# 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.

You have to name the file containing the main method `compile.c`. You also have to write a makefile that generates the executable file `./compile` by compiling and linking `compile.c` and any additional files you use to organize your code.

The executable file should accept two command line arguments as follows:

```
./compile <input> <output>
```

where `<input>` is the name of the containing the PL/0 code and `<output>` the name of the file containing the PM/0 code generated by your compiler.

Lex and parser errors should be written to standard output.

**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.
```

## Submission Instructions

Submit via Webcourses:

1. Source code of the tiny-PL/0 compiler (must include the source code for: scanner, parser/code generator).
2. A make file that generates the executable file `./compile`.
3. Ten files `incorrect1.pl0,...,incorrect9.pl0` containing incorrectly formed PL/0 programs and the ten files `error0.txt,...,error9.txt` containing the output of your PL/0 compiler showing the corresponding error messages. You must demonstrate ten different errors (lexical and parser errors).
4. Five files `correct0.pl0,…,correct4.pl0` of correctly formed PL/0 programs and the files `correct0.pm0,…,correct4.pm0` containing the corresponding generated PM/0 code. Five files `stacktrace0,…,stacktrace4.txt` containing the stacktrace generated by your virtual machine when run on the corresponding generated PM/0 code.
5. 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. Print out the message “No errors, program is syntactically correct”.

2. Write out the generated PM/0 (assembly) code to file.

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

2. Stop the compilation process.

## 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
                      ]
condition         ::= “odd” expression
                      | expression  rel-op  expression.
rel-op      ::= “=” | “<>” | “<” | “<=” |“>” | “>=”
expression  ::= [ “+” | “-” ] term { (“+” | “-”) term }
term        ::= factor { ( “*” | “/” ) factor }
factor      ::= ident | number | “(” expression “)”

number and ident are tokens with semantic values.

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. Note that ident and number are also terminal symbols.**

## 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 != "semicolomsym" 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 != "semicolomsym" then ERROR;
              GET(TOKEN)
        end;
        while TOKEN = "procsym" do begin
              GET(TOKEN);
              if TOKEN != "identsym” then ERROR;
              GET(TOKEN);
              if TOKEN != "semicolomsym" then ERROR;
              GET(TOKEN);
              BLOCK;
              if TOKEN != "semicolomsym" 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 = "semicolomsym" 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.