
NJIT
THE EDGE IN KNOWLEDGE

CS 280
Programming Language Concepts

Fall 2022

Programming Assignment 2

Building a Recursive-Descent Parser

Programming Assignment 2

- Objectives
  - building a recursive-descent parser for our simple language.

- Notes:
  - Read the assignment carefully to understand it.
  - Understand the functionality of each function, and the required error messages to be printed out.

- The syntax rules of a small programming language are given below using EBNF notations.

Programming Language Definition

1. Prog ::= PROGRAM IDENT StmtList END PROGRAM
2. StmtList ::= Stmt; { Stmt; }
3. Stmt ::= DeclStmt | ControlStmt
4. DeclStmt ::= (INT | FLOAT | BOOL) VarList
5. VarList ::= Var { ,Var }
6. ControlStmt ::= AssigStmt | IfStmt | PrintStmt
7. PrintStmt ::= PRINT (ExprList)
8. IfStmt ::= IF (Expr) THEN StmtList [ELSE StmtList] END IF
9. AssignStmt ::= Var = Expr
10. Var ::= IDENT
11. ExprList ::= Expr { , Expr }
12. Expr ::= LogORExpr ::= LogANDExpr { || LogANDRxpr }
13. LogANDExpr ::= EqualExpr { && EqualExpr }
14. EqualExpr ::= RelExpr [== RelExpr]
15. RelExpr ::= AddExpr [(< | >) AddExpr]
16. AddExpr :: MultExpr { (+ | -) MultExpr }
17. MultExpr ::= UnaryExpr { (* | /) UnaryExpr }
18. UnaryExpr ::= (- | + | !) PrimaryExpr | PrimaryExpr
19. PrimaryExpr ::= IDENT | ICONST | RCONST | SCONST | BCONST | (Expr)

Example Program of the Language

```
PROGRAM Cylinder
/*Clean Program with If-Then-Else*/
  INT r, a, h, b;
  r = 8;
  h = 10;
  a = 0;
  FLOAT surface, volume;
  IF ( r > 5 && h > 10) THEN
    volume = 3.14 * r * r * h;
    surface = 2 * 3.14 * r * h + 2 * 3.14 * r * r;
    PRINT  ("The output results are " , volume, " ", surface, " " , h,
    " ", r);
  ELSE
    PRINT  ("No computations should be done!" );
  END IF;

END PROGRAM
```

Description of the Language

- The language has three types: INT, FLOAT and BOOL.
- The precedence rules of operators in the language are as shown in the table of operators precedence levels.
- The PLUS, MINUS, MULT, DIV, AND, OR operators are left associative.
- A variable has to be declared in a declaration statement.
- An IfStmt evaluates a logical expression (Expr) as a condition. If the logical condition value is true, then the StmtList in the Then-part are executed, otherwise they are not. An else part for an IfSmt is optional. Therefore, If an Else-part is defined, the StmtList in the Else-part are executed when the logical condition value is false.
- A PrintStmt evaluates the list of expressions (ExprList), and prints their values in order from left to right followed by a newline.

Description of the Language

- The ASSOP operator (=) in the AssignStmt assigns a value to a variable. It evaluates the Expr on the right-hand side and saves its value in a memory location associated with the left-hand side variable (Var). A left-hand side variable of a numeric type can be assigned a value of either one of the numeric types (i.e., INT, FLOAT) of the language. For example, an integer variable can be assigned a real value, and a real variable can be assigned an integer value. In either case, conversion of the value to the type of the variable must be applied. A BOOL var in the left-hand side of an assignment statement must be assigned a Boolean value.
- The binary operations for addition, subtraction, multiplication, and division are performed upon two numeric operands (i.e., INT, FLOAT) of the same or different types. If the operands are of the same type, the type of the result is the same type as the operator's operands. Otherwise, the type of the result is REAL. The binary logic operations for the AND and OR are applied on two Boolean operands.

Description of the Language

- The LTHAN and GTHAN relational operators and the EQUAL operator operate upon two operands of compatible types. The evaluation of a relational expression, based on LTHAN or GTHAN operators, or an Equality expression, based on the Equal operator, produce either a true or false value.
- The unary sign operators (+ or -) are applied upon unary numeric operands (i.e., INT, FLOAT). While the unary NOT operator is applied upon a Boolean operand (i.e., BOOL).
- It is an error to use a variable in an expression before it has been assigned.

Table of Operators Precedence Levels

Precedence	Operator	Description	Associativity
1	Unary +, -, and !	Unary plus, minus, and logical NOT	Right-to-Left (ignored)
2	*, /	Multiplication and Division	Left-to-Right
3	+, -	Addition and Subtraction	Left-to-Right
4	<, >	Relational operators < and >	(no cascading)
5	==	Equality operator	(no cascading)
6	&&	Logical AND	Left-to-Right
7	||	Logical OR	Left-to-Right

Recursive-Descent Parser

- The parser includes one function per syntactic rule or nonterminal.
- Each function recognizes the right hand side of the rule.
  - If the function needs to read a token, it can read it using getNextToken().
  - If the function needs a nonterminal symbol, it calls the function for that nonterminal symbol.
- There is no explicit generation of a parse tree to be implemented. However the recursive-descent parser is tracing the parse tree implicitly for the input program.
  - Use printout statements to enable you to debug your implementation of the parser. Notice, this is not part of the assignment.

Given Files

- “lex.h”
- “lex.cpp” (you can use your implementation, or copy and use my lexical analyzer when I publish it).
- “parse.h”
- “prog2.cpp”: main function as a driver program for testing your parser implementation.
- “parse.cpp” file with definitions and the implementation of some functions of the parser.

parse.h

- All recursive-descent functions take a reference to an input stream, and a line number, and return a Boolean value.
  - The value returned is false if the call to the function has detected a syntax error. Otherwise, it is true.
  - `PrimaryExpr` function takes an extra parameter for the passed sign operator. Note, the function will make use of the sign in the evaluation of expressions when the interpreter will be built.

parse.h

- Functions' prototypes in "parse.h"

```
extern bool Prog(istream& in, int& line);
extern bool StmtList(istream& in, int& line);
extern bool DeclStmt(istream& in, int& line);
extern bool ControlStmt(istream& in, int& line);
extern bool Stmt(istream& in, int& line);
extern bool PrintStmt(istream& in, int& line);
extern bool IfStmt(istream& in, int& line);
extern bool AssignStmt(istream& in, int& line);
extern bool VarList(istream& in, int& line);
extern bool Var(istream& in, int& line);
extern bool ExprList(istream& in, int& line);
extern bool LogANDExpr(istream& in, int& line);
extern bool Expr(istream& in, int& line);//or LogORExpr
extern bool EqualExpr(istream& in, int& line);
extern bool RelExpr(istream& in, int& line);
extern bool AddExpr(istream& in, int& line);
extern bool MultExpr(istream& in, int& line);
extern bool UnaryExpr(istream& in, int& line);
extern bool PrimaryExpr(istream& in, int& line, int sign);
extern int ErrCount();
```

parse.cpp

- Token Lookahead
  - Remember that we need to have one token for looking ahead.
  - A mechanism is provided through functions that call the existing getNextToken, and include the pushback functionality. This is called a “wrapper”.
  - Wrapper for lookahead is given in “parse.cpp”.

```
namespace Parser {
	bool pushed_back = false;
	LexItem pushed_token;
	static LexItem GetNextToken(istream& in, int& line){
		if( pushed_back ) {
			pushed_back = false;
			return pushed_token;
		}
		return getNextToken(in, line);
	}
	static void PushBackToken(LexItem & t) {
		if( pushed_back ) {
			abort();
		}
		pushed_back = true;
		pushed_token = t;
	}
}
```

Wrapper for lookahead (given in “parse.cpp”)

- To get a token:
  - `Parser::GetNextToken(in, line)`
- To push back a token:
  - `Parser::PushBackToken(t)`

- NOTE after push back, the next time you call Parser::GetNextToken(), you will retrieve the pushed-back token.
- NOTE an exception is thrown if you push back more than once (using abort()).

parse.cpp

- A map container that keeps a record of the defined variables in the parsed program, defined as:

```
map<string, bool> defVar;
```

  - The key of the `defVar` is a variable name, and the value is a Boolean that is set to true when the first time the variable has been declared in a declaration statement, otherwise it is false.
  - The use of a variable that has not been declared is an error.
  - It is an error to redefine a variable.

parse.cpp

- Static `int` variable for counting errors, called `error_count` and a function to return its value, called `ErrCount()`.

```
static int error_count = 0;
int ErrCount(){
    return error_count;
}
```

- A function definition for handling the display of error messages, called `ParserError`.

```
void ParseError(int line, string msg){
   ++error_count;
   cout << line << ": " << msg << endl;
}
```

parse.cpp

- Implementations Examples of some functions:
  - ControlStmt
  - PrintStmt
  - ExprList

Implementation Examples: Stmt

- ControlStmt function

```
ControlStmt ::= AssigStmt | IfStmt | PrintStmtmt
```

  - The function is called from Stmt function.
  - The function calls `Parser::GetNextToken()` to determine which function to call based on the next token.
  - If the next token does not belong to the first token of any of the possible statements, it is pushed back and the status is returned false.

Implementation Examples: ControlStmt

```
bool ControlStmt(istream& in, int& line) {
bool status;
LexItem t = Parser::GetNextToken(in, line);
switch( t.GetToken() ) {
   case PRINT:
   status = PrintStmt(in, line); break;
   case IF:
   status = IfStmt(in, line); break;
   case IDENT:
        Parser::PushBackToken(t);
        status = AssignStmt(in, line); break;
   default:
        Parser::PushBackToken(t);
        return false;
   }
        return status;
}//End of ControlStmt
```

Implementation Examples: WriteLnStmt

- `PrintStmt` function
  - Grammar rule

    ```
    PRINTStmt := PRINT (ExprList)
    ```

  - The function checks for the left and right parentheses
  - The function calls ExprList()
  - Checks the returned value from ExprList. If it returns false an error message is printed, such as

    ```
    Missing expression after Print
    ```

    - Then returns a false value
  - Evaluation: the function prints out the list of expressions' values, and returns successfully. More to come about the interpreters actions in Programming Assignment 3.

Implementation Examples: PrintStmt

```
bool PrintStmt(istream& in, int& line) {
  LexItem t;
  t = Parser::GetNextToken(in, line);
  if( t != LPAREN ) {
        ParseError(line, "Missing Left Parenthesis");
        return false;
  }
  bool ex = ExprList(in, line);
  if( !ex ) {
        ParseError(line, "Missing expression after PRINT");
        return false;
  }
  t = Parser::GetNextToken(in, line);
  if(t != RPAREN ) {
        ParseError(line, "Missing Right Parenthesis");
        return false;
  }
  return ex;
}//End of PrintStmt
```

Implementation Examples: ExprList

- ExprList Function
  - Grammar rule:

```
ExprList ::= Expr {, Expr}
```

Implementation Examples: Expr

```
bool ExprList(istream& in, int& line) {
  bool status = false;
  status = Expr(in, line);
  if(!status){
        ParseError(line, "Missing Expression");
        return false;
  }
  LexItem tok = Parser::GetNextToken(in, line);

  if (tok == COMMA) {
        status = ExprList(in, line);
  }
  else if(tok.GetToken() == ERR){
        ParseError(line, "Unrecognized Input Pattern");
        cout << "(" << tok.GetLexeme() << ")" << endl;
        return false;
  }
  else{
        Parser::PushBackToken(tok);
        return true;
  }
  return status;
}//End of ExprList
```

Generation of Syntactic Error Messages

- The result of an unsuccessful parsing is a set of error messages printed by the parser functions.
- If the parser fails, the program should stop after the parser function returns.
- Lexical analyzer's error messages should be included as well. You can still use the same function `ParseError()` given and add the lexeme causing the problem.
- The assignment does not specify the exact error messages that should be printed out by the parser; however, **the format of the messages should be the line number, followed by a colon and a space, followed by some descriptive text** (as given by the `ParseError()` function in "parse.h" file.
- Suggested parser error messages are shown in the example test cases at the end.

Testing Program “prog2.cpp”

- You are given the testing program “prog2.cpp” that reads a file name from the command line. The file is opened for reading.
- A call to `Prog()` function is made. If the call fails, the program should stop and display a message as "Unsuccessful Parsing", and display the number of errors detected. For example:

```
Unsuccessful Parsing
Number of Syntax Errors: 3
```

- If the call to `Prog()` function succeeds, the program should stop and display the message "Successful Parsing", and the program stops.

Test Cases Files

- You are provided by a set of 19 test cases files associated with Programming Assignment 2. Vocareum automatic grading will be based on these testing files. You may use them to check and test your implementation. These are available in compressed archive "PA2 Test Cases.zip" on Canvas assignment. The testing case of each file is defined in the Grading table in the assignment handout.
- Automatic grading of clean source code test file will be based on checking against the output message:

```
Successful Parsing
```

Test Cases Files

- In each of the other testing files, there is one syntactic error at a specific line. The automatic grading process will be based on the statement number at which this error has been found and the number of associated error messages with this syntactic error.
- You can use whatever error message you like. There is no check against the contents of the error messages.
- A check of the number of errors your parser has produced and the number of errors printed out by the program are made.

Test Cases: Example 1

```
PROGRAM circle
        /*Incorrect Type*/
        FLOAT r, a, p, b;
        REAL i, j ;
        r = 8;
        p = 0;
        a = 0;
        j = -1;
        IF ( r > 5) THEN a = (3.14) * r * r; END IF;
        IF (a < 100) THEN p = 2 * 3.14 * b; END IF;
        /*Display the results*/
        PRINT ( "The output results are " , r, p, b);
END PROGRAM
```

Output:

```
4: Undeclared Variable
4: Missing Left-Hand Side Variable in Assignment statement
4: Incorrect control Statement.
4: Syntactic error in Program Body.
4: Incorrect Program Body.
Unsuccessful Parsing
Number of Syntax Errors 5
```

Test Cases: Example 2

```
PROGRAM circle
        /*Missing comma in declaration statement*/
        INT r, a p, b;
        FLOAT i, j;
        r = 8;
        p = 0;
        a = 0;
        j = -1;
        IF ( r > 5) THEN
          a = (3.14) * r * r;
          p = 2 * 3.14 * b;
        END IF;
        /*Display the results*/
        PRINT ( "The output results are " , r, p, b);
END  PROGRAM
```

Output

```
3: Missing semicolon at end of Statement.
3: Incorrect Program Body.
Unsuccessful Parsing
Number of Syntax Errors 2
```

Test Cases: Example 3

```
PROGRAM Cylinder
  /*Illegal Equality Expression*/
  INT r, a, h, b;
  r = 8; h = 10; a = 0;
  FLOAT surface, volume; BOOL flag;
  IF ( r > 5 && h > 10) THEN
    volume = 3.14 * r * r * h;
    surface = 2 * 3.14 * r * h + 2 * 3.14 * r * r;
  END IF;
  flag = (r == h == 10);
  PRINT  ("The output results are " , volume, " ", surface, " flag: "
  , flag);
END  PROGRAM
```

Output:

```
10: Illegal Equality Expression.
10: Missing expression after Left Parenthesis
10: Missing Expression in Assignment Statement
10: Incorrect control Statement.
10: Syntactic error in Program Body.
10: Incorrect Program Body.
Unsuccessful Parsing
Number of Syntax Errors 6
```

Test Cases: Example 4

```
PROGRAM Cylinder
  /*Clean Program with If-Then-Else*/
  INT r, a, h, b;
  r = 8;
  h = 10;
  a = 0;
  FLOAT surface, volume;
  IF ( r > 5 && h > 10) THEN
     volume = 3.14 * r * r * h;
     surface = 2 * 3.14 * r * h + 2 * 3.14 * r * r;
     PRINT  ("The output results are " , volume, " ", surface, " " ,
     h, " ", r);
   ELSE
     PRINT  ("No computations should be done!" );
  END IF;
END  PROGRAM
```

Output:

```
Successful Parsing
```


NJIT
THE EDGE IN KNOWLEDGE