CSCI 447 – Spring 2003
Recursive-Descent Parsing

Professor: Muhammed Mudawwar
Due Date: Wednesday, April 30, 2003

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.

Name Table:

One table should be used for all identifier names, regardless of where these identifiers appear. The scanner should enter these names into this table. Memory should be allocated for all entered names. There should be a unique entry for each name no matter how many times it appears in the source file. Hashing should be used for the fast lookup and insertion.

Literal Table:

One table should be used for all literal constants, regardless of where they appear. A literal constant should appear exactly once in the literal table, even if multiple occurrences of the literal constant exist in the source file. There are five categories of literal symbols: **ILIT** is for integer literals, **RLIT** is for real literals, **CLIT** is for character literals, **BLIT** is Boolean literals, and **SLIT** is for string literals. The scanner should enter literals into the literal table. Hashing should use the literal *value* for fast lookup.

Literal symbols have the following attributes:

- *op*: literal operator (**ILIT**, **RLIT**, **CLIT**, **BLIT**, or **SLIT**)
- *value*: integer value (integers, chars, and boolean), real value, or pointer to string in string table
- *size*: size of the literal in bytes

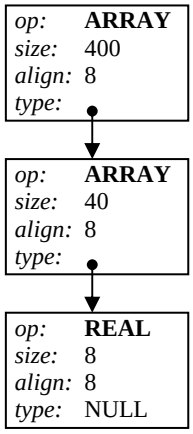
Type Table:

One type table is used for all types. There are seven categories of type symbols: **INT**, **REAL**, **CHAR**, **BOOL**, **ARRAY**, **RECORD**, and **FIELD**. The parser is responsible of entering new types as they are encountered. **RECORD** and **FIELD** names should be also entered into the name table. Hashing should be used for the fast lookup and insertion of type symbols.

Type symbols have the following attributes:

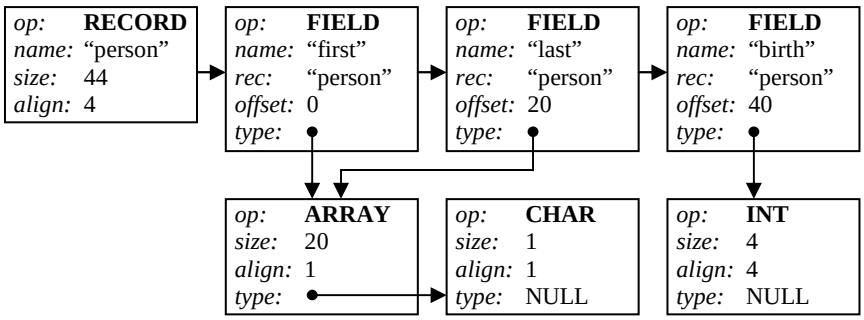
- *op*: type operator (**INT**, **REAL**, **CHAR**, **BOOL**, **ARRAY**, **RECORD**, or **FIELD**)
- *name*: record or field name, a pointer to a name in the name table (NULL otherwise)
- *rec*: enclosing record type (for **FIELD** symbols only)
- *size*: type size in bytes
- *align*: type alignment, or field *offset* in bytes
- *type*: array element type or field type (NULL otherwise)

The basic types: **INT**, **REAL**, **CHAR**, and **BOOL** are represented by one type symbol. The sizes of these types are assumed to be 4, 8, 1, and 1, respectively. Actually, these sizes depend on the underlying architecture. Their alignments are also assumed to be 4, 8, 1, and 1. **INT** and **REAL** objects must be *aligned* in memory. An object at a given memory address is aligned if the address is divisible by its *align* factor. For example, the address of an **INT** object should be divisible by 4 and the address of a **REAL** object should be divisible by 8. For **ARRAY**, its *align* factor is equal to the *align* factor of its element *type*. For **RECORD**, its *align* factor is equal to the maximum **FIELD** *align* factor. **ARRAY** types are represented by linked structures. For example, **real**[10][5] can be represented as (**ARRAY**, S400, A8, (**ARRAY**, S40, A8, (**REAL**, S8, A8))) as shown on the right. Observe that dividing *size* 400 in the first **ARRAY** symbol by *size* 40 in the second **ARRAY** symbol will give 10, which is the number of elements in the first dimension. Similarly, dividing size 40 by 8 will give the number of elements in the second dimension.



RECORD and **FIELD** symbols are inserted in the type table in order, such that from the **RECORD** symbol we can reach all the **FIELD** symbols in order. Alternatively, **FIELD** symbols can have a pointer to their enclosing **RECORD** type symbols. The *rec* field is used for that purpose. Observe that the *rec* field can be a pointer to the **RECORD** symbol, rather than being a pointer to the record name, as shown in the example below.

```
record person {
  first: char[20];
  last:  char[20];
  birth: int;
}
```



For **INT**, **REAL**, **CHAR**, and **BOOL** types, the hashing function should use the type operator, *op*, for fast lookup and insertion. For **ARRAY**, the hashing function should use the *op* and *type* attributes combined to ensure the uniqueness of an **ARRAY** type symbol. For **RECORD**, the hashing function should use the *name* attribute. For **FIELD**, the hashing function should use the *name* and *rec* attributes combined for the fast lookup and insertion of a field symbol.

Object Table:

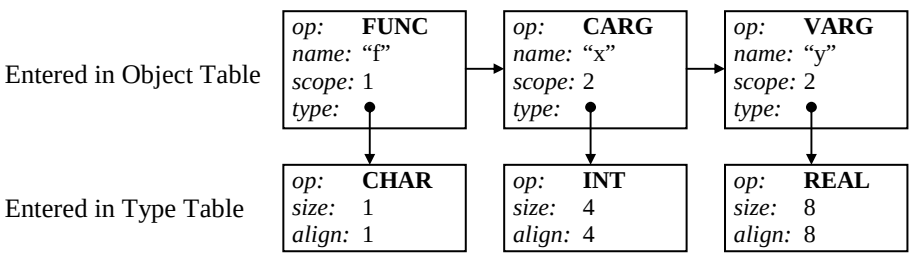
One table is used for all objects: constants, variables and functions. This includes local and global variables, named constants, functions, and parameters. There are five categories of objects that can be entered into this table: **CONST** is for named constants, **VAR** is for variables, **CARG** is for constant arguments, **VARG** is for variable arguments, and **FUNCTION** is for functions. The parser is responsible of entering object symbols into the object table when processing declarations.

Object symbols have the following attributes:

- *op*: object operator (**CONST**, **VAR**, **CARG**, **VARG**, or **FUNCTION**)
- *name*: object name, which is a pointer to a name in the name table
- *line*: line number in the source file
- *pos*: character position in the source file
- *scope*: scope number – a unique number associated with every scope
- *type*: object type, or function result type

The scope distinguishes between objects having the same name and appearing in different scopes. The global scope number is 1. Object symbols are entered in the object table in order such that from a function symbol we can reach all argument symbols in order. In addition, hashing should be used for the fast lookup and insertion of object symbols. For example, a function header is entered as shown below. The *line* and *pos* attributes are not shown here for simplicity. Notice that the function symbol *f* is at scope number 1 because its visibility is global, while the argument symbols *x* and *y* are at scope number 2 because they are visible only in function *f*. The argument symbols, *x* and *y*, should become invisible (when looking up the object table) upon exit from scope number 2.

```
function f(x:int, var y:real):char
```



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.

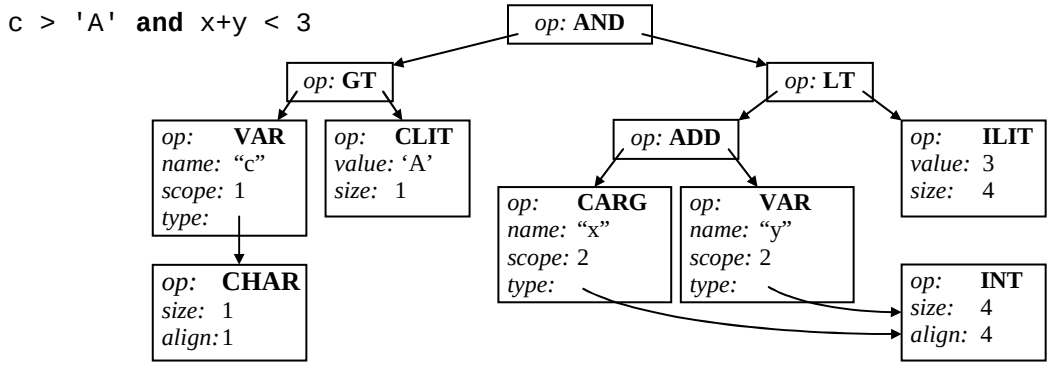
Translation into a Syntax Tree:

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.

2- A leaf node is either an object symbol, a literal symbol, or a field symbol.

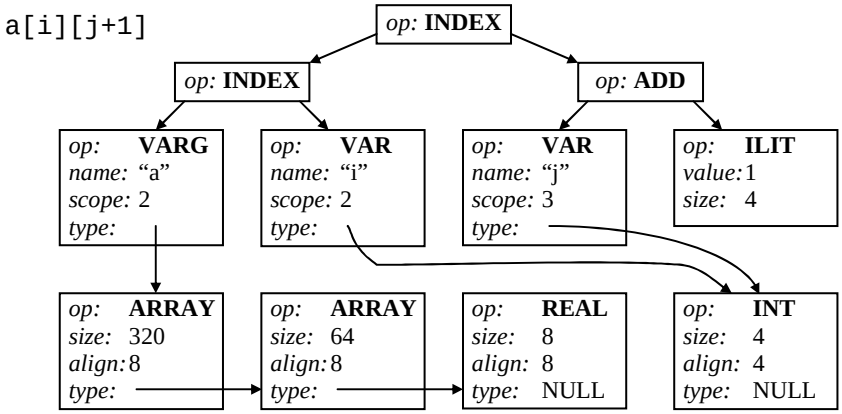
Expressions are translated into syntax trees as shown below. The following is the translation of an expression, where *c* is a global variable with scope number 1, *x* is a constant argument, and *y* is a local variable nested under scope number 2. The variables, parameters and their types should have been entered into the object and type tables.



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

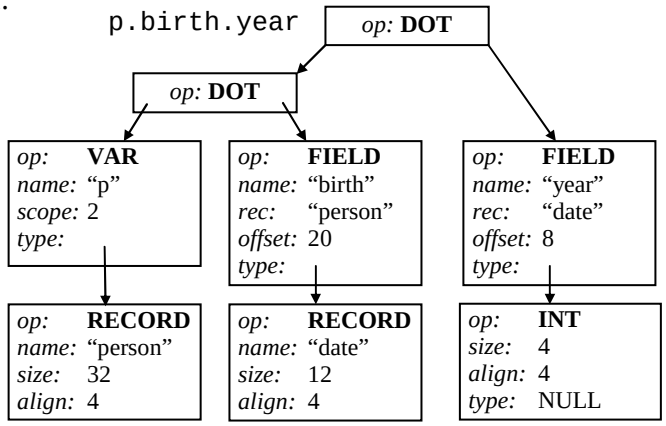
```
+     –     *     /     mod   or    and   not
<     <=    >     >=    =     <>    []    .
```

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].

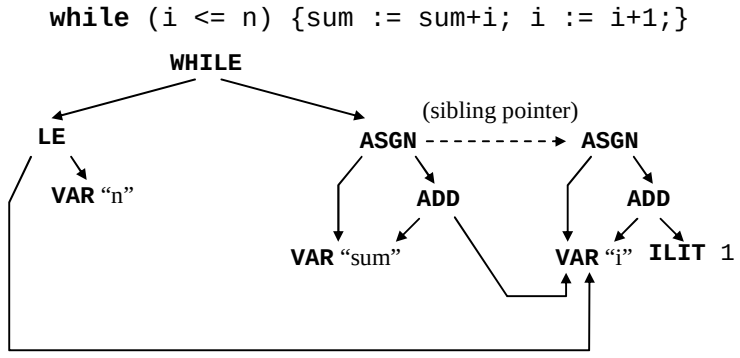


For field access, the **DOT** operator is used. The following is an example on the **DOT** operator, were *p* is a record variable of type *person*:

```
record date {
  day:  int;
  month:int;
  year: int;
}
record person {
  name: char[20];
  birth:date;
}
var p:person;
```

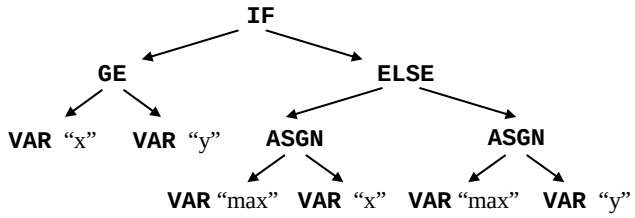


3 For statements, we need statement nodes. Statement nodes have a *sibling* pointer for sequencing. An assignment statement is translated into an **ASGN** node. The left pointer points at the object on the left hand side of the assignment operator, and the right pointer points at the expression on the 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 can be implemented with 3 children pointers and a sibling pointer. However, to make it consistent with the other nodes, we will limit the **IF** node to just 2 children pointers and a sibling pointer. We will use an **ELSE** node for the optional **else** part. The following example shows the syntax tree of an **if** statement with an **else** part. The same **VAR** symbol appears twice for clarity.

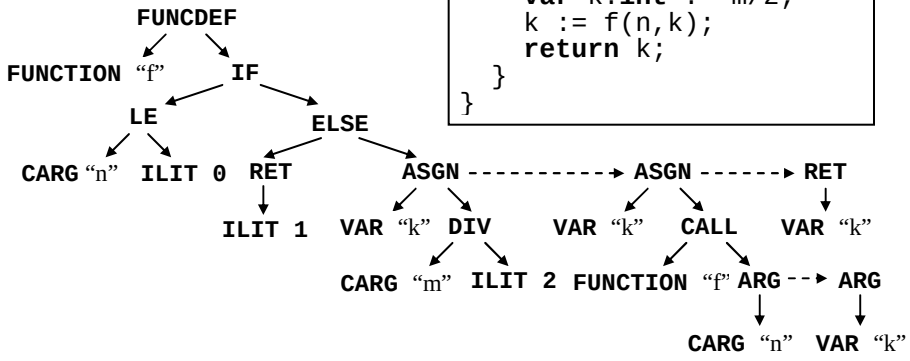
```
if (x >= y)
  max := x;
else
  max := y;
```



4 To translate function definitions, a **FUNCDEF** node is used. The left pointer points at a **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 **FUNCTION** symbol and a right pointer pointing at an actual **ARG** (argument) list. Each **ARG** node points at an actual argument. 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, where *f* appears on line 1 (L1), position 10 (P10), and scope number 1 (SCP1). Indentation helps visualizing the structure of the syntax tree. Use curly braces for sequencing, when sibling pointer is not NULL.

```
(FUNCDEF
  (FUNCTION f L1 P10 SCP1 INT)
  (IF
    (LE
      (CARG n L1 P14 SCP2 INT)
      (ILIT 0 S4)
    )
    (ELSE
      (RET
        (ILIT 1 S4)
      )
```

```
      { (ASGN
          (VAR k L4 P9 SCP3 INT)
          (DIV
            (CARG m L1 P12 SCP2 INT)
            (ILIT 2 S4)
          )
        )
        (ASGN
          (VAR k L4 P9 SCP3 INT)
          (CALL
            (FUNCTION f L1 P10 SCP1 INT)
            { (ARG
                (CARG n L1 P14 SCP2 INT)
              )
              (ARG
                (VAR k L4 P9 SCP3 INT)
              )
            }
          )
        )
        (RET
          (VAR k L4 P9 SCP3 INT)
        )
      }
    )
  )
)
```

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 *op*, its *value* and its *size* on a separate line.

3 Output the list of type symbols in the type table. For each type symbol, output its *op*, its *name* (if not NULL), its *size*, its *alignment*, and its element *type* between nested parentheses (when the type link is not NULL). For field symbols, you should output the field *name*, the record type *name* that encloses the field, the field *offset*, and the field *type*. Each type symbol should appear on a separate line.

4 Output the list of object symbols in the object table. For each object symbol, output its *op*, its *name*, its *line* number, its *position*, its *scope* number, and its *type*.

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