

University of Central Florida

Department of Computer Science

COP 3402: System Software

Homework #3 (Parser)

REQUIRMENT:

Use the github classrooms link found in the webcourses announcement. Go over the README.md carefully and look at the provided test cases and output for examples.

All assignments must compile and run on the Eustis server. Please see course website for details concerning use of Eustis.

Objective:

In this assignment, you must implement a Recursive Descent Parser the PL/0 grammar. In addition, you must create a compiler driver to combine all of the compiler parts into one single program.

Example of a program written in PL/0:

```
var x, w;  
begin  
  x:= 4;  
  in w;  
  if w > x then  
    w:= w + 1  
  else  
    w:= x;  
  out w;  
end.
```

Component Descriptions:

The **Parser** is a program that reads in the output of the Scanner (HW2) and parses the lexemes (tokens). It must be capable of reading in the tokens produced by your Scanner (HW2) and produce, as output, a message that states whether the PL/0 program is well-formed (syntactically correct) if it follows the grammar rules in Appendix B. Otherwise, if the program does not follow the grammar, a message indicating the type of error present must be printed. A list of the errors to be considered can be found in Appendix C.

Appendix A:

Based on Wirth's definition for EBNF we have the following rules:

[] means **an optional item**, ^[L]_[SEP]

{ } means **repeat 0 or more times**.

Terminal symbols are enclosed in quote marks.

A period is used to indicate the end of the definition of a syntactic class.

EBNF of PL/0:

```
program ::= block "." .
block ::= const-declaration var-declaration proc-declaration statement.
constdeclaration ::= [ "const" ident "=" number { "," ident "=" number } ";" ].
var-declaration ::= [ "var" ident { "," ident } ";" ].
proc-declaration ::= { "procedure" ident ";" block ";" } statement .
statement ::= [ ident ":" expression
               | "call" ident
               | "begin" statement { ";" statement } "end"
               | "if" condition "then" statement [ "else" statement ]
               | "while" condition "do" statement
               | "read" ident
               | "write" ident
               | e ] .

condition ::= "odd" expression
            | expression rel-op expression.

rel-op ::= "=" | "<>" | "<" | "<=" | ">" | ">=" .
expression ::= [ "+" | "-" ] term { ( "+" | "-" ) term } .
term ::= factor { ( "*" | "/" ) factor } .
factor ::= ident | number | "(" expression ")" .
number ::= digit { digit } .
ident ::= letter { letter | digit } .
digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
letter ::= "a" | "b" | ... | "y" | "z" | "A" | "B" | ... | "Y" | "Z" .
```

PL/0 Lexical Conventions:

A numerical value is assigned to each token (internal representation) as follows:

nulsym = 1, identsym = 2, numbersym = 3, plussym = 4, minussym = 5, multsym = 6, slashsym = 7, oddsym = 8, eqsym = 9, neqsym = 10, lessym = 11, leqsym = 12, gtrsym = 13, geqsym = 14, lparsym = 15, rparsym = 16, commasym = 17, semicolonsym = 18, periodsym = 19, becomessym = 20, beginsym = 21, endsym = 22, ifsym = 23, then sym = 24, whilesym = 25, dosym = 26, callsym = 27, constsym = 28, varsym = 29, procsym = 30, writesym = 31, readsym = 32, elsesym = 33.

Reserved words: const, var, procedure, call, begin, end, if, then, else, while, do, read, write.

Special symbols: '+', '-', '*', '/', '(', ')', '=', ',', '.', '<', '>', ';', ':'.

Identifiers: $\text{identsym} = \text{letter} (\text{letter} \mid \text{digit})^*$

Numbers: $\text{numbersym} = (\text{digit})^+$

Invisible characters: tab, white spaces, newline

Comments denoted by: /* . . . */

Appendix B:

Error messages for the tiny PL/0 Parser:

1. Use = instead of :=.
2. = must be followed by a number.
3. Identifier must be followed by =.
4. **const**, **var**, **procedure** must be followed by identifier.
5. Semicolon or comma missing.
6. Incorrect symbol after procedure declaration.
7. Statement expected.
8. Incorrect symbol after statement part in block.
9. Period expected.
10. Semicolon between statements missing.
11. Undeclared identifier.
12. Assignment to constant or procedure is not allowed.
13. Assignment operator expected.
14. **call** must be followed by an identifier.
15. Call of a constant or variable is meaningless.
16. **then** expected.
17. Semicolon or } expected.
18. **do** expected.
19. Incorrect symbol following statement.
20. Relational operator expected.
21. Expression must not contain a procedure identifier.
22. Right parenthesis missing.
23. The preceding factor cannot begin with this symbol.
24. An expression cannot begin with this symbol.
25. This number is too large.
26. Identifier too long.
27. Invalid symbol.

Note:

1. Identifiers: Maximum 11 characters.
2. Numbers: Maximum 5 digits.
3. Invalid symbols are not accepted (or example % does not belong to PL/0 grammar).
4. Comments and invisible characters must be ignored and not tokenized.

Appendix C:

Recursive Descent Parser for a PL/0 like programming language in pseudo code:

As follows you will find the pseudo code for a PL/0 like parser. This pseudo code will help you out to develop your parser and intermediate code generator for tiny PL/0:

```
procedure PROGRAM;
begin
  GET(TOKEN);
  BLOCK;
  if TOKEN != "periodsym" then ERROR
end;

procedure BLOCK;
begin
  if TOKEN = "constsym" then begin
    repeat
      GET(TOKEN);
      if TOKEN != "identsym" then ERROR;
      GET(TOKEN);
      if TOKEN != "eqsym" then ERROR;
      GET(TOKEN);
      if TOKEN != NUMBER then ERROR;
      GET(TOKEN)
    until TOKEN != "commasym";
    if TOKEN != "semicolonsym" then ERROR;
    GET(TOKEN)
  end;
  if TOKEN = "intsym" then begin
    repeat
      GET(TOKEN);
      if TOKEN != "identsym" then ERROR;
      GET(TOKEN)
    until TOKEN != "commasym";
    if TOKEN != "semicolonsym" then ERROR;
    GET(TOKEN)
  end;
  while TOKEN = "procsym" do begin
    GET(TOKEN);
    if TOKEN != "identsym" then ERROR;
    GET(TOKEN);
    if TOKEN != "semicolonsym" then ERROR;
    GET(TOKEN);
```

```

        BLOCK;
        if TOKEN != "semicolon" then ERROR;
        GET(TOKEN)
    end;
    STATEMENT
end;

```

```

procedure STATEMENT;
begin
    if TOKEN = "ident" then begin
        GET(TOKEN);
        if TOKEN != "become" then ERROR;
        GET(TOKEN);
        EXPRESSION
    end
    else if TOKEN = "call" then begin
        GET(TOKEN);
        if TOKEN != "ident" then ERROR;
        GET(TOKEN)
    end
    else if TOKEN = "begin" then begin
        GET TOKEN;
        STATEMENT;
        while TOKEN = "semicolon" do begin
            GET(TOKEN);
            STATEMENT
        end;
        if TOKEN != "end" then ERROR;

        GET(TOKEN)
    end
    else if TOKEN = "if" then begin
        GET(TOKEN);
        CONDITION;
        if TOKEN != "then" then ERROR;
        GET(TOKEN);
        STATEMENT
    end
    else if TOKEN = "while" then begin
        GET(TOKEN);
        CONDITION;
        if TOKEN != "do" then ERROR;
        GET(TOKEN);
        STATEMENT
    end
end;

```

```

procedure CONDITION;

```

```

begin
  if TOKEN = "oddsym" then begin
    GET(TOKEN);
    EXPRESSION
  else begin
    EXPRESSION;
    if TOKEN != RELATION then ERROR;
    GET(TOKEN);
    EXPRESSION
  end
end;

procedure EXPRESSION;
begin
  if TOKEN = "plussym" or "minussym" then GET(TOKEN);
  TERM;
  while TOKEN = "plussym" or "minussym" do begin
    GET(TOKEN);
    TERM
  end
end;

procedure TERM;
begin
  FACTOR;
  while TOKEN = "multsym" or "slashsym" do begin
    GET(TOKEN);
    FACTOR
  end
end;

procedure FACTOR;
begin
  if TOKEN = "identsym" then
    GET(TOKEN)
  else if TOKEN = NUMBER then
    GET(TOKEN)
  else if TOKEN = "(" then begin
    GET(TOKEN);
    EXPRESSION;
    if TOKEN != ")" then ERROR;
    GET(TOKEN)
  end
  else ERROR
end;

```