CS 280
Programming Language Concepts

Spring 2025

Programming Assignment 2

Building a Recursive-Descent Parser

for Simple Ada-Like Language

Programming Assignment 2

- Objectives
  - building a recursive-descent parser for our Simple Ada-like language (SADAL).
- 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.

SADAL Language Definition

1. Prog ::= PROCEDURE ProcName IS ProcBody
2. ProcBody ::= DeclPart BEGIN StmtList END ProcName ;
3. ProcName ::= IDENT
4. DeclPart ::= DeclStmt { DeclStmt }
5. DeclStmt ::= IDENT {, IDENT } : [CONSTANT] Type [(Range)] [:= Expr] ;
6. Type ::= INTEGER | FLOAT | BOOLEAN | STRING | CHARACTER
7. StmtList ::= Stmt { Stmt }
8. Stmt ::= AssignStmt | PrintStmts | GetStmt | IfStmt
9. PrintStmts ::= (PutLine | Put) (Expr) ;
10. GetStmt := Get (Var) ;
11. IfStmt ::= IF Expr THEN StmtList { ELSIF Expr THEN StmtList } [ELSE StmtList] END IF ;

SADAL Language Definition

12. AssignStmt ::= Var := Expr ;
13. Var ::= IDENT
14. Expr ::= Relation {(AND | OR) Relation }
15. Relation ::= SimpleExpr [(= | /= | < | <= | > | >=) SimpleExpr]
16. SimpleExpr ::= STerm { (+ | - | &) STerm }
17. STerm ::= [(+ | -)] Term
18. Term ::= Factor { (* | / | MOD) Factor }
19. Factor ::= Primary [** Primary] | NOT Primary
20. Primary ::= Name | ICONST | FCONST | SCONST | BCONST | CCONST | (Expr)
21. Name ::= IDENT [(Range)]
22. Range ::= SimpleExpr [. . SimpleExpr]

Example Program of SADAL Language

```
procedure prog17 is
        -- { Testing syntax of range definition in a substring
operation }
        x, y_1, w123: integer := 2;
        str1 : string := "Welcome";
        str2 : string (7);
        z : float := 0.0;
        flag , bing : boolean:= false ;
        ch : character;
Begin
        put("Value of x= "); put(x);
        str2 := str1(z+2..w123*3);
        if (z >= y_1 ) then
                z := z + Y_1;
                x := x -1;
        end if;
        put("Value of z= ");
        putline(z);
END prog1;
```

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.
  - Function that take an extra parameter for the passed sign operator are *Term*, *Factor*, and *Primary*. Note, these functions 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 ProcBody(istream& in, int& line);
extern bool DeclPart(istream& in, int& line);
extern bool DeclStmt(istream& in, int& line);
extern bool Type(istream& in, int& line);
extern bool StmtList(istream& in, int& line);
extern bool Stmt(istream& in, int& line);
extern bool PrintStmts(istream& in, int& line);
extern bool GetStmt(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 Expr(istream& in, int& line);
extern bool Relation(istream& in, int& line);
extern bool SimpleExpr(istream& in, int& line);
extern bool STerm(istream& in, int& line);
extern bool Term(istream& in, int& line, int sign);
extern bool Factor(istream& in, int& line, int sign);
extern bool Primary(istream& in, int& line, int sign);
extern bool Name(istream& in, int& line);
extern bool Range(istream& in, int& line);
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”.

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()).

```
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;
        }
}
```

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;
}
```

Implementation Examples: `StmtList`

```
//StmtList ::= Stmt { Stmt }
bool StmtList(istream& in, int& line){
        bool status;
        LexItem tok;
        status = Stmt(in, line);
        tok = Parser::GetNextToken(in, line);
        while(status && (tok != END && tok != ELSIF && tok != ELSE)){
                Parser::PushBackToken(tok);
                status = Stmt(in, line);
                tok = Parser::GetNextToken(in, line);
        }
        if(!status){
                ParseError(line, "Syntactic error in statement list.");
                return false;
        }
        Parser::PushBackToken(tok); //push back the END token
        return true;
}//End of StmtList
```

Implementation Examples: `DeclPart`

```
//DeclPart ::= DeclStmt { DeclStmt }
bool DeclPart(istream& in, int& line) {
  bool status = false;
  LexItem tok;
  status = DeclStmt(in, line);
  if(status){
    tok = Parser::GetNextToken(in, line);
    if(tok == BEGIN ){
      Parser::PushBackToken(tok);
      return true;
    }
    else {
      Parser::PushBackToken(tok);
      status = DeclPart(in, line);
    }
  }
  else{
    ParseError(line, "Non-recognizable Declaration Part.");
    return false;
  }
  return true;
}//end of DeclPart function
```

Generation of Syntactic Error Messages

- If the parser fails, the program should stop after the *Prog()* parser function returns.
- In case of successful parsing, the parser should display the list of all the declared variables in the scanned input file in order. Then, it should display the message "(DONE)" on a new line before returning successfully to the driver program. Check the format of the displayed list of variables associated with the clean programs in testprog17-testprog19.
- The results of an unsuccessful parsing are a set of error messages printed by the parser functions, as well as the error messages that might be detected by the lexical analyzer. The parser should detect the first discovered syntactic error and return back unsuccessfully.

Generation of Syntactic Error Messages

- The format of the error messages should be the line number, followed by a colon and a space, followed by some descriptive text. See the list of required error messages for certain test cases.
- Typically, the first displayed error message is directly associated with the designated test case defined in the Grading Table below. All other following error messages are generated as a consequence of the detected syntactic error, which its message is displayed first.
  - The assignment specifies a list of exact error messages that should be displayed by the parser for certain detected syntactic errors, which they are associated with designated test cases, as shown in the given table below for the error messages associated with specified test cases.

Driver 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.
  - The automatic grading of a clean source code file will be based on checking against the displayed list of declared variables, and the two messages generated by the parser and the driver program, respectively. Check the test cases of clean programs for the format of displayed outputs. For example, testprog17 output displays the following list of declared variables:

```
Declared Variables:
bing, ch, flag, prog17, str1, str2, w123, x, y_1, z

(DONE)
Successful Parsing
```

Test Cases Files

- The automatic grading process for test cases with unsuccessful parsing will be based on:
  - The statement number at which the first syntactic error has been detected,
  - The first error message being displayed (for specified test cases in the given table), and
  - The number of associated error messages being generated with this syntactic error. A check of the number of errors your parser has produced and the number of errors printed out by the driver program is checked.
  - Note, for all the other test cases that are not listed in the table, you can use whatever error message you like. There is no check against the contents of the first error message being displayed for those test cases, but there is a check that a certain textual message being displayed.

Table of Error Messages Associated with Specified Test Cases

Test Case	Error Message
testprog1: Incorrect type name	"Incorrect Declaration Type."
testprog2: Variable Redefinition	"Variable Redefinition"
testprog3: Missing Procedure Name	"Missing Procedure Name."
testprog4: Missing comma in Declaration Statement	"Missing comma in declaration statement."
testprog5: Missing Semicolon	"Missing semicolon at end of statement"
testprog6: Missing PROCEDURE Keyword	"Incorrect compilation file."
Testprog7: Missing closing parenthesis	"Missing Right Parenthesis"
Testprog8: Missing opening parenthesis	"Missing Left Parenthesis"
Testprog9: Undeclared variable	"Using Undefined Variable"
testprog10: Missing assignment operator	"Missing Assignment Operator"

Example 1 (testprog1):
Incorrect Declaration Type

```
procedure prog1 is
        -- { Testing incorrect type name }

        x, y_1, w234: integer;
        str : string := "Welcome";
        z : float;
        flag , bing : bool:= false ;

begin
        get(x);
        put("Value of x= "); put(x);
        put("Value of z= ");
        putline(z);
END prog1;
```

```
7: Incorrect Declaration Type.
7: Non-recognizable Declaration Part.
7: Incorrect compilation file.
Unsuccessful Parsing
Number of Syntax Errors 3
```

Example 2 (testprog2): Variable Redeclaration

```
procedure prog2 is
        -- { Testing Variable Redefinition }

        x, y_1, w234: integer;
        str : string := "Welcome";
        z, x : float;
        flag, bing : bool:= false ;

begin
        get(x);
        put("Value of x= "); put(x);
        put("Value of z= ");
        putln(z);
END prog1;
```

Output:

```
6: Variable Redefinition
6: Incorrect identifiers list in Declaration Statement.
6: Non-recognizable Declaration Part.
6: Incorrect compilation file.
Unsuccessful Parsing
Number of Syntax Errors 4
```

Example 3 (test1prog7): Missing semicolon at end of statement

```
procedure prog5 is
        -- { Missing semicolon }

        x, y_1, w234: integer;
        str : string := "Welcome";
        z : float;
        flag, bing : boolean:= false ;

begin
        get(x);
        put("Value of x= "); put(x)
        put("Value of z= ");
        putln(z);
END prog1;
```

```
11: Missing semicolon at end of statement
11: Invalid put statement.
11: Syntactic error in statement list.
11: Incorrect Proedure Body.
11: Incorrect Procedure Definition.
Unsuccessful Parsing
Number of Syntax Errors 5
```

Example 4 (testprog17): Clean Program

```
procedure prog17 is
        -- { Testing syntax of range definition in a substring operation }
        x, y_1, w123: integer := 2;
        str1 : string := "Welcome";
        str2 : string (7);
        z : float := 0.0;
        flag , bing : boolean:= false ;
        ch : character;
Begin
        put("Value of x= "); put(x);
        str2 := str1(z+2..w123*3);
        if (z >= y_1 ) then
                z := z + Y_1;
                x := x -1;
        end if;
        put("Value of z= ");
        putline(z);
END prog1;
```

```
Declared Variables:
bing, ch, flag, prog17, str1, str2, w123, x, y_1, z

(DONE)
Successful Parsing
```