

COP 3402: System Software Fall 2016

Project #2 Parser & Code Generator for Tiny PL/0

Objective:

Your task is to implement a Recursive Descent Parser and an Intermediate Code Generator for tiny PL/0. In addition, you must create a compiler driver combining all compiler parts into a single program.

Your compiler driver must support the following compiler directives (command line arguments):

- 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:

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

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

Example of a program written in tiny PL/0:

```
var x, w;  
begin  
  x := 4;  
  read w;  
  if w > x then  
    w := w + 1;  
  else  
    w := x;  
  write 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 (Programming Project Module #1), the Parser/Code Generator (new in this assignment) and the Virtual Machine (#0).

The **Parser** is a program that reads in the output of the Scanner and parses the lexemes (tokens). It must be capable of reading in the tokens produced by your Scanner and produce, as output, a message that states whether the PL/0 program is well-formed (syntactically correct), that is, 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 is given in Appendix C. In addition, the Parser must generate the Symbol Table, which contains all of the variables, procedure names and constants 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, that is, the Symbol Table and parsed code. As output, it produces the assembly language for your Virtual Machine. This functionality will be interleaved with the Parser functionality (that is, 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

Submit via Webcourses:

1. Source code of the tiny- PL/0 compiler (must include the source code for: scanner, parser/code generator, virtual machine and compile driver).
2. A make file and text file (named “readme.txt”) with instructions on how to compile your project on Eustis, and how to use your compiler (instructions for all components must be included).
3. A document (in PDF format) with incorrectly formed PL/0 programs and screenshots of the output of your PL/0 compiler showing the corresponding error message to each incorrect PL/0 program. This document must demonstrate at least ten different errors (lexical and parser errors).
4. All files should be zipped into a single file.

Appendix A: Traces of Execution

Example 1

```
var x, y;  
begin  
  x := y + 56;  
end.
```

The output should look like:

1.- A print out of the token (internal representation) file:

```
29 2 x 17 2 y 18 21 2 x 20 2 y 4 3 56 18 22 19
```

2.- Print out the message “No errors, program is syntactically correct”.

3.- Print out the generated PM/0 (assembly) code.

4.- Print out the stack trace of the program on the virtual machine.

Example 2, if the input is:

```
var x, y;  
begin  
  x := y + 56;  
end      ← ( notice period expected after the “end” reserved word)
```

The output should look like:

1.- A print out of the token (internal representation) file:

```
29 2 x 17 2 y 18 21 2 x 20 2 y 4 3 56 18 22 19
```

2.- Print the message “Error number xxx, period expected”.

3.- Stop the compilation process. The VM is not executed since an error appeared at the parsing step.

Appendix B: EBNF of tiny PL/0

```
program ::= block "." .
block   ::= const-declaration var-declaration statement.
const-declaration ::= [ "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
                        | ε ] .
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 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 = "varsym" 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 != "semicolonsym" then ERROR;
        GET(TOKEN)
    end;
    STATEMENT
end;
```

```

procedure STATEMENT;
begin
    if TOKEN = "identsym" then begin
        GET(TOKEN);
        if TOKEN != "becomessym" then ERROR;
        GET(TOKEN);
        EXPRESSION
    end
    else if TOKEN = "callsym" then begin
        GET(TOKEN);
        if TOKEN != "identsym" then ERROR;
        GET(TOKEN)
    end
    else if TOKEN = "beginsym" then begin
        GET TOKEN;
        STATEMENT;
        while TOKEN = "semicolon" do begin
            GET(TOKEN);
            STATEMENT
        end;
        if TOKEN != "endsym" then ERROR;

        GET(TOKEN)
    end
    else if TOKEN = "ifsym" then begin
        GET(TOKEN);
        CONDITION;
        if TOKEN != "thensym" then ERROR;
        GET(TOKEN);
        STATEMENT
    end
    else if TOKEN = "whilesym" then begin
        GET(TOKEN);
        CONDITION;
        if TOKEN != "dosym" 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.

```
#define MAX_SYMBOL_TABLE_SIZE 100

typedef struct symbol {
    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 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.