

University of Central Florida
Department of Computer Science
COP 3402: System Software
Spring 2018

Homework #3 (Parser- Code Generator)

Due Sunday, April 22, 2018

This is a team project (max. two people per team)

REQUIRMENT:

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 and an Intermediate Code Generator for tiny PL/0. In addition, you must create a compiler driver to combine all of the compiler parts into one single program.

Your compiler driver must support the following compiler directives:

- l : print the list of lexemes/tokens (scanner output) to the screen
- a : print the generated assembly code (parser/codegen output) to the screen
- v : print virtual machine execution trace (virtual machine output) to the screen

Example commands:

- | | |
|---------------------------------|--|
| <code>./compile -l -a -v</code> | Print all types of output to the console |
| <code>./compile -v</code> | Print only the VM execution trace to the console |
| <code>./compile</code> | Print nothing to the console except for “in” and “out” |

Note: You may want to also print all forms of output to a single output file in order to avoid losing points on the other requirements if your implementation of compiler directives does not work.

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 **compiler driver** is a program that manages the parts of the compiler. It must handle the input, output, and execution of the Scanner (HW2), the Parser (HW3), the Intermediate Code Generator (HW3) and the Virtual Machine (HW1).

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. In addition, the Parser must fill out the Symbol Table, which contains all of the variables, procedure and constants names within the PL/0 program. See Appendix E for more information regarding the Symbol Table. If the program is syntactically correct and the Symbol Table is created without error, the execution of the compiler driver continues with intermediate code generation.

The **Intermediate Code Generator** is a program that takes, as input, the output from the Parser, i.e. the Symbol Table and parsed code. As output, it produces the assembly language for your Virtual Machine (HW1). This functionality may be interleaved with the Parser functionality (i.e. generate assembly code as you parse the token list). Once the code has been generated for your Virtual Machine, the execution of the compiler driver continues by executing the generated assembly code on your Virtual Machine

Submission Instructions:

1.- Submit via WebCourses:

1. Source code of the tiny- PL/0 compiler.
2. A text file with instructions on how to use your program entitled readme.txt. (THIS IS VERY IMPORTANT. IF YOUR BUILD INSTRUCTIONS ARE NOT CLEAR ENOUGH TO BUILD THE CODE WE WILL NOT BE ABLE TO GRADE THE ASSIGNMENT.)
3. A text file composed of the input file to your Scanner and the output of your Parser to demonstrate a correctly formed tiny- PL/0 program. The Parser output should indicate the program is syntactically correct. Following the statement that the program is syntactically correct, the text file should contain the generated code from your intermediate code generator and the stack output from your Virtual Machine running your code.
4. All files should be compressed into a single .zip format.
5. Late assignments will not be accepted.

Appendix A:

Using the hw3test.sh

The test script, supplied in hw3test.sh, shown below, makes the following assumptions:

1. The source code is uploaded to the directory containing the input program files and the test shell.
2. The test shell, hw3test.sh, has execute permissions. To confirm this, enter the following command: `chmod +x hw3test.sh`.

In the example output below, each of the four input program files, named p1a thru p4a, are processed by the code in ./a.out. This code is the executable built from your lexer (scanner) built in Homework #2, parser/code generator (Homework #3) which output will be used as input into the virtual machine built in Homework #1.

```
NID@net1547:~/cop3402/hw3$ ./hw3test.sh ./a.out
Case #1a >p1a.txt -l -a Complete
Case #1b >p1a.txt -l -a -v Complete
Case #2a >p2a.txt -l -a Complete
Case #2b >p2a.txt -l -a -v Complete
Case #3a >p3a.txt -l -a Complete
Case #3b >p3a.txt -l -a -v Complete
Case #4a >p4a.txt -l -a Complete
Case #4b >p4a.txt -l -a -v Complete
NID@net1547:~/cop3402/hw3$
```

Please note that the four input PL-0 source files are organized as follows:

- **p1a.txt** is algebraic symbols, formula, and calculation testing.
- **p2a.txt** is the if/then testing.
- **p3a.txt** is loop testing.
- **p4a.txt** is variable naming, formula, loop, and calculation testing.

In the event that there is a discrepancy between the program output the base line files, diff will report the discrepancies.

Appendix B:

EBNF of tiny PL/0:

```
program ::= block "." .
block ::= const-declaration var-declaration statement.
constdeclaration ::= [ "const" ident "=" number {"," ident "=" number} ";" ].
var-declaration ::= [ "var" ident {"," ident} ";" ].
statement ::= [ ident ":" expression
                | "begin" statement { ";" statement } "end"
                | "if" condition "then" 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".
```

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.

Appendix C:

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.

Note: Not all of these error messages may be used, and you may choose to create some error messages of your own to more accurately represent certain situations.

Appendix D:

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;

```

Appendix E:

Symbol Table

Recommended data structure for the symbol.

```
typedef struct
{
    int kind;           // const = 1, var = 2, proc = 3
    char name[12];      // name up to 11 chars
    int val;            // number (ASCII value)
    int level;          // L level
    int addr;           // M address
} symbol;
```

```
symbol_table[MAX_SYMBOL_TABLE_SIZE];
```

For constants, you must store kind, name and value.

For variables, you must store kind, name, L and M.

For procedures, you must store kind, name, L and M.

Appendix F:

Grading Rubric

Deduction	Description
-100	Does not compile on Eustis
-100	Does not accept input filename from the command line.
-15	Does not handle file i/o or resource errors gracefully.
-20	Crashes on p1a.txt input, the algebraic and sequence test.
-20	Crashes on p2a.txt input, the if/then text.
-20	Crashes on p3a.txt input, the loop text.
-20	Crashes on p4a.txt input, the loop and extended variable name text.
- 5	-v, the virtual machine output is significantly different from the base. (Grader will allow spacing and minor format discrepancies.)

Note:

- 1 Crashes in each of the four cases defined above include not being able to process the symbols, variables, and data to produce “executable” VM code.
- 2 The lexer and parser/generator output are expected to be compliant with the output shown in the corresponding base files. That is the output from processing p1a.txt with the -l, -a, and -v options should closely match the base-p1a-lav.txt output file.