NJIT

THE EDGE IN KNOWLEDGE

CS 280
Programming Language Concepts

Spring 2024

Programming Assignment 2

Building a Recursive-Descent Parser

for a Simple Fortran95-Like Language

Programming Assignment 2

- Objectives
  - building a recursive-descent parser for our simple Fortran 95-like language (SFort95).

- 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 programming language are given below using EBNF notations.

SFort95 Language Definition

```
1.  Prog ::= PROGRAM IDENT {Decl} {Stmt} END PROGRAM IDENT
2.  Decl ::= Type :: VarList
3.  Type ::= INTEGER | REAL | CHARARACTER [(LEN = ICONST)]
4.  VarList ::= Var [= Expr] {, Var [= Expr]}
5.  Stmt ::= AssigStmt | BlockIfStmt | PrintStmt | SimpleIfStmt
6.  PrintStmt ::= PRINT *, ExprList
7.  BlockIfStmt ::= IF (RelExpr) THEN {Stmt} [ELSE {Stmt}] END IF
8.  SimpleIfStmt ::= IF (RelExpr) SimpleStmt
9.  SimpleStmt ::= AssigStmt | PrintStmt
10. AssignStmt ::= Var = Expr
11. ExprList ::= Expr {, Expr}
```

SFort95 Language Definition

```
12.RelExpr ::= Expr [ ( == | < | > ) Expr ]

13.Expr ::= MultExpr { ( + | - | // ) MultExpr }

14.MultExpr ::= TermExpr { ( * | / ) TermExpr }
15.TermExpr ::= SFactor { ** SFactor }

16.SFactor ::= [+ | -] Factor

17.Var ::= IDENT

18.Factor ::= IDENT | ICONST | RCONST | SCONST | (Expr)
```

Example Program of SFort95 Language

```
PROGRAM circle
      !Variable declarations
      REAL :: r, a, p, b
      CHARACTER :: str1, str2
      a = (3.14) * r * r
      IF ( r == 5) THEN
        p = 2 * 3.14 * b
        print *, a, p, r
      else
            str1 = str1  // str2
            print *, str1, str2
      END IF
END PROGRAM circle
```

Description of the Language

- The language has three types: integer, real, and character.
- Declaring a numeric variable does not automatically assign a value, say zero, to this variable, until a value has been assigned to it. Variables can be given initial values in the declaration statements using initialization expressions and literals.

```
INTEGER :: i = 5, j = 100
REAL :: x, y = 1.0E5
```

- CHARACTER variables can refer to one character, or to a string of characters which is achieved by adding a length specifier to the object declaration. In this language, all character variables are assumed to be initialized with blank character(s) based on their lengths, if they are not initialized. The following declarations of the variables `grade` of one character, and `name` of 10 characters. Both are initialized with blank string.

Description of the Language

```
CHARACTER :: grade
CHARACTER(LEN=10):: name
```

- The following two declarations of the character variable are initialized, where light variable is initialized by the first character of the string ‘Amber’, and the boat variable is initialized by the string 'Wellie' padded, to the right, with 3 blanks.

```
CHARACTER :: light = 'Amber'
CHARACTER(LEN=9) :: boat = 'Wellie'
```

- 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 +, -	Unary plus, minus	Right-to-Left
2	**	Exponentiation	Right-to-Left
3	*, /	Multiplication, Division	Left-to-Right
4	+, -, //	Addition, Subtraction, Concatenation	Left-to-Right
5	<, >, ==	Less than, greater than, and equality	(no cascading)

Description of the Language

- The precedence rules of operators in the language are as shown in the table of operators’ precedence levels.
- The PLUS (+), MINUS, MULT (*), DIV (/), and CATenation (//) operators are left associative.
- The unary operators (+, -) and exponentiation (**) are right-to-left associative.
- A SimpleIfStmt evaluates a relational expression (RelExpr) as a condition. The If-clause for a SimpleIfStmt is just one Simple statement (SimpleStmt). If the condition value is true, then the If-clause statement is executed, otherwise it is not. No Else-clause is associated with a SimpleIfStmt.

Description of the Language

- A BlockIfStmt evaluates a relational expression (RelExpr) as a condition. If the condition value is true, then the If-clause statements are executed, otherwise they are not. An else clause for a BlockIfSmt is optional. Therefore, if an Else-clause is defined, the Else-clause statements are executed when the condition value is false.
- A PrintStmt statement evaluates the list of expressions (ExprList), and prints their values in order from left to right followed by a newline.
- The binary operations of numeric operators as addition, subtraction, multiplication, and division are performed upon two numeric operands.
  - 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 concatenation operator is performed upon two operands of character types (i.e. strings).
- The exponentiation operation is performed on REAL type operands only.

Description of the Language

- Similarly, relational operators (==, <, and >) operate upon two compatible type operands. The evaluation of a relational expression produces either a logical true or false value. For all relational operators, no cascading is allowed.
  - Numeric types (i.e., INTEGER and REAL) are compatible, and evaluation is based on converting an integer type operand to real if the two operands are of different numeric types. However, numeric types are not compatible with CHARACTER type.

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 must be assigned a numeric value. Type conversion must be automatically applied if the right-hand side numeric value of the evaluated expression does not match the numeric type of the left-hand side variable. .
    - While a left-hand side variable of character type must be assigned a value of a string that has the same length as the left-hand variable length in the declaration statement. If the length of the RHS string is greater than the LHS variable the string would be truncated, otherwise the string would be padded by blank characters.

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() (a wrapper function to handle token look ahead in the context of token pushback).
  - 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”
- “parser.h”
- “prog2.cpp”: main function as a driver program for testing your parser implementation.
- “GivenParserPart.cpp” file with definitions and the implementation of some functions of the parser that will be used to implement the parser in the file “parser.cpp”.

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.
  - `Factor` 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 Decl(istream& in, int& line);
extern bool Type(istream& in, int& line);
extern bool VarList(istream& in, int& line);
extern bool Stmt(istream& in, int& line);
extern bool SimpleStmt(istream& in, int& line);
extern bool PrintStmt(istream& in, int& line);
extern bool BlockIfStmt(istream& in, int& line);
extern bool SimpleIfStmt(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 RelExpr(istream& in, int& line);
extern bool Expr(istream& in, int& line);
extern bool MultExpr(istream& in, int& line);
extern bool TermExpr(istream& in, int& line);
extern bool SFactor(istream& in, int& line);
extern bool Factor(istream& in, int& line, int sign);
extern int ErrCount();
```

GivenParserPart.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 “GivenParserPart.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 “GivenParserPart.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()).

GivenParserPart.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.

- Note, no information is stored for the type of each variable. The collection of such information will be considered in PA 3 for building an interpreter.

GivenParserPart.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;
}
```

GivenParserPart.cpp

- Implementations Examples of some functions:
  - PrintStmt
  - ExprList

Implementation Examples: PrintStmt

- `PrintStmt` function
  - Grammar rule

    ```
    PrintStmt:= PRINT *, ExpreList
    ```

  - The function checks for the DEF token and comma.
  - 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 != DEF ) {
    ParseError(line, "Print statement syntax error.");
    return false;}
    t = Parser::GetNextToken(in, line);
    if( t != COMMA ) {
        ParseError(line, "Missing Comma.");
        return false;
    }
  bool ex = ExprList(in, line);
  if( !ex ) {
        ParseError(line, "Missing expression after writeln");
        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 examples of 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:

```
Successful Parsing
```

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.
- Grading of test cases with syntactic error is based on:
  - A check of some message has been printed,
  - the number of errors your parser has produced and
  - the number of errors printed out by the program are made.

Test Cases Files

- Grading of a clean test case will be based on checking against specific messages displayed based on some syntactic condition, and the two messages generated by the parser and the driver program, respectively.
    - Check the test cases of clean programs for the displayed message(s) and the provided table below. The specific messages are generated based on the language syntax being parsed. For example the generated output messages for test1 are:

```
Print statement in a Simple If statement.
(DONE)
Successful Parsing
```

Function	When	Message format	Test Case Example
Decl()	Character string is defined with a Length value constant.	"Definition of Strings with length of 20 in declaration statement."	test3
VarList()	A variable initialization is recognized.	"Initialization of the variable a in the declaration statement."	test2
SimpleIfStmt()	A Print or Assignment statements are parsed.	"Print statement in a Simple If statement."	test1
BlockIfStmt()	A closing END IF is parsed.	"End of Block If statement at nesting level 1"	test4 and test5

Example 1 (test7): Variable Redeclaration

```
PROGRAM circle
        !Variable redefinition
        REAL :: r, a, p, b
        CHARacter :: p, str2
END PROGRAM circle
```

Output:

```
4: Variable Redefinition
4: Missing Variables List.
4: Declaration Syntactic Error.
4: Incorrect Declaration in Program
Unsuccessful Parsing
Number of Syntax Errors 4
```

Example 2 (test110): Print statement syntax error

```
PROGRAM circle
       !Print syntax error
       INTEGER :: r, a, b , p
       CHARACTER :: str1, str2
       !Display the results
       PRINT * "The output results are " , a+4, p*2, -b
END PROGRAM circle
```

Output

```
6: Print statement syntax error.
6: Incorrect Statement in program
Unsuccessful Parsing
Number of Syntax Errors 2
```

Example 3 (test13): Block If statement missing THEN

```
PROGRAM circle
!Block If statement syntax error
       REAL :: r, a, p, b

       a = (3.14) * r * r
       IF ( r == 5)
         p = 2 * 3.14 * b
       else
              p = 0.0
       END IF

END PROGRAM circle
```

Missing THEN.

The parser thought it has recognized a Simple If statement.

The error message assumes what is missing is the End of Program.

Output:

```
Assignment statement in a Simple If statement.
8: Missing END of Program
Unsuccessful Parsing
Number of Syntax Errors 1
```

Example 4 (test5): Clean Program

```
PROGRAM circle
       !Variable declarations
       REAL :: r, a, p, b
       character (LEN = 15) :: str = "Hello World!", str2
       a = (3.14) * r * r
       IF ( r == 5) THEN
         IF ( r == 5) THEN
         p = 2 * 3.14 * b
         print *, a, p, r
       else
              str = str  // str2
              print *, str, str2
       END IF
       END IF
END PROGRAM circle
```

Output:

```
Initialization of the variable str in the declaration statement.
Definition of Strings with length of 15 in declaration statement.
End of Block If statement at nesting level 2
End of Block If statement at nesting level 1
(DONE)
Successful Parsing
```

NJIT

THE EDGE IN KNOWLEDGE