitrary number of parameters. The parameters of *input* must be variables, while the parameters of *output* can be variables, constants, or expressions. A simple program written in the M-language is shown below:

```
const pi:real := 3.14159;           -- global constant
function main() {
  var r:real;                       -- local var in function main
  output("Enter circle radius: ");  -- for writing output
  input (r);                        -- for reading input
  output("Area = ", pi*r*r, '\n');
}
```

Types

The types supported in the M language are **int**, **real**, **char**, **bool**, arrays, and records. Arrays begin at index 0 and are stored in row major order. For example, **int**[10] is an integer array of 10 elements, and **real**[5][7] is 2D array with 5×7 real elements. The sizes of dimensions must be integer literal constants.

Records can be also defined in the M language. Records are defined outside functions. The following are examples of record definitions:

```
record date {
  day   : int;
  month : int;
  year  : int;
}
```

```
record person {
  first : char[15];
  last  : char[15];
  birth : date;
}
```

```
record people {
  count : int;
  list  : person[100];
}
```

Functions and Parameters

A function has an optional list of parameter and an optional result type. There are two modes of formal parameters: **const** parameters are input parameters to a function, while **var** parameters are output parameters. A **const** parameter is a read-only parameter that cannot be modified. A **var** parameter can be read and written. If the *mode* of a formal parameter is not specified, it defaults to **const**.

Const and **var** parameters are passed by reference. **Const** parameters are passed by reference for efficiency purposes to avoid the copying of arrays and records. The only exception is for **const** scalar parameters of type **int**, **real**, **char**, and **bool**, which are passed by value. The **return** statement specifies the result of a function. Function overloading is NOT allowed in this limited version of the M language. The following are examples of function definitions:

```
function power(x:real, const n:int):real {
  -- x and n are const parameters
  -- const keyword is optional for parameters
  if (n = 0) return 1;
  else if (n = 1) return x;
  else if (n < 0)  return 1/power(x,-n);    -- recursive call
  else {
    var p:real := power(x,n/2);             -- recursive call
    if n mod 2 = 0 { return p*p; }
    else { return p*p*x; }
  }
}
function swap(var x,y:real) {
  -- x and y are var parameters
  var temp:real := x;
  x := y
  y := temp
}
```

Blocks and Statements

A block is a sequence of zero or more statements surrounded by curly braces { }. For example, the body of a function is a block. Constants and variables can be declared inside a block and will have the scope of the block. The M language supports an assignment statement for copying, an **if** statement for selection, a **while** statement for repetition, a **return** statement for returning function results, and a function call statement for calling functions.

Operators and Expressions

The M language supports a number of operators. The logical operators are: **or**, **and**, and **not**. The relational operators are: <, <=, >, >=, =, and <>. The addition operators are + and –. The multiplication operators are *****, /, and **mod**. An expression may include variables, constants, function calls, parentheses, as well as all the above operators. The **or** operator has the least precedence. The **and** operator has the next precedence, then the relational operators, then the addition operators, then the multiplication operators, then the unary operators. The unary operators are +, –, and **not**. Parentheses are given the highest precedence.

Operators of the same precedence are evaluated from left to right (left associative). However, unary operators are evaluated from right to left (right associative), and relational operators can't be evaluated left-to-right or right-to-left (non-associative). For example, the expression **`a < b < c`** should cause a syntax error at the second **<**. The proper expression should be **`a < b and b < c`**.

Arrays can be assigned and can be indexed. Array assignment will copy the whole array element by element. The square brackets [ ] are used for accessing array elements. Similarly, record variables can be assigned. The dot operator is used for accessing record elements.

Grammar Rules

The following is the grammar of the M language. Extended BNF notation is used. Terminal symbols (tokens) are in **bold**. Non-terminal symbols appear in *italic*. Optional sequences are enclosed in brackets [ ]. Repetitive sequences are enclosed in curly braces { }. Alternative sequences are separated by vertical bars |. For example, a *Program* consists of an arbitrary number (including zero) of constant declarations, variable declarations, record definitions, and function definitions, according to the first production.

```
Program    → { ConstDecl | VarDecl | RecordDef | FuncDef }
ConstDecl  → const Decl ':=' Expr ';'
VarDecl    → var Decl [':=' Expr] ';'
Decl       → idList ':' TypeExpr
idList     → id {',' id}
TypeExpr   → int | real | char | bool | id
           |  TypeExpr '[' intconst ']'
RecordDef  → record id '{' Decl ';' {Decl ';'} '}'
FuncDef    → function id '(' [FormalList] ')' [':' TypeExpr] Block
FormalList → Formal {',' Formal}
Formal     → [const] Decl | var Decl
Block      → '{' {stmt} '}'
Stmt       → ConstDecl
           |  VarDecl
           |  Block
           |  Object ':=' Expr ';'
           |  if '(' Expr ')' Stmt [else Stmt]
           |  while '(' Expr ')' Stmt
           |  return Expr ';'
           |  FuncCall ';'
FuncCall   → id '(' [ExprList] ')'
ExprList   → Expr {',' Expr}
Object     → id {Suffix}
Suffix     → '[' Expr ']' | '.' id
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
           |  Literal
           |  FuncCall
           |  '(' Expr ')'
Literal    → boolconst | intconst | realconst | charconst | strconst
UnaryOp    → addop | not
```

Lex Specification:

The M language is not case sensitive. Identifiers and keywords can appear in lowercase or uppercase. Do the necessary modifications to your lex specification so that it can now be used with the above grammar.

The other tokens are the same, except that the scanner should recognize and return one additional token, which is the dot operator.

Symbols and Symbol Tables:

There are different categories of symbols. A *name* symbol is a symbol that holds only the name of an identifier. A literal symbol is a symbol that holds the value and the type of a literal constant. A type symbol is a symbol that holds a type name and its attributes. An identifier symbol holds a **const** or a **var** identifier and its attributes, a function symbol is a symbol that holds a function name and its attributes, and a field symbol holds a field name and its attributes. There are also many instances of symbol tables as discussed below. Hashing techniques should be used to speed up the lookup and insertion of symbols.

Name Table:

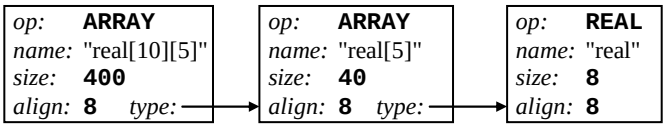
One table should be used for all identifier names, regardless of where these names appear. The scanner should enter an identifier name and return a pointer to its entry. There should be a unique entry for each name no matter in how many scopes it appears.

Literal Table:

One table should be used for all literal constants, regardless of where they appear. A literal constant should appear exactly once. A literal symbol should store the value and type of each literal constant. The type of a literal is a pointer to a type symbol. The type of an integer literal is "int", the type of real literal is "real", the type of a char literal is "char", the type of a bool literal is "bool", and the type of a string literal is "char[*n*]", where *n* is the length of the string literal. The length of a string literal does NOT include the extra surrounding quotes and backslashes in escape sequences.

Type Table:

One type table should be used for all types. A type should appear once in a type table no matter how many times it is used. Each type symbol has a type *operator*, a type *name*, a type *size*, a type *alignment*, and a *pointer* to another type. A type *operator* can be **`INT`**, **`REAL`**, **`BOOL`**, **`CHAR`**, **`ARRAY`**, and **`RECORD`**. The built-in types **`INT`**, **`REAL`**, **`BOOL`**, and **`CHAR`**, should be initially entered in the type table. The parser is responsible of entering new types as they are encountered. For example, **real**[10][5] can be represented by the following list of type symbols.

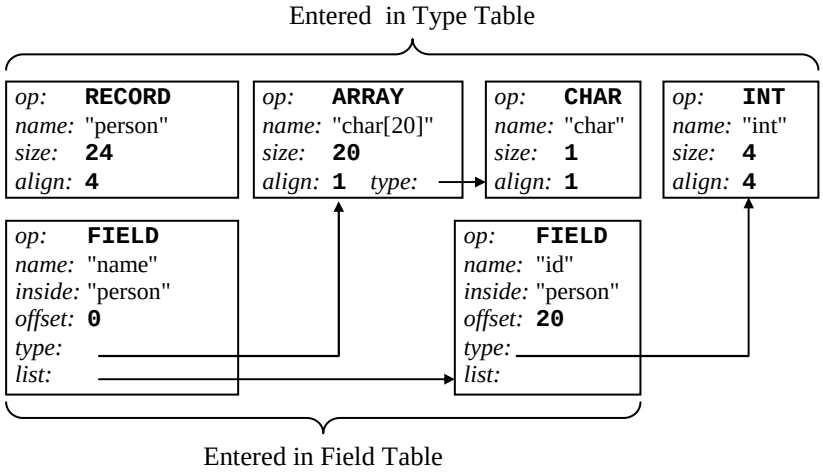


Field Table:

For records, the record name, size and alignment are entered into a type symbol in the type table. However, the field names and their types are entered into a field table. One table is used for all record fields. To have a unique entry for a record field, the field name and the record name are both entered. A **`FIELD`** symbol should contain the field *name*, the record name (*inside*), the field *type*, and the field *offset* (byte offset within its record). The field name and the record name should be hashed together and looked up together.

For example, the record *person* and its fields are represented as shown below:

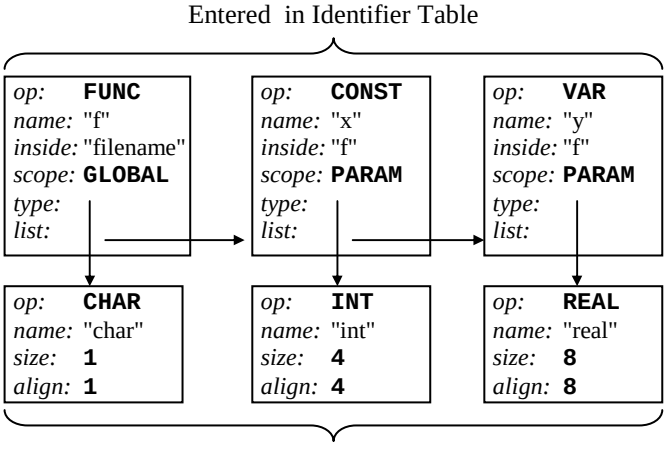
```
record person {
  name: char[20];
  id:   int;
}
```



Identifier Table:

One table is used for all **const** and **var** identifiers as well as functions. This includes global and local variables, named constants, functions, and parameters. An identifier symbol includes the *name*, the *inside*, the *type* and the *scope* of an identifier. The *inside* is the name of the function that contains the local identifier, or the name of the file that contains the global identifier. The scope distinguishes between identifiers having the same name and appearing in different scopes. Identifier symbols at a given scope are removed from the buckets of the hash table upon exit from that scope. However, these symbols are kept linked, using a *list* pointer. The *list* pointer serves to build a linked list of all symbols entered in a given table according to their insertion order. This is not restricted to identifier symbols in the identifier table, but applies to other symbols in other tables as well.

For example, given **`function f(x:int, var y:real):char`**, the function type is the function result type. This type should exist or entered into the type table along with the parameter types. The function symbol and parameter symbols are entered into the identifier table.



Syntax Error Handling:

If a syntax error occurs while parsing, an error message should be reported specifying the line and position number, the lookahead token that caused the error, and the expected token(s). 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.

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. An error message should specify the token line and position number and describe the error that occurred.

1 All variables must be declared before they can be used. Functions should be defined before they can be called. Record types must be defined before they can be used. The *input* and *output* function symbols should be initially inserted into the identifier table to locate them when they are called. Similarly, the built-in types **int**, **real**, **char**, and **bool** should be initially inserted into the type table.

2 Identifiers cannot be declared more than once within the same scope. However, the same identifier name can appear in different scopes. Function names cannot be overloaded (with the exception of the predefined *input* and *output* functions, which can accept an arbitrary number and types of parameters).

3 Formal **const** parameters can be read only inside a function. They cannot appear on the left hand side of an assignment statement. Furthermore, **const** parameters, **const** identifiers, and literals cannot be passed to **var** parameters. Formal **var** parameters must receive **var** identifiers as their actual parameters. **Var** parameters can be read and written and can be passed as actuals to formal **const** parameters.

4 It is illegal to index non-array identifiers. Only arrays can be indexed.

5 It is illegal to use the dot operator with non-record identifiers. Only record fields can be accessed using the dot operator.

Type checking for expressions and function calls is not required in this assignment, but you are invited to do type checking if you have time.

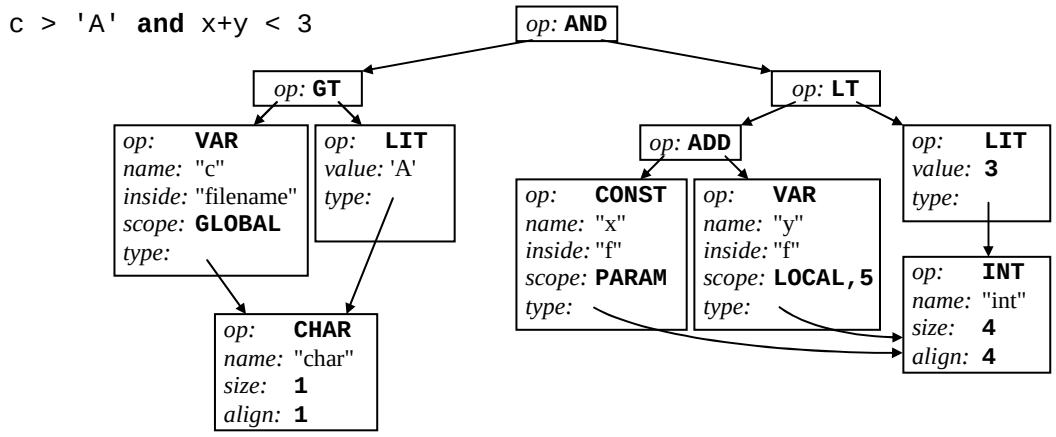
Translation into a Syntax Tree:

A syntax tree is used as an intermediate representation. Expressions and statements should be translated into a syntax tree. There are different categories of nodes:

1 An operator node stores the operator: **ADD**, **SUB**, **MUL**, **DIV**, **MOD**, ... etc. It should have two pointers. Unary operators use only one pointer, while binary operators use two pointers. To check the types of expressions, you need to add type information to each node (optional).

2- A symbol node is an identifier symbol or a literal symbol in a symbol table.

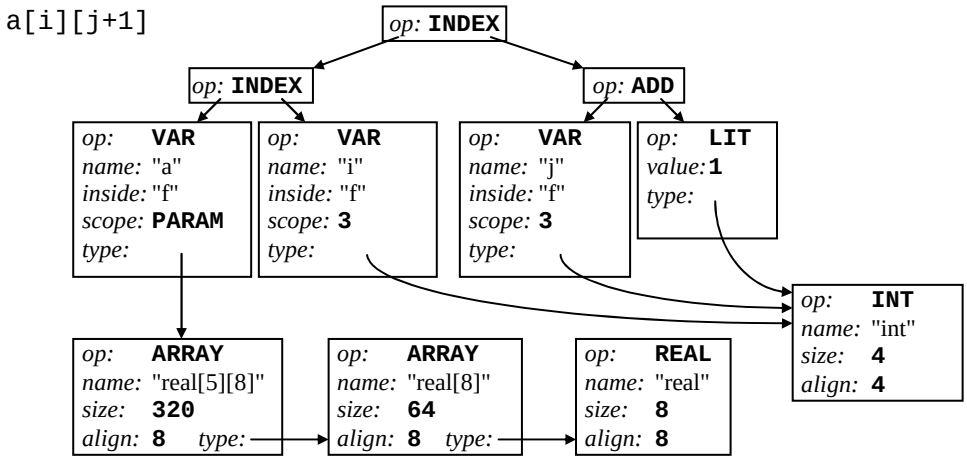
Expressions are translated into syntax trees as shown below. The following is the translation of an expression, where *c* is a **global var** inside *filename*, *x* is a **const parameter** inside function *f*, and *y* is a **local var** inside function *f* nested under scope number 5. The variables, parameters and their types should have been entered in symbol tables. Observe that scope is an integer number, where **GLOBAL** is given the value 1, **PARAM** is given the value 2, and **LOCAL** is given values ≥ 3 to distinguish between various scopes nested under some function *f*. The local scope number begins at 3 upon entry of a new function and is incremented upon entry of a block. All identifiers belonging to a given block should be removed from the buckets of an identifier table upon exit of the block. However, these symbols should not be deleted because they are linked to the generated syntax tree.



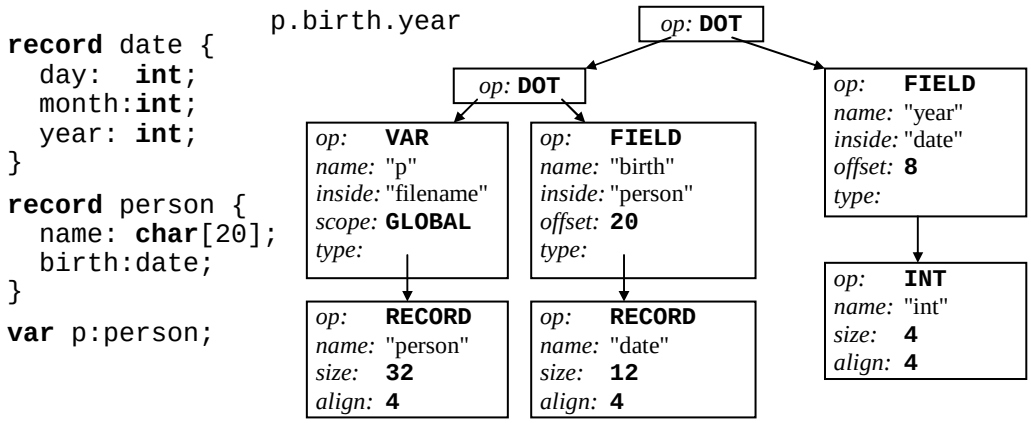
The following is a list of operators that can appear in a source file:

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

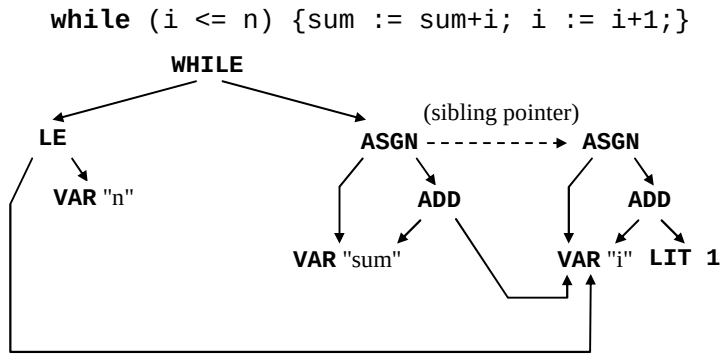
For indexing, the **INDEX** operator is used. The left pointer points at the array object, while the right pointer points at the index expression between brackets. The following is an example on the **INDEX** operator, where *a* is assumed to be of type **real**[5][8].



For field access, the **DOT** operator is used. The following is an example on the **DOT** operator, were *p* is a record variable of type *person*:

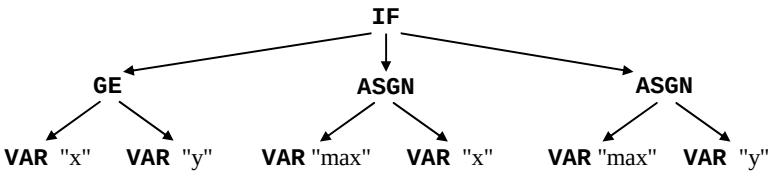


3 For statements, we need statement nodes. Statement nodes have a *sibling* pointer for sequencing. An assignment statement is translated into an **ASGN** node. The left pointer points at the object on the left hand side of the assignment operator, and the right pointer points at the expression on the right hand side. The **while** statement is translated into a **WHILE** node is used. The left pointer points at a Boolean expression, and the right pointer points at a statement sequence. Statement sequences are implemented using the *sibling* pointer. The following is a syntax tree of a **while** statement.



The **if** statement is translated into an **IF** node. The **IF** node requires three child pointers, in addition to a sibling pointer (for sequencing). The left pointer points at the Boolean expression, the middle pointer points at then-part, and the right pointer points at the else-part. The following example shows the syntax tree of an **if** statement. The same **VAR** symbol appears twice for clarity.

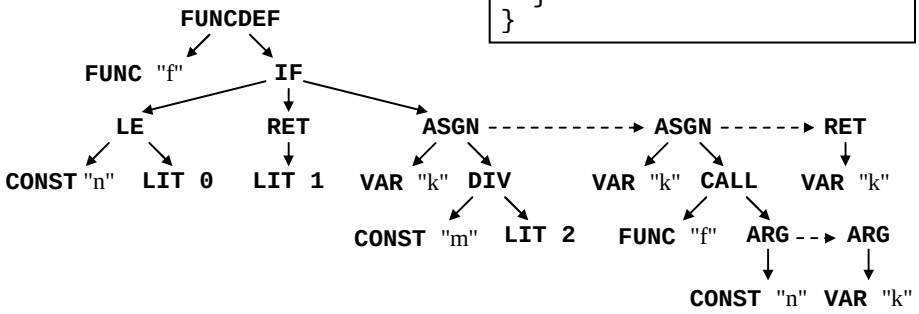
```
if (x >= y) max := x; else max := y;
```



4 To translate function definitions, a **FUNCDEF** node is used. The left pointer points at a **FUNC** (function) symbol and the right pointer points at a statement sequence. To translate a function call, a **CALL** node is used. The **CALL** node has a left pointer pointing at a **FUNC** symbol and a right pointer pointing at an actual **ARG** (argument) list. Each **ARG** node points at an actual argument or parameter. The **return** statement is translated into a **RET** node.

The following recursive function *f* illustrates the translation of functions, function calls, and return statements.

```
function f(m,n:int):int {
  if (n <= 0) return 1;
  else {
    var k:int := m/2;
    k := f(n,k);
    return k;
  }
}
```



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 above function *f* is displayed below. Indentation helps visualizing the structure of the syntax tree. Use curly braces for sequencing, when sibling pointer is not NULL.

```
(FUNCDEF
  (FUNC f int GLOBAL)
  (IF
    (LE
      (CONST n int PARAM f)
      (LIT 0 int)
    )
    (RET
      (LIT 1 int)
    )
```

```
    { (ASGN
        (VAR k int LOCAL4 f)
        (DIV
          (CONST m int PARAM f)
          (LIT 2 int)
        )
      )
      (ASGN
        (VAR k int LOCAL4 f)
        (CALL
          (FUNC f int GLOBAL)
          { (ARG
              (CONST n int PARAM f)
            )
            (ARG
              (VAR k int LOCAL4 f)
            )
          }
        )
      )
      (RET
        (VAR k int LOCAL4 f)
      )
    }
  )
)
```

Output the Symbol Tables:

1 Output the list of names in the name table, each name on a separate line.

2 Output the list of literal symbols in the literal table. For each literal symbol, output its value and its type name on a separate line.

3 Output the list of type symbols in the type table. For each type symbol, output its name, its size, its alignment, and the name of its element type (when the type link is not NULL).

4 Output the list of field symbols in the field table. For each field symbol, output its name, its record name, its offset, and its type.

5 Output the list of identifier symbols in the identifier table. For each identifier symbol, output its class (**CONST**, **VAR**, or **FUNC**), its name, its inside name, its type name, and its scope.

Report:

A detailed report should be written explaining your design and implementation. Specifically, you need to discuss the parsing functions and what they do, the implementation of the syntax tree, the symbol structures for various symbols, symbol tables, the handling of syntax and semantic errors.

To Submit:

1 The report document.
2 A floppy disk containing all your files (source files, executable program, test inputs and outputs).

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 Symbols, symbol tables, implementation details and documentation
5 Handling syntax and semantic errors, error reporting
6 Report Document