

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.

Appendix A:

EBNF of PL/0:

program ::= block "." .
block ::= const-declaration var-declaration procedure-declaration statement.
const-declaration ::= ["const" ident "=" number { "," ident "=" number } ";"].
var-declaration ::= ["var" ident { "," ident } ";"].
procedure-declaration ::= { "procedure" ident ";" block ";" }
statement ::= [ident ":" expression
 | "call" ident
 | "begin" statement { ";" statement } "end"
 | "if" condition "then" statement ["else" statement]
 | "while" condition "do" statement
 | "read" ident
 | "write" ident] .
condition ::= "odd" expression
 | expression rel-op expression.
rel-op ::= "=" | "<" | "<=" | ">" | ">=" .
expression ::= ["+" | "-"] term { ("+" | "-") term } .
term ::= factor { ("*" | "/") factor } .
factor ::= ident | number | "(" expression ")" .

Terminals:

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" .

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

[] means an optional item.

{ } 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.

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

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

Identifiers: identsym = letter (letter | digit)*

Numbers: numbersym = (digit)+

Invisible characters: tab, white spaces, newline

Comments denoted by: /* ... */

Appendix B:

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
  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;
  end;
```

```

        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;

```