HW #4 (Parser Code Generation Complete Version)

COP3402: System Software

Fall 2025

Instructor: Jie Lin, Ph.D.

Due date: **Friday, November 21, 2025 at 11:59 PM ET**

Last updated: November 12, 2025

Disclaimer: This document specifies all deliverables and constraints for HW4. If anything is unclear, contact the instructor before implementing assumptions.

Updates: All official updates and clarifications will be posted as **Webcourses announcements**. Check Webcourses regularly for critical updates.

Due Date: Friday, November 21, 2025 at 11:59 PM ET

Submission Method: Submit only via Webcourses. Submissions by email, chat/DM, cloud links, or any other channel are not accepted.

Timestamp: The Webcourses submission timestamp is authoritative. All deadlines use U.S. Eastern Time.

Group Work: This is a group assignment. Submit a single solution per group that clearly lists every member in the header comment and in the Webcourses group submission.

Language Restriction: All code must be written in C and compiled with GCC on Eustis. Code in any other language will receive a score of zero.

CRITICAL - Testing Requirement: The instructor and TAs will **NOT** manually verify whether your generated assembly code is correct. We can only verify that your output is aligned to the specification format. **You MUST use your virtual machine to test that your generated instructions execute correctly.** If your VM does not run correctly on valid input, it means you are generating wrong instructions.

Contents

1 Academic Integrity, AI Usage, and Late Policy

1.1 Academic Integrity Expectations

If you plan to use AI tools while preparing your assignment, you must disclose this usage. Complete the **AI Usage Disclosure Form** provided with this assignment. If you used AI, include a separate markdown file describing:

- The name and version of the AI tool used.
- The dates used and specific parts of the assignment where the AI assisted.
- The prompts you provided and a summary of the AI output.
- How you verified the AI output against other sources and your own understanding.
- Reflections on what you learned from using the AI.

If you did not use any AI, check the appropriate box on the form. **Each team member must submit their own signed AI disclosure form individually in separate submissions (not as a group submission, since group submissions only allow one member to submit and will override other submissions).** Submit the signed form and the markdown file (if applicable) along with your assignment. Failure to disclose AI usage will be treated as academic dishonesty.

1.2 Plagiarism Detection

All submissions will be processed through JPlag, which detects the similarity score between you and the other students' submitted code. If the similarity score is above certain threshold, your code will be considered as plagiarism.

While AI tools may assist with brainstorming or initial code draft (if properly disclosed), the final submission must represent your own work and understanding. It is important to notice that if the similarity score is above the threshold, it does not matter if you have the AI disclosure or not, your program will be considered plagiarism. Also importantly, AI tends to draft the code in a similar way, if you just copy and paste, the similarity score will be very high.

1.3 Late Policy

Due vs. late. Anything submitted after the posted due date/time is late.

Late window. Late submissions are accepted for up to 48 hours after the due date/time; after that, the assignment is missed (score 0).

Penalty (points, not percentages). 5 points are deducted for each started 12-hour block after the due date/time (any fraction counts as a full block):

- 0:00:01–12:00:00 late → -5 points
- 12:00:01–24:00:00 late → -10 points

- 24:00:01–36:00:00 late → -15 points
- 36:00:01–48:00:00 late → -20 points
- After 48:00:00 late → Not accepted; recorded as missed (0 points)

Example: If the assignment is scored out of 100 points, the penalties above are deducted from your earned score.

Technical issues. Individual device/network problems do not justify exceptions—submit early and verify your upload.

2 Assignment Overview

HW4 represents the culmination of an incremental development process that began with HW1 and progressed through HW2 and HW3. This assignment extends your HW3 implementation by adding support for three critical PL/0 language features: **procedure calls**, **else clauses**, and **if statements**. Additionally, you must update your HW1 virtual machine (VM) to recognize and execute the **EVEN instruction** (OPR 0 11) as specified in Appendix A.

2.1 Incremental Development Context

This assignment builds upon your previous work:

- **HW1:** Implemented the PM/0 virtual machine that executes assembly code
- **HW2:** Developed the lexical analyzer (`lex.c`) that tokenizes PL/0 source code
- **HW3:** Created the parser and code generator (`parsercodegen.c`) for a subset of PL/0
- **HW4 (Current):** Extends HW3 with procedure calls, else clauses, and if statements; updates the VM for EVEN instruction support

2.2 Complete Compilation and Execution Workflow

Your complete system operates as a multi-stage pipeline:

1. **Lexical Analysis:** Your HW2 scanner (`lex.c`) reads a PL/0 source file and produces a token stream file (e.g., `tokens.txt`)
2. **Parsing and Code Generation:** Your HW4 parser (`parsercodegen_complete.c`) reads the token stream file, parses it according to the extended PL/0 grammar (including procedure calls, else clauses, and if statements), and generates PM/0 assembly code in the file `elf.txt`
3. **Execution:** Your updated HW1 virtual machine (vm.c) reads `elf.txt` and executes the assembly instructions. The VM must now support the EVEN instruction (OPR 0 11) for testing whether a number is even

4. **Success Condition:** If the input PL/0 source file is error-free and syntactically correct, the VM will execute successfully and produce the expected output

Key Point: This workflow demonstrates the complete compilation process from high-level source code to executable machine code, mirroring real-world compiler design.

3 Implementation Procedure

This assignment requires three C source files working together in a complete compilation pipeline. You will reuse components from previous assignments and extend them with new functionality.

3.1 Scanner (`lex.c`) from HW3 — No Changes Required

- **Reuse your HW3 scanner without modification.** The lexical analyzer remains unchanged for this assignment.
- Your `lex.c` file should already write token output to a file (as required in HW3).
- The output file contains the token stream that feeds into your parser.
- `lex.c` must accept exactly **ONE** command-line argument: the input PL/0 source file.
- **No modifications to the scanner are required for HW4.**

3.2 Parser/Code Generator (`parsercodegen_complete.c`) for HW4

- **Expand your HW3 `parsercodegen.c` file** to support the new grammar features (procedure calls, else clauses, and if statements).
- **Rename the file to `parsercodegen_complete.c`** to distinguish it from your HW3 submission.
- **Important distinction:** `parsercodegen_complete.c` requires **NO** command-line arguments.
- The input filename should be hard-coded in `parsercodegen_complete.c` (e.g., `tokens.txt` or `token_list.txt`—whatever your `lex.c` outputs).
- The parser reads the token file produced by `lex.c`, validates the grammar according to the extended PL/0 specification (see Section 4), and generates PM/0 assembly code (see Appendix A for the ISA specification).
- The output file (`elf.txt`) should contain PM/0 instructions in the format: `OP L M` (one instruction per line).
- **New grammar features to implement:**

- Procedure declarations and procedure calls
- Else clauses for if statements
- Complete if-then-else statement support

3.3 Virtual Machine (`vm.c`) from HW1 — EVEN Instruction Support

- **Modify your HW1 virtual machine** to recognize and execute the **EVEN instruction** (OPR 0 11).
- The EVEN instruction tests whether the value at the top of the stack is even. See Table 2 in Appendix A for the complete specification.
- **EVEN instruction behavior:**
  - Opcode: 02 (OPR)
  - L: 0
  - M: 11
  - Operation: `pas[sp]` ← `(pas[sp] % 2 == 0)`
  - Stack pointer update: No change to `sp`
  - Result: Replaces top of stack with 1 if the value is even, 0 if odd
- Your VM must read the `elf.txt` file produced by `parsercodegen_complete.c` and execute the instructions.
- **This VM update is essential** for executing PL/0 programs that use the `even` keyword in conditional expressions.

4 Grammar Specification

Your parser must accept exactly the productions in Figure 1. In EBNF notation, nonterminals are enclosed in angle brackets (e.g., `<program>`, `<statement>`), while terminals are keywords, symbols, or literals shown in quotes (e.g., `"const"`, `":="`) or unquoted special symbols (e.g., `empty` for epsilon).

PL/0 Grammar (EBNF) for HW4

```
<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> "fi"
    | "while" <condition> "do" <statement>
    | "read" <ident>
    | "write" <expression>
    | empty
    ]

<condition> ::= "even" <expression>
            | <expression> <rel-op> <expression>

<expression> ::= <term> { ("+" | "-") <term> }

<term> ::= <factor> { ("*" | "/" ) <factor> }

<factor> ::=
      <ident>
    | <number>
    | "(" <expression> ")"

<number> ::= <digit> { <digit> }
<ident>  ::= <letter> { <letter> | <digit> }

<rel-op>  ::= "=" | "<>" | "<" | "<=" | ">" | ">="
<digit>  ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<letter> ::= "a" | "b" | ... | "z" | "A" | "B" | ... | "Z"
```

Figure 1: Extended PL/0 Grammar for HW4 (includes procedures, call, and else)

Important notes about this grammar:

- **NEW for HW4:** This grammar now includes `<procedure-declaration>` which allows zero or more procedure definitions.
- **NEW for HW4:** The `<statement>` production now includes the `"call" <ident>` statement for procedure calls.
- **NEW for HW4:** The `"if"` statement now includes an `"else"` clause: `"if" <condition> "then" <statement> "else" <statement> "fi"`.
- Statement can be empty (epsilon production).
- Procedure declarations appear after variable declarations and before the main statement block.

Reserved words, symbols, and token type names must exactly match the table provided in HW2. Identifiers and numbers retain the lexemes emitted by the scanner.

5 Compilation Instructions

Compile and run on **Eustis**. The only permitted language is C.

Compilation Commands

```
// Compile the scanner to produce executable
gcc -O2 -std=c11 -o lex lex.c

// Compile the parser/code generator to produce executable
gcc -O2 -std=c11 -o parsercodegen_complete parsercodegen_complete.c

// Compile the virtual machine to produce executable
gcc -O2 -std=c11 -o vm vm.c
```

Note: The only permitted language is C. All three components (scanner, parser/code generator, and virtual machine) must compile successfully on Eustis.

Do not require additional libraries beyond the C standard library. Any makefiles or helper scripts must preserve the same compiler flags and target names.

6 Testing Instructions

Invoke programs from the terminal/command line on **Eustis** (the university grading server).

6.1 Scanner Requirements

- `lex.c` must accept exactly **ONE** command-line parameter: the path to the PL/0 source program to be scanned.
- Do not prompt for input.
- Reject incorrect argument counts with a helpful usage message and exit.
- The scanner should write all output to a file (not standard output for the token list).
- Never request additional input from the user.

6.2 Parser/Code Generator Requirements

- `parsercodegen_complete.c` requires **NO** command-line arguments.
- The input filename should be hard-coded (e.g., `tokens.txt` or `token_list.txt`).
- The parser reads the token file and generates assembly code.

6.3 Virtual Machine Requirements

- `vm.c` must accept exactly **ONE** command-line argument: the path to the assembly code file (e.g., `elf.txt`).
- The VM reads the assembly code file and executes the instructions.
- The VM must support the EVEN instruction (OPR 0 11) as specified in Appendix A.

6.4 Command-Line Usage Examples

To successfully run the entire compilation and execution pipeline:

Complete Pipeline Execution

```
// Step 1: Run the scanner to tokenize the PL/0 source file
./lex inputfile.txt

// Step 2: Run the parser/code generator to produce assembly code
./parsercodegen_complete

// Step 3: Run the virtual machine to execute the assembly code
./vm elf.txt
```

Pipeline Explanation:

1. `./lex inputfile.txt` reads the PL/0 source file and generates a token stream file (e.g., `tokens.txt`)
2. `./parsercodegen_complete` reads the token stream file and generates PM/0 assembly code in `elf.txt`
3. `./vm elf.txt` reads the assembly code file and executes the program

Explanation: This works because `lex` produces the token list file, and `parsercodegen` reads that token list file (with hard-coded filename) for parsing and code generation.

7 Input and Output Specifications

This section describes the exact input and output requirements for all three components: the scanner (lex.c), the parser/code generator (parsercodegen_complete.c), and the virtual machine (vm.c), as well as comprehensive error handling requirements. For complete working examples, see Appendix B (correct input), Appendix C (scanning error), and Appendix D (syntax error).

7.1 Scanner (lex.c) Input and Output

7.1.1 Input

- **Input file:** A PL/0 source program provided as a command-line argument
- **Format:** Plain text file containing PL/0 source code
- **Example:** `./lex input.txt`
- **See:** Appendix B for a complete example of correct PL/0 input

7.1.2 Output

- **Output file:** The scanner must write tokens to a file with a hard-coded filename
  - **Filename flexibility:** You may use `tokens.txt`, `token_list.txt`, or similar variations
  - **Important:** Your parser/code generator must read from the same filename that your scanner writes to
  - The exact filename is your choice, but it must be consistent between scanner and parser
- **Token format flexibility:** You may choose your own token format
  - **Example format 1:** Two lines per token (token type on line 1, token value on line 2 if applicable)
  - **Example format 2:** One line per token (token type and value on same line)

  - **Example format 3:** Any other format that your parser can read
  - **Important:** The format shown in Appendix B is just ONE example—your implementation may use a different format
  - Your scanner and parser must work together with whatever format you choose

- **Error handling:** If lexical errors are detected (identifier too long, number too long, or invalid symbols), replace the erroneous lexeme with token type **1** (skipsym) and continue processing the entire file
  - **See:** Appendix C for a complete example of scanning error handling
- **No terminal output required** for the scanner (all output goes to file)

7.2 Parser/Code Generator (parsercodegen_complete.c) Input and Output

7.2.1 Input

- **Input file:** Token list file with hard-coded filename (must match the output filename your scanner uses)
- **Format:** Token list produced by your scanner (must be able to read whatever format your scanner produces)
- **Important:** Your parser and scanner work together—they must agree on both filename and format

7.2.2 Output

- **Terminal output:**
  - If no errors: Display generated assembly code with line numbers and symbol table
    - **See:** Appendix B for an example of correct terminal output
  - If errors detected: Display appropriate error message (see Section 7.5)
    - **See:** Appendix C for scanning error output
    - **See:** Appendix D for syntax error output
- **File output (`elf.txt`):**
  - If no errors: Write assembly code in numeric format (one instruction per line: `OP L M`)
    - **See:** Appendix B for an example of correct `elf.txt` output
  - If errors detected: Write the same error message as displayed on terminal
    - **See:** Appendix C and Appendix D for error output examples

7.3 Virtual Machine (vm.c) Input and Output

7.3.1 Input

- **Input file:** Assembly code file provided as a command-line argument
- **Format:** Numeric PM/0 assembly code (one instruction per line: `OP L M`)
- **Example:** `./vm elf.txt`
- **Important:** The VM must read the file specified as the command-line argument
- **See:** Appendix B for an example of correct `elf.txt` input format

7.3.2 Output

- **Terminal output:**
  - Display the execution trace showing the state of the PM/0 machine after each instruction
  - Include program counter (PC), base pointer (BP), stack pointer (SP), and stack contents
  - Show input/output operations (READ and WRITE instructions)
  - **See:** Appendix A for PM/0 instruction set details
- **EVEN instruction support:**
  - The VM must recognize and execute the EVEN instruction (OPR 0 11)
  - Operation: Replace top of stack with 1 if the value is even, 0 if odd
  - **See:** Section 3.3 for EVEN instruction specification

7.4 Error Handling Requirements

Your parser/code generator must detect and report errors appropriately. When an error is detected, the program must:

1. Stop processing immediately (do not attempt to continue parsing or code generation)
2. Output the appropriate error message to the terminal
3. Write the same error message to `elf.txt`
4. Exit gracefully

7.5 Required Error Messages

Your `parsercodegen_complete.c` implementation **must** support the following error messages. Each error message must be output **exactly** as specified below (case-sensitive, exact wording):

7.5.1 Scanning Error (Lexical Error)

Scanning Error Message - REQUIRED

Error Message:
`Error: Scanning error detected by lexer (skipsym present)`

When to output: Token type 1 (skipsym) is found in the token list
Meaning: The scanner detected a lexical error (identifier too long, number too long, or invalid symbol)
Complete Example: See Appendix C for a detailed walkthrough showing:

- Input file with lexical error (identifier too long)
- Scanner output with token type 1
- Parser detection and error message output
- Terminal and `elf.txt` output

7.5.2 Syntax Errors (Grammar Violations)

The following error messages correspond to violations of the PL/0 grammar rules. See Appendix D for a complete example of syntax error handling.

Syntax Error Messages - ALL REQUIRED

Your parser must support ALL of the following error messages:

1. `Error: program must end with period`
   - **Complete Example:** See Appendix D for detailed walkthrough
2. `Error: const, var, read, procedure, and call keywords must be followed by identifier`
3. `Error: symbol name has already been declared`
4. `Error: constants must be assigned with =`
5. `Error: constants must be assigned an integer value`
6. `Error: constant and variable declarations must be followed by a semicolon`
7. `Error: undeclared identifier`
8. `Error: only variable values may be altered`
9. `Error: assignment statements must use :=`
10. `Error: begin must be followed by end`
11. `Error: if must be followed by then`
12. `Error: else must be followed by fi`
13. `Error: if statement must include else clause`
14. `Error: while must be followed by do`
15. `Error: condition must contain comparison operator`
16. `Error: right parenthesis must follow left parenthesis`
17. `Error: arithmetic equations must contain operands, parentheses, numbers, or symbols`
18. `Error: call statement may only target procedures`
19. `Error: procedure declaration must be followed by a semicolon`

Important: Each error message must be output **exactly** as shown above (case-sensitive, exact wording). The autograder will check for exact string matches.

7.5.3 Important Notes on Error Messages

- All error messages must begin with `Error:` followed by a space
- Error messages are case-sensitive and must match exactly as specified
- When an error is detected, output the error message to both terminal and `elf.txt`
- Do not output multiple error messages—stop at the first error encountered

8 Submission Instructions

All submissions must be made through Webcourses only. Do not submit via email, direct message, or any other method. Programs must be compilable and tested on **Eustis**. If your programs do not compile on Eustis, you will receive significant point deductions. Follow these instructions carefully to avoid penalties.

8.1 Code Requirements

- **Program names.** Submit exactly **three** source files: `lex.c`, `parsercodegen_complete.c`, and `vm.c`.

- **Header comment (copy/paste).** Place this header comment at the top of **all three** source files (`lex.c`, `parsercodegen_complete.c`, and `vm.c`).

REQUIRED Header Comment - Copy and Paste Into Your Code

Instructions: Copy the entire comment block below and paste it at the very top of **all three source files**: `lex.c`, `parsercodegen_complete.c`, and `vm.c`. Replace `<Full Name 1>, <Full Name 2>` with the actual names of all group members.

```
/*
Assignment:
HW4 - Complete Parser and Code Generator for PL/0
      (with Procedures, Call, and Else)

Author(s): <Full Name 1>, <Full Name 2>

Language: C (only)

To Compile:
  Scanner:
    gcc -O2 -std=c11 -o lex lex.c
  Parser/Code Generator:
    gcc -O2 -std=c11 -o parsercodegen_complete parsercodegen_complete.c
  Virtual Machine:
    gcc -O2 -std=c11 -o vm vm.c

To Execute (on Eustis):
  ./lex <input_file.txt>
  ./parsercodegen_complete
  ./vm elf.txt

where:
  <input_file.txt> is the path to the PL/0 source program

Notes:
  - lex.c accepts ONE command-line argument (input PL/0 source file)
  - parsercodegen_complete.c accepts NO command-line arguments
  - Input filename is hard-coded in parsercodegen_complete.c
  - Implements recursive-descent parser for extended PL/0 grammar
  - Supports procedures, call statements, and if-then-else
  - Generates PM/0 assembly code (see Appendix A for ISA)
  - VM must support EVEN instruction (OPR 0 11)
  - All development and testing performed on Eustis

Class: COP3402 - System Software - Fall 2025

Instructor: Dr. Jie Lin

Due Date: Friday, November 21, 2025 at 11:59 PM ET
*/
```

8.2 What to Submit

IMPORTANT: All items marked as **REQUIRED** MUST be submitted. Failure to submit any required items will result in automatic penalties as specified in the grading section.

8.2.1 Category A - Always Required:

- Your source code files (`lex.c`, `parsercodegen_complete.c`, and `vm.c`)
- The AI Usage Disclosure Form with your signature for each group member

8.2.2 Category B - Conditionally Required:

- **If you used AI:** A separate markdown file describing your AI usage - **REQUIRED when AI was used**

WARNING: The submission requirements above are unambiguous. Any missing required items will result in automatic point deductions. There are no exceptions for misunderstanding these requirements.

8.3 Submission Guidelines

- **Submit through Webcourses only.** Do not submit via email, direct message, or any other method.
- Submit your source files and any required documentation before the due date. Late submissions may incur penalties.
- Ensure all three source files (`lex.c`, `parsercodegen_complete.c`, and `vm.c`) compile without warnings using the commands in Section 5.
- **All programs must compile and run on Eustis.** If your programs do not compile on Eustis, you will receive significant point deductions.
- `lex.c` must accept exactly ONE command-line argument (the input PL/0 source file path). Print a usage message and exit for any other argument count.
- `parsercodegen_complete.c` must accept NO command-line arguments and use a hard-coded input filename (e.g., `tokens.txt`).
- `vm.c` must accept exactly ONE command-line argument (the assembly code file path, e.g., `elf.txt`).

9 Grading

This homework assignment is graded based on correctness and completeness through detailed manual examination and review of your implementation. Grading follows a deduction-based model: all students start with a maximum score of 100 points, and points are deducted for errors, missing requirements, compilation failures, academic integrity violations, and other issues discovered during the grading process.

9.1 Submission Files

The following deductions apply to missing or incorrect submission files:

- **-100 points:** Any of the three required source files (`lex.c`, `parsercodegen_complete.c`, or `vm.c`) is missing from the submission. No exceptions.
- **-100 points:** Program names are not `lex.c`, `parsercodegen_complete.c`, and `vm.c` exactly as specified. **Note:** Webcourses may automatically rename files during resubmission (e.g., `lex-1.c`, `lex-2.c`). Such automatic renaming by Webcourses will **NOT** result in deductions.
- **-100 points:** Missing the header usage instructions/comment block in any of the three source files (see Section 8.1 for detailed requirements). This is considered an academic integrity violation.
- **-100 points:** Author section in the header comment is not modified to include the actual names of all group members. This is considered an academic integrity violation.

9.2 Compilation and Execution

The following deductions apply to compilation and execution issues:

- **-100 points:** If **any one** of the three source files (`lex.c`, `parsercodegen_complete.c`, or `vm.c`) cannot compile on Eustis or does not compile from command line using the prescribed commands.
- **-100 points:** `lex.c` does not support exactly 1 command line parameter (the input PL/0 source file path).
- **-100 points:** `parsercodegen_complete.c` requires command line arguments instead of using a hard-coded input filename.
- **-100 points:** `vm.c` does not require exactly 1 command line argument (the assembly code file path, e.g., `elf.txt`).
- **-100 points:** Programs cannot read the required input files.
- **-100 points:** Code is written in a language other than C.

9.3 Academic Integrity

Academic integrity violations result in severe penalties:

- **-100 points:** Any instance of plagiarism, including direct copying from other students or sources, fabricated symbol table entries, fabricated implementations, or use of AI-generated code without substantial modification and understanding.
- **-100 points:** Missing or incomplete header comment blocks (see Section 8.1).
- **-100 points:** Failing to modify the author section in the header comment to include the actual names of all group members.
- **-100 points:** Implementation that does not follow the specified PL/0 grammar.
- **-100 points:** `lex.c` does not write output to a file (fails to create the token list file in the directory), OR `parsercodegen_complete.c` does not read any input file for the token list. This indicates failure to implement the core file I/O functionality required for the assignment.
- **-5 points:** Missing or incomplete AI Usage Disclosure Form.
- Plagiarism detection is performed using automated similarity analysis (see Section 9.8 at the end of this section for detailed information).

9.4 Code Generation and Output Format

The following deductions apply to code generation and output formatting issues:

- **-100 points:** `parsercodegen_complete.c` is not modified from HW3's source code. If your submission is identical or nearly identical to HW3, it indicates no work was done for HW4 and will receive this deduction.
- **-10 points:** Errors in LOD (load) and STO (store) instruction generation.
- **-5 points per instance:** Implementation does not correctly follow the grammar specification.
- **-5 points:** Incorrect PM/0 assembly output compared to expected results.
- **-5 points:** Output not formatted according to the ISA specification in Appendix A.

9.5 Virtual Machine

The following deductions apply to virtual machine implementation issues:

- **-100 points:** `vm.c` is not modified from HW1's source code to support the EVEN instruction (OPR 0 11). If your VM does not implement the EVEN instruction, it will receive this deduction.

9.6 Symbol Table Requirements

The following deductions apply to symbol table implementation errors:

- **-100 points:** Implementing symbol table management using any algorithm other than the marking algorithm (e.g., deletion algorithm). This is a violation of the assignment requirements. **You must use the marking algorithm only.**
- **-10 points:** Incorrect marking algorithm implementation.
- **-5 points:** Incorrect value field for variable symbols or procedure symbols.
- **-5 points:** Incorrect lexicographical level for symbols (must properly track nesting depth with procedures).
- **-5 points:** Incorrect address calculation for variables.
- **-5 points:** Any other symbol table implementation errors.

9.7 Error Handling

The following deductions apply to error handling issues:

- **-5 points:** Incorrect or missing error messages for invalid input.
- **-5 points:** Not detecting all required errors from HW2 scanner.

Note on Late Submissions: Late submissions are penalized according to Section 1.3.

Note on Additional Deductions: Graders reserve the right to apply additional 5-point deductions for significant errors discovered during manual review that do not fit into the predefined categories above. This grading rubric is comprehensive but not exhaustive.

9.8 Plagiarism Detection and Manual Verification

Important Notice: We take academic integrity seriously. All submissions undergo automated similarity analysis through JPlag. If similarity scores exceed established thresholds, the following process will be initiated:

1. **Manual Verification:** Submissions with high similarity scores will be manually reviewed by instructional staff.
2. **Zero Tolerance Policy:** If manual verification reveals direct copying from other students or direct usage of AI-generated code without substantial modification and understanding, the entire assignment will receive a score of zero (**-100 points**).
3. **AI Disclosure Irrelevant:** Having an AI disclosure form does not exempt submissions from plagiarism penalties if direct copying is detected.

4. **Similarity Patterns:** AI tools often generate similar code patterns. Simply copying and pasting AI-generated code will likely result in high similarity scores and trigger manual review.

The primary grading criterion is correctness and completeness combined with adherence to academic integrity standards. Follow all instructions precisely; deviations may result in additional deductions at the graders' discretion.

FINAL GRADE CALCULATION: Your final grade is calculated by starting with 100 points and subtracting all applicable deductions from the categories above. Academic integrity violations result in automatic assignment scores of zero (-100 points), regardless of technical performance.

A Instruction Set Architecture (ISA)

The PM/0 virtual machine supports nine opcodes. Each instruction is encoded by a three-number tuple $\langle OP, L, M \rangle$. The tables below summarize each opcode along with a brief description and pseudocode. See Table 2 for `OPR` sub-operations.

Note: Your parser and code generator must emit instructions in this ISA format. The generated assembly code file should contain one instruction per line in the format: `OP L M`.

Table 1: PM/0 Instruction Set (Core)

Opcode	OP Mnemonic	L	M	Description & Pseudocode
01	LIT	0	n	**Literal push.** `sp` $\leftarrow$ `sp` -1 `pas[sp]` $\leftarrow$ n
02	OPR	0	m	**Operation code;** see Table 2 for specific operations. *See OPR table for operation details*
03	LOD	n	a	**Load value to top of stack** from offset a in the AR n static levels down. `sp` $\leftarrow$ `sp` -1 `pas[sp]` $\leftarrow$ `pas[base(bp,`n`)` $-a$`]`
04	STO	n	o	**Store top of stack** into offset o in the AR n static levels down. `pas[base(bp,`n`)` $-o$`]` $\leftarrow$ `pas[sp]` `sp` $\leftarrow$ `sp` $+1$

05	CAL	n	a	**Call procedure** at code address a; create activation record.
				`pas[sp`-1`]` $\leftarrow$ `base(bp,`n`)`
				`pas[sp`-2`]` $\leftarrow$ `bp`
				`pas[sp`-3`]` $\leftarrow$ `pc`
				`bp` $\leftarrow$ `sp`-1
				`pc` $\leftarrow 499 - a$

06	INC	0	n	**Allocate n locals** on the stack. `sp` $\leftarrow$ `sp` $-n$
07	JMP	0	a	**Unconditional jump** to address a. `pc` $\leftarrow 499 - a$
08	JPC	0	a	**Conditional jump:** if value at top of stack is 0, jump to a; pop the stack. `if pas[sp]` $=$ `0 then pc` $\leftarrow 499 - a$ `sp` $\leftarrow$ `sp` $+1$
09	SYS	0	1	**Output integer** value at top of stack; then pop. `print(pas[sp])` `sp` $\leftarrow$ `sp` $+1$
09	SYS	0	2	**Read an integer** from stdin and push it. `sp` $\leftarrow$ `sp` -1 `pas[sp]` $\leftarrow$ `readInt()`
09	SYS	0	3	**Halt the program.** `halt`

Table 2: PM/0 Arithmetic and Relational Operations (OPR, opcode 02, L=0)

Opcode	OP Mnemonic	L	M	Description & Pseudocode

02	RTN	0	0	**Return from subroutine** and restore caller's AR.
				`sp` ← `bp` +1
				`bp` ← `pas[sp−2]`
				`pc` ← `pas[sp−3]`

02	ADD	0	1	**Addition.** `pas[sp+1]` ← `pas[sp+1]` + `pas[sp]` `sp` ← `sp` +1
02	SUB	0	2	**Subtraction.** `pas[sp+1]` ← `pas[sp+1]` − `pas[sp]` `sp` ← `sp` +1
02	MUL	0	3	**Multiplication.** `pas[sp+1]` ← `pas[sp+1]` ∗ `pas[sp]` `sp` ← `sp` +1
02	DIV	0	4	**Integer division.** `pas[sp+1]` ← `pas[sp+1]` / `pas[sp]` `sp` ← `sp` +1
02	EQL	0	5	**Equality comparison** (result 0/1). `pas[sp+1]` ← `(pas[sp+1]` == `pas[sp])` `sp` ← `sp` +1
02	NEQ	0	6	**Inequality comparison** (result 0/1). `pas[sp+1]` ← `(pas[sp+1]` ≠ `pas[sp])` `sp` ← `sp` +1
02	LSS	0	7	**Less-than comparison** (result 0/1). `pas[sp+1]` ← `(pas[sp+1]` < `pas[sp])` `sp` ← `sp` +1
02	LEQ	0	8	**Less-or-equal comparison** (result 0/1). `pas[sp+1]` ← `(pas[sp+1]` ≤ `pas[sp])` `sp` ← `sp` +1
02	GTR	0	9	**Greater-than comparison** (result 0/1). `pas[sp+1]` ← `(pas[sp+1]` > `pas[sp])` `sp` ← `sp` +1
02	GEQ	0	10	**Greater-or-equal comparison** (result 0/1). `pas[sp+1]` ← `(pas[sp+1]` ≥ `pas[sp])` `sp` ← `sp` +1
02	EVEN	0	11	**Even check** (result 0/1). `pas[sp]` ← `(pas[sp] % 2 == 0)`

Important: For this assignment (HW4), the grammar now includes procedures. You **must** implement the `CAL` (call procedure) and `RTN` (return from subroutine) instructions. The `L` parameter in `LOD` and `STO` instructions must correctly reflect the lexicographical level difference between the current scope and the variable's declaration scope.

B Appendix B: Complete Example for Correct Input

This appendix provides a complete, correct example showing the entire pipeline from PL/0 source code through lexical analysis to parser/code generation output. This example demonstrates what your implementation should produce.

B.1 Input: PL/0 Source Code

The following PL/0 program is the input file that `lex.c` reads:

Input File

```
var a, b;
procedure x;
    var q, w;
    begin
        q := 1;
    end;
begin
    a := 2;
    call x;
end.
```

B.2 Scanner Output: tokens.txt

After running `./lex input.txt`, your `lex.c` may produce a `tokens.txt` file similar to the following. **Note:** This is one possible implementation. Your token file format may differ based on your HW2 design, but it must contain equivalent information.

Example tokens.txt File

```
29
2 a
16
2 b
17
30
2 x
17
29
2 q
16
2 w
17
20
2 q
19
3 1
17
21
17
20
2 a
19
3 2
17
27
2 x
17
21
18
```

Hint: This shows one way to represent tokens. Token type 29 represents `var`, token type 2 represents identifiers (followed by the identifier name), token type 16 represents comma, token type 30 represents `procedure`, token type 27 represents `call`, etc. Your implementation may use a different format (e.g., a lexeme table), but it must convey the same token sequence to the parser.

B.3 Parser/Code Generator Terminal Output

After running `./parsercodegen_complete`, your program reads the token file and outputs the following to the terminal. This output shows both the generated assembly code and the symbol table:

Terminal Output

```
Assembly Code:

Line     OP   L   M
  0     JMP   0  18
  1     JMP   0   6
  2     INC   0   5
  3     LIT   0   1
  4     STO   0   3
  5     OPR   0   0
  6     INC   0   5
  7     LIT   0   2
  8     STO   0   3
  9     CAL   0   3
 10     SYS   0   3

Symbol Table:

Kind | Name        | Value | Level | Address | Mark
---------------------------------------------------
   2 |           a |     0 |     0 |       3 |    1
   2 |           b |     0 |     0 |       4 |    1
   3 |           x |     0 |     0 |       3 |    1
   2 |           q |     0 |     1 |       3 |    1
   2 |           w |     0 |     1 |       4 |    1
```

Explanation:

- The assembly code shows the PM/0 instructions in human-readable format with line numbers.
- The symbol table shows all declared variables and procedures with their properties.
- Kind 2 indicates a variable, Kind 3 indicates a procedure.
- Level indicates the lexicographical nesting depth (0 for main program, 1 for procedures declared in main, etc.).
- Address shows the memory offset for variables or the code address for procedures.
- Mark indicates the symbol's availability during parsing/code generation. Mark = 0 means the symbol is available for use at the current point in instruction generation. Mark = 1 means the symbol is NOT available (unavailable) at the current lexicographical level—this marking mechanism is directly related to lexical scoping depth.

- Note the use of CAL (call procedure) and RTN (return from subroutine) instructions for procedure support.

B.4 Parser/Code Generator File Output: elf.txt

In addition to terminal output, `parsercodegen_complete` must write the executable code to a file (e.g., `elf.txt`). This file contains the same instructions but in numeric format (opcode, L, M) without line numbers or mnemonics:

File Output: elf.txt

```
7 0 18
7 0 6
6 0 5
1 0 1
4 0 3
2 0 0
6 0 5
1 0 2
4 0 3
5 0 3
9 0 3
```

Important Notes:

- The first instruction is `7 0 18` (JMP 0 18), which jumps to the main program code (skipping procedure declarations).
- Each line contains three space-separated integers: opcode, L-value, and M-value.
- This file format is required for the PM/0 virtual machine to execute your code.
- Refer to Appendix A for the complete ISA specification and opcode mappings.

B.5 Virtual Machine Terminal Output

After running `./vm elf.txt`, your virtual machine reads the assembly code file and executes the instructions, producing the following execution trace:

VM Terminal Output

```
        L   M   PC  BP  SP  stack
Initial values: 499 466 467
JMP 0   18  481 466 467
INC 0   5   478 466 462 0    0    0    0    0
LIT 0   2   475 466 461 0    0    0    0    0    2
STO 0   3   472 466 462 0    0    0    2    0
CAL 0   3   496 461 462 0    0    0    2    0
JMP 0   6   493 461 462 0    0    0    2    0
INC 0   5   490 461 457 0    0    0    2    0    | 466  466  469  0    0
LIT 0   1   487 461 456 0    0    0    2    0    | 466  466  469  0    0    1
STO 0   3   484 461 457 0    0    0    2    0    | 466  466  469  1    0
RTN 0   0   469 466 462 0    0    0    2    0
SYS 0   3   466 466 462 0    0    0    2    0
```

Explanation:

- Each line shows the instruction being executed and the state of the PM/0 machine.
- The stack is displayed with the pipe symbol (|) separating activation records.
- CAL creates a new activation record for the procedure call.
- RTN returns from the procedure and restores the previous activation record.
- The execution trace demonstrates proper procedure call and return behavior.

B.6 Summary

This complete example demonstrates:

1. How `lex.c` transforms PL/0 source code into a token stream.
2. How `parsercodegen_complete.c` reads the token stream and generates PM/0 assembly code.
3. How `vm.c` executes the generated assembly code.
4. The required output formats for scanner, parser/code generator, and virtual machine.
5. The symbol table structure and content including procedures.
6. Proper procedure call and return execution with activation records.

Use this example to verify your implementation produces correct output for valid PL/0 programs with procedures.

C Appendix C: Scanning Error Example

This appendix demonstrates how your implementation should handle input files containing **lexical/scanning errors**. The key principle is that **the scanner (lex.c) continues processing the entire input file even when errors are detected**, but represents errors as a special error token (token type 1). The parser (parsercodegen_complete.c) then detects this error token and halts immediately.

C.1 Input: PL/0 Source Code with Error

The following PL/0 program contains a lexical error (identifier name too long):

Input File with Error

```
const z = 5;
var xcjioasunihujacioj, y;
begin
    xcjioasunihujacioj := y * 2;
end.
```

Error: The identifier `xcjioasunihujacioj` is 18 characters long, which exceeds the maximum allowed length of 11 characters.

C.2 Scanner Behavior and Output

Important: Your `lex.c` scanner must **NOT** stop when it encounters an error. Instead, it should:

1. Continue processing the entire input file
2. Replace any erroneous lexeme with token type **1** (error token/skipsym)
3. Generate a complete token list including the error token(s)

After running `./lex input.txt`, your `lex.c` produces the following `tokens.txt` file:

tokens.txt with Error Token

```
28
2 z
8
3 5
17
29
1
16
2 y
17
20
1
19
2 y
6
3 2
17
21
18
```

Explanation:

- Token type **1** represents an error token (skipsym)
- The first occurrence of token **1** (line 7) represents the identifier that was too long in the variable declaration
- The second occurrence of token **1** (line 11) represents the same identifier in the assignment statement
- All three scanner errors (name too long, number too long, invalid symbols) are represented by token type **1**
- The scanner continues processing and generates tokens for all valid lexemes

C.3 Parser/Code Generator Behavior

When `parsercodegen_complete.c` reads the `tokens.txt` file and encounters token type **1**, it must:

1. Immediately recognize that the scanner detected an error
2. Stop parsing and code generation
3. Output a unified error message
4. Write the same error message to both terminal and file

C.4 Terminal Output

After running `./parsercodegen`, the terminal displays:

Terminal Output for Error

```
Error: Scanning error detected by lexer (skipsym present)
```

C.5 File Output: elf.txt

The `elf.txt` file contains the same error message:

elf.txt for Error

```
Error: Scanning error detected by lexer (skipsym present)
```

C.6 Error Handling Summary

C.6.1 Scanner (lex.c) Responsibilities

- Detect all three types of lexical errors:
  1. Identifier name too long (exceeds 11 characters)
  2. Number too long (exceeds maximum allowed digits)
  3. Invalid symbols (characters not in the PL/0 alphabet)
- Replace each error with token type **1** (skipsym)
- **Continue processing** the entire input file
- Generate a complete token list including error tokens

C.6.2 Parser/Code Generator (parsercodegen_complete.c) Responsibilities

- Read the token list from the file
- Check for the presence of token type **1**
- If token type **1** is found:
  - Stop immediately (do not attempt to parse)
  - Output the unified error message to terminal
  - Write the same error message to `elf.txt`
- The exact error message must be: `Error: Scanning error detected by lexer (skipsym present)`

C.6.3 Key Principle

All three scanner errors produce the same unified error message from the parser. The parser does not need to distinguish between different types of lexical errors—it only needs to detect that an error occurred (presence of token type 1) and halt with the appropriate message.

D Appendix D: Syntax Error Example

This appendix demonstrates how your implementation should handle **syntax errors** (grammar violations). Unlike Appendix C which dealt with **scanning errors** (lexical errors detected by `lex.c`), this appendix focuses on **syntax errors** (grammar violations detected by `parsercodegen_complete.c`).

D.1 Key Distinction: Scanning vs. Syntax Errors

- **Scanning Errors (Appendix C):** Lexical errors detected by the scanner (e.g., identifier too long, number too long, invalid symbols). The scanner represents these as token type 1 (skipsym), and the parser detects the presence of this error token.
- **Syntax Errors (This Appendix):** Grammar violations detected by the parser during parsing (e.g., missing period, missing semicolon, incorrect statement structure). The scanner produces valid tokens, but the token sequence does not match the grammar rules.

D.2 Input: PL/0 Source Code with Syntax Error

The following PL/0 program contains a syntax error (missing period at the end):

Input File with Missing Period

```
var x, y;
begin
    read y;
    x := y * 2;
end
```

Error: According to the grammar specification (see Figure 1), a program must follow the production:

```
<program> ::= <block> "."
```

The period symbol after `end` is **required** but missing in this input.

D.3 Scanner Output: tokens.txt

The scanner (`lex.c`) processes this input and produces valid tokens. **Note:** The scanner does not detect this error because all lexemes are valid—the error is in the grammar structure, not in individual tokens.

tokens.txt for Syntax Error Example

```
29
2 x
16
2 y
17
20
32
2 y
17
2 x
19
2 y
6
3 2
17
21
```

Explanation:

- All tokens are valid (no token type 1 present)
- Token type 21 represents `end`
- Token type 18 (period) is missing at the end
- The scanner successfully tokenized all lexemes
- The error will be detected by the parser, not the scanner

D.4 Parser Behavior

When `parsercodegen_complete.c` parses the token list:

1. It successfully parses the variable declaration (`var x, y;`)
2. It successfully parses the `begin...end` block
3. After parsing `end` (token 21), it expects a period (token 18) according to the grammar rule: `<program> ::= <block> "."`
4. It encounters end-of-file instead of the required period
5. It detects a syntax error and halts

D.5 Terminal Output

After running `./parsercodegen`, the terminal displays:

Terminal Output for Syntax Error

```
Error: program must end with period
```

D.6 File Output: elf.txt

The `elf.txt` file contains the same error message:

elf.txt for Syntax Error

```
Error: program must end with period
```

D.7 Syntax Error Handling Summary

D.7.1 Scanner (lex.c) Behavior

- Processes all lexemes successfully
- Produces valid tokens (no token type 1)
- Does **NOT** detect syntax errors
- Syntax error detection is the parser's responsibility

D.7.2 Parser (parsercodegen_complete.c) Responsibilities

- Parse tokens according to the grammar rules
- Detect when the token sequence violates grammar rules
- For missing period error specifically:
  - After parsing the block, expect token type 18 (period)
  - If period is missing, output the specific error message
  - Halt parsing and code generation
- Output error message to both terminal and `elf.txt`
- The exact error message must be: `Error: program must end with period`

D.7.3 Key Principle

The scanner handles lexical errors by emitting error tokens (token type 1). The parser handles both scanning errors (by detecting token type 1) and syntax errors (by detecting grammar violations during parsing). Each type of error produces a specific, appropriate error message.

E Appendix E: Symbol Table Structure

This appendix provides the recommended data structure for implementing the symbol table in your `parsercodegen_complete.c` implementation.

E.1 Recommended Symbol Table Structure

Symbol Table Data Structure - RECOMMENDED

Important: While you may use alternative data structures (e.g., linked lists, dynamically allocated arrays), the **structure fields must remain the same** as shown below. The autograder expects these specific field names and types.

```
#define MAX_SYMBOL_TABLE_SIZE 500

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
    int mark;       // to indicate unavailable or deleted
} symbol;

symbol symbol_table[MAX_SYMBOL_TABLE_SIZE];
```

Field Descriptions:

- `kind`: Type of symbol (1 = constant, 2 = variable, 3 = procedure)
- `name[10]`: Symbol name (up to 11 characters including null terminator)
- `val`: For constants, stores the constant value
- `level`: Lexicographical level (L in PM/0 instructions)
- `addr`: Memory address (M in PM/0 instructions)
- `mark`: Indicator of symbol availability during parsing/code generation. Mark = 0 means the symbol is available for use. Mark = 1 means the symbol is NOT available (unavailable) at the current point in instruction generation. This marking mechanism is directly related to lexicographical level (lexical scoping depth).

E.2 Symbol Table Size

- **Maximum size:** 500 symbols (`MAX_SYMBOL_TABLE_SIZE = 500`)
- **Sufficiency:** A table size of 500 is sufficient for all test cases in this assignment
- **Recommendation:** Use the simple array-based structure shown above for ease of implementation
- **Alternative structures:** You may use linked lists or dynamically allocated memory if you prefer, but the structure fields must remain the same

E.3 Required Storage for Different Symbol Types

E.3.1 For Constants (kind = 1)

When storing a constant symbol, you **must** populate the following fields:

Required Fields for Constants

- `kind = 1`
- `name`: The constant's identifier name
- `val`: The constant's integer value
- `level`: The lexicographical level where the constant is declared (0 for main program, increases with procedure nesting)
- `mark = 0` (initially available)

Note: The `addr` field is not used for constants.

E.3.2 For Variables (kind = 2)

When storing a variable symbol, you **must** populate the following fields:

Required Fields for Variables

- `kind = 2`
- `name`: The variable's identifier name
- `level`: The lexicographical level where the variable is declared (0 for main program, increases with procedure nesting)
- `addr`: The variable's memory address (M value for LOD/STO instructions)
- `mark = 0` (initially available)

Note: The `val` field is not used for variables.

E.3.3 For Procedures (kind = 3)

When storing a procedure symbol, you **must** populate the following fields:

Required Fields for Procedures

- `kind = 3`
- `name`: The procedure's identifier name
- `level`: The lexicographical level where the procedure is declared (0 for main program, increases with procedure nesting)
- `addr`: The code address where the procedure begins (used as M value for CAL instruction)
- `mark = 0` (initially available)

Note: The `val` field is not used for procedures.

E.4 Symbol Table Operations

Your implementation should support the following operations:

- **Insert:** Add a new symbol to the table (check for duplicate names first)
- **Lookup:** Search for a symbol by name
- **Mark:** Set the `mark` field to 1 to indicate the symbol is NOT available (unavailable) at the current point in instruction generation. The marking mechanism is directly related to lexicographical level—when exiting a scope (lexical level), symbols at that level are marked as unavailable. Mark = 0 means available, Mark = 1 means unavailable.

F Appendix F: Pseudocode

This appendix provides pseudocode for implementing the recursive-descent parser and code generator. The pseudocode closely follows the grammar structure and demonstrates the general approach for parsing and code generation.

CRITICAL WARNING: This pseudocode is NOT an exact implementation of the grammar!

Important points:

- The pseudocode provides a **starting point** but is **NOT 100% accurate**
- You **MUST** carefully study the grammar specification (Figure 1) and understand how it differs from this pseudocode
- **Blindly translating this pseudocode to C will result in deductions**
- You are responsible for modifying and adapting the pseudocode to correctly match the grammar
- Use this as a **guide**, not as a complete solution
- The pseudocode is a good starting point, but you must verify every detail against the grammar

Each function below is presented in a separate box for clarity. Study each function carefully and compare it with the corresponding grammar production.

F.1 Symbol Table Helper Function

```
SYMBOLTABLECHECK (string)
    linear search through symbol table looking at name
    return index if found, -1 if not
```

F.2 PROGRAM Function

```
PROGRAM
    BLOCK
    if token != periodsym
        error
    emit HALT
```

F.3 BLOCK Function

```
BLOCK
    CONST-DECLARATION
    numVars = VAR-DECLARATION
    PROCEDURE-DECLARATION
    emit INC (M = 3 + numVars)
    STATEMENT
```

F.4 CONST-DECLARATION Function

```
CONST-DECLARATION
    if token == const
        do
            get next token
            if token != identsym
                error
            if SYMBOLTABLECHECK (token) != -1
                error
            save ident name
            get next token
            if token != eqlsym
                error
            get next token
            if token != numbersym
                error
            add to symbol table (kind 1, saved name, number, 0, 0)
            get next token
        while token == commasym
        if token != semicolonsym
            error
        get next token
```

F.5 VAR-DECLARATION Function

```
VAR-DECLARATION - returns number of variables
    numVars = 0
    if token == varsym
        do
            numVars++
            get next token
            if token != identsym
                error
            if SYMBOLTABLECHECK (token) != -1
                error
            add to symbol table (kind 2, ident, 0, 0, var# + 2)
            get next token
        while token == commasym
        if token != semicolonsym
            error
        get next token
    return numVars
```

F.6 PROCEDURE-DECLARATION Function

```
PROCEDURE-DECLARATION
    while token == procsym
        get next token
        if token != identsym
            error
        get next token
        if token != semicolonsym
            error
        get next token
        add to symbol table (kind 3, ident, level, codeIndex)
        BLOCK(level + 1)
        if token != semicolonsym
            error
        get next token
```

F.7 STATEMENT Function

```
STATEMENT
    if token == identsym
        symIdx = SYMBOLTABLECHECK (token)
        if symIdx == -1
            error
        if table[symIdx].kind != 2 (not a var)
            error
        get next token
        if token != becomessym
            error
        get next token
        EXPRESSION
        emit STO (M = table[symIdx].addr)
        return
    if token == callsym
        get next token
        if token != identsym
            error
        idx = SYMBOLTABLECHECK(token)
        if idx == -1
            error
        if symbolTable[idx].kind != 3
            error
        emit CAL (L, symbolTable[idx].addr)
        get next token
        return
    if token == beginsym
        do
            get next token
            STATEMENT
        while token == semicolonsym
        if token != endsym
            error
        get next token
        return
    if token == ifsym
        get next token
        CONDITION
        jpcIdx = current code index
        emit JPC
        if token != thensym
            error
```

```
        get next token
        STATEMENT
        code[jpcIdx].M = current code index
        if token != fisym
            error
        get next token
        return
    if token == whilesym
        get next token
        loopIdx = current code index
        CONDITION
        if token != dosym
            error
        get next token
        jpcIdx = current code index
        emit JPC
        STATEMENT
        emit JMP (M = loopIdx)
        code[jpcIdx].M = current code index
        return
    if token == readsym
        get next token
        if token != identsym
            error
        symIdx = SYMBOLTABLECHECK (token)
        if symIdx == -1
            error
        if table[symIdx].kind != 2 (not a var)
            error
        get next token
        emit READ
        emit STO (M = table[symIdx].addr)
        return
    if token == writesym
        get next token
        EXPRESSION
        emit WRITE
        return
```

F.8 CONDITION Function

```
CONDITION
    if token == oddsym
        get next token
        EXPRESSION
        emit ODD
    else
        EXPRESSION
        if token == eqlsym
            get next token
            EXPRESSION
            emit EQL
        else if token == neqsym
            get next token
            EXPRESSION
            emit NEQ
        else if token == lessym
            get next token
            EXPRESSION
            emit LSS
        else if token == leqsym
            get next token
            EXPRESSION
            emit LEQ
        else if token == gtrsym
            get next token
            EXPRESSION
            emit GTR
        else if token == geqsym
            get next token
            EXPRESSION
            emit GEQ
        else
            error
```

F.9 EXPRESSION Function

```
EXPRESSION
    if token == minussym
        get next token
        TERM
        emit NEG
        while token == plussym || token == minussym
            if token == plussym
                get next token
                TERM
                emit ADD
            else
                get next token
                TERM
                emit SUB
    else
        if token == plussym
            get next token
        TERM
        while token == plussym || token == minussym
            if token == plussym
                get next token
                TERM
                emit ADD
            else
                get next token
                TERM
                emit SUB
```

F.10 TERM Function

```
TERM
    FACTOR
    while token == multsym || token == slashsym || token == modsym
        if token == multsym
            get next token
            FACTOR
            emit MUL
        else if token == slashsym
            get next token
            FACTOR
            emit DIV
        else
            get next token
            FACTOR
            emit MOD
```

F.11 FACTOR Function

```
FACTOR
    if token == identsym
        symIdx = SYMBOLTABLECHECK (token)
        if symIdx == -1
            error
        if table[symIdx].kind == 1 (const)
            emit LIT (M = table[symIdx].value)
        else (var)
            emit LOD (M = table[symIdx].addr)
        get next token
    else if token == numbersym
        emit LIT
        get next token
    else if token == lparentsym
        get next token
        EXPRESSION
        if token != rparentsym
            error
        get next token
    else
        error
```