
NJIT
THE EDGE IN KNOWLEDGE

CS 280
Programming Language Concepts

Spring 2023

Programming Assignment 2

Building a Recursive-Descent Parser for SPL Language

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 the SPL programming language are given below using EBNF notations.

SPL Language Definition

1. Prog ::= StmtList
2. StmtList ::= Stmt ;{ Stmt; }
3. Stmt ::= AssignStme | WriteLnStmt | IfStmt
4. WriteLnStmt ::= writeln (ExprList)
5. IfStmt ::= if (Expr) '{' StmtList '}' [else '{' StmtList '}']
6. AssignStmt ::= Var = Expr
7. Var ::= NIDENT | SIDENT
8. ExprList ::= Expr { , Expr }
9. Expr ::= RelExpr [(-eq|==) RelExpr]
10. RelExpr ::= AddExpr [(-lt | -gt | < | >) AddExpr]
11. AddExpr :: MultExpr { (+ | - | .) MultExpr }
12. MultExpr ::= ExponExpr { (* | / | **) ExponExpr }
13. ExponExpr ::= UnaryExpr { ^ UnaryExpr }
14. UnaryExpr ::= [(- | +)] PrimaryExpr
15. PrimaryExpr ::= IDENT | SIDENT | NIDENT | ICONST | RCONST | SCONST | (Expr)

Example Program of SPL Language

```
#Clean program with nested if statement
        $r = 50;
        @flag = 'true';
        if (@flag -lt 'true' ) {
                $y_1 = 5;
                if($y_1 > 4)
                {
                        @flag = 'hello';
                        $y_1 = $y_1 + 1;
                }
                else
                {
                        @flag = 'goodbye';
                        $y_1 = $y_1 -1;
                };
                writeln ('Value = ', $r ,'$y_1 = ', @flag);
        }
        else {
                $y_1 = (7.5);
                @flag = 'goodbye';
        };
        $r = $r + @flag;
```

Description of the SPL Language

- The language has two types: Numeric, and String.
- The SPL language does not have explicit declaration statements. However, variables are implicitly declared as Numeric type by a variable name starting with "$", or as String type by a variable name starting with "@".
- All SPL variables must first be initialized by an assignment statement before being used.
- The precedence rules of operators in the language are as shown in the table of operators' precedence levels.
- The PLUS, MINUS, MULT, DIV, CAT, and SREPEAT operators are left associative.

Table of Operators Precedence Levels

Precedence	Operator	Description	Associativity
1	Unary +, -	Unary plus, and minus,	Right-to-Left
2	^	Exponent	Right-to-Left
3	*, /, **	Multiplication, Division, and string repetition	Left-to-Right
4	+, -, . (Dot)	Addition, Subtraction, and String concatenation	Left-to-Right
5	<, > -gt, -lt	• Numeric Relational • String Relational	(no cascading)
6	== -eq	• Numeric Equality • String Equality	(no cascading)

Description of the SPL Language

- The binary operations of numeric operators as addition, subtraction, multiplication, and division are performed upon two numeric operands. While the binary string operator for concatenation is performed upon two string operands. If one of the operands does not match the type of the operator, that operand is automatically converted to the type of the operator.
- Similarly, numeric relational and equality operators (==, <, and >) operate upon two numeric type operands. While, string relational and equality operators (-eq, -lt, -gt) operate upon two string type operands. The evaluation of a relational or equality expression, produces either a true or false value. If one of the operands does not match the type of the operator, that operand is automatically converted to the type of the operator. For all relational and equality operators, no cascading is allowed.

Description of the SPL Language

- The exponent operator is applied on a numeric type operand, and the exponent value must be a numeric type value. No automatic conversion to numeric type operand is applied in case of an expression with exponent operator. Note that, exponent operators follow right-to-left association.
- The binary operation for string repetition (**) operates upon a string operand as the first operand, where the second operand must be a numeric expression of integer value.
- The unary sign operators (+ or -) are applied upon unary numeric type operands only.

Description of the SPL Language

- An IfStmt evaluates a logical expression (Expr) as a condition. If the logical condition value is true, then the StmtList in the If-clause are executed, otherwise they are not. An else clause for an IfSmt is optional. Therefore, If an Else-clause is defined, the StmtList in the Else-clause are executed when the logical condition value is false.
- A WriteLnStmt evaluates the list of expressions (ExprList), and prints their values in order from left to right followed by a newline.

Description of the SPL 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 must be assigned a numeric value. While a left-hand side variable of a String type must be assigned a string value. Type conversion must be automatically applied if the right-hand side value of the evaluated expression does not match the type of the left-hand side variable.
- It is an error to use a variable in an expression before it has been assigned.

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).
- “parser.h”
- “prog2.cpp”: main function as a driver program for testing your parser implementation.
- “parser.cpp” file with definitions and the implementation of some functions of the parser.

parser.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.

parser.h

- Functions' prototypes in "parser.h"

```
extern bool Prog(istream& in, int& line);
extern bool StmtList(istream& in, int& line);
extern bool Stmt(istream& in, int& line);
extern bool SimpleStmt(istream& in, int& line);
extern bool SimpleIfStmt(istream& in, int& line);
extern bool WritelnStmt(istream& in, int& line);
extern bool IfStmt(istream& in, int& line);
extern bool AssignStmt(istream& in, int& line);
extern bool Var(istream& in, int& line);
extern bool ExprList(istream& in, int& line);
extern bool Expr(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 ExponExpr(istream& in, int& line);
extern bool UnaryExpr(istream& in, int& line);
extern bool PrimaryExpr(istream& in, int& line, int sign);
```

parser.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 “parser.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 “parser.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()).

parser.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.

parser.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;
}
```

parser.cpp

- Implementations Examples of some functions:
  - WriteLnStmt
  - ExprList

Implementation Examples: WriteLnStmt

- `WriteLnStmt` function
  - Grammar rule

    ```
    WriteLnStmt := writeln (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: `WriteLnStmt`

```
bool WriteLnStmt(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: ExprList

```
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.
  - If the scanning of the input file is completed with no detected errors, the parser should display the message (DONE) on a new line before returning successfully to the caller program.
  - 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 "parser.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 case 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 "PA 2 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:

```
(DONE)
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.

Example 1: Undefined variable

```
# Testing variables assignment

$x1 = 0.5; #numeric variable
@_y1 = 'Welcome!' ** 3;        #string variable
@str = 'Welcome' . @_Y1;
 $z = 0.0 ;

@r_25 = 50.;
```

Output:

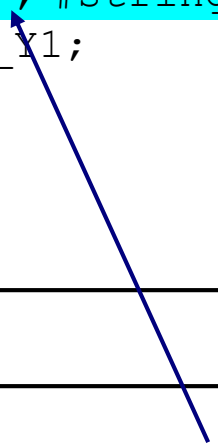
```
1. Line # 5: Using Undefined Variable
2. Line # 5: Missing operand after operator
3. Line # 5: Missing Expression in Assignment Statement
4. Line # 5: Incorrect Assignment Statement.
5. Line # 5: Syntactic error in Program Body.
6. Line # 5: Missing Program
Unsuccessful Parsing
Number of Syntax Errors 6
```

Example 2: Missing operand after operator

```
# Missing operand after operator

$x1 = 0.5; #numeric variable
@_y1 = 'Welcome!' ** ; #string variable
@str = 'Welcome' . @_Y1;
 $z = 0.0 ;

@r_25 = 50.;
```



Output

```
1. Line # 4: Missing operand after operator
2. Line # 4: Missing Expression in Assignment Statement
3. Line # 4: Incorrect Assignment Statement.
4. Line # 5: Syntactic error in Program Body.
5. Line # 5: Missing Program
Unsuccessful Parsing
Number of Syntax Errors 5
```

Example 3: Illegal Relational Expression

```
$r = 50;
        @flag = 'true';
    #Illegal relational rxpression
        if (@flag -lt $r < 'true' ) {
                $y_1 = 5;
                @flag = 'hello';
        }
        else {
                $y_1 = (7.5);
                @flag = 'goodbye';
        writeln ('Value = ', $r, '$y_1 = ', @flag);
        $r = $r + @flag;
```

Output:

```
1. Line # 4: Missing Right Parenthesis of If condition
2. Line # 4: Incorrect If-Statement.
3. Line # 4: Syntactic error in Program Body.
4. Line # 4: Missing Program
Unsuccessful Parsing
Number of Syntax Errors 4
```

Example 4: Clean Program

```
#Clean program with nested if statement
        $r = 50;
        @flag = 'true';
        if (@flag -lt 'true' ) {
                $y_1 = 5;
                if($y_1 > 4)
                {
                        @flag = 'hello';
                        $y_1 = $y_1 + 1;
                }
                else
                {
                        @flag = 'goodbye';
                        $y_1 = $y_1 -1;
                };
         writeln ('Value = ', $r ,'$y_1 = ', @flag);
        }
        else {
                $y_1 = (7.5);
                @flag = 'goodbye';
        };
        $r = $r + @flag;
```

Output:

```
(DONE)
Successful Parsing
```


NJIT
THE EDGE IN KNOWLEDGE