COP 3402: System Software

Prof. Montagne

July 22, 2018

PL/0 User Manual

*An introduction to compile-time and run-time operations*

By Manuel Govea and Christian Whittted

Table of Contents

**Type chapter level (level 1) 1**

**Type chapter level (level 2) 2**

Type chapter title (level 3) 3

**Type chapter level (level 1) 4**

**Type chapter level (level 2) 5**

Type chapter title (level 3) 6

# Introduction

**Foreword**

This project was undertaken in the Summer Semester of 2018, as a part of the University of Central Florida, COP 3402 (Systems Software) course, under the instruction of Professor Euripides Montagne by Manuel Govea and Christian Whitted. It serves as the capstone project of the course.

## **Abstract**

The following documentation serves as a User Manual, design outline, and project summation for the Govea-Whitted implementation of the PM/0 virtual machine, lexical analyzer, and parser for the PL\0 Programming Language. For further reading on the code behind the virtual machine, lexer, and parser, see the Source Code Documentation section.

**Public GitHub Repository:**

<https://github.com/TheVoidLives/pl0>

# PM\0 Virtual Machine

Design:

Usage:

# PL\0 Programming Language

# Compilation Guide

**Description:** The following section outlines the proper procedure to successfully compile the PL\M\0 compiler and virtual machine.

**Materials:** This project consists of two files:

1. hw4compiler.c
2. pl0.h

**Note:** The pl0.h file must be in the same directory as the hw4compiler.c file for compilation to be successful. Attempting to compile this project without the pl0.h file will result in compilation errors.

**Instructions:** To compile this project, open a terminal, navigate to the folder containing the materials outline above using the appropriate command (typically ‘cd’). Once complete, enter one of the following commands:

1. gcc hw4compiler.c
2. gcc -o hw4compiler hw4compiler.c

An executable file will be created and placed in the local folder. This executable can then be run from the local directory.

**Note:** Option (1) will result in an ‘a.out’ file being created, while option (2) will result in a ‘hw4compiler’ file being created.

Runtime Guide

**Description:** The following section outlines the proper procedure to successfully run the PL\M\0 compiler and virtual machine, once it has been compiled. Additionally, it assumes that the user has access to a valid \*NIX based environment, a valid sh-terminal (bash, fsh, zsh, etc), and the materials outlined within the *Compilation Guide*.

**Materials:** The following files are required to run the hw4compiler:

1. The compiled hw4compiler executable file.
   1. File Name: *a.out* or *hw4compiler*.
2. A valid PL\0 Program.

**Note:** Attempting to run an invalid PL\0 Program will result in an error of the appropriate type being displayed. In order to prevent this, the user must ensure that the program fits the grammar of the pl0 language as outlined in the PL\0 EBNF section and the syntax of the pl0 language as outlined in the PL\0 Programming Language section.

**Instructions:** In order to run the PL\0 Compiler and PM\0 Virtual Machine, open an applicable sh-based terminal and navigate the folder containing the compiled PL\M\0 executable file using the appropriate command (typically ‘cd’). If the PL\0 compiler and virtual machine has not been compiled, see the *Compilation Guide* above..At this point, the executable file can be run using the period forward slash operator (./) followed by the PL\M\0 executable file, a space, and the an appropriate program file (as input).

**Minimum Runtime Arguments:**

1. ./a.out lexicalInput.txt

2. ./hw4compiler lexicalInput.txt

**Note:** When run with the minimum number of arguments. no output will be displayed unless the program calls the PL0 "write" function, or an error is encountered. For further input, an output flag must be specified. See Glossary of Runtime Flags for more details.

# 

# PL\0 EBNF

**Description:** An overview of the EBNF (grammar) of the PL/0 programming language.

program ::= block "**.**" **.**

block ::= const-declaration var-declaration procedure-declaration statement**.**

constdeclaration ::= ["**const**" ident "**=**" number {"**,**" ident "**=**" number} "**;**"]**.**

var-declaration ::= [ "**int** "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**" expression

| **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.**

# Glossary of Runtime Flags

**Example Runtime Arguments:**

1. ./a.out -l -a -v lexicalInput.txt

2. ./hw4compiler -l -a -v lexicalInput.txt

## **Runtime Flags:**

-l | Prints the Source Code, Lexeme Table, and Lexeme List to stdout

and the out.txt file (in the local directory).

-a | Prints the output of the parser (generated assembly code) to stdout

and the out.txt file (in the local directory).

-v | Prints the Virtual Machine execution trace to stdout and the out.txt

file (in the local directory).

## **Debugging Flags:**

-x | Explicitly sets the Lexer input file. This flag must be thrown if the -m flag is

thrown. The name of the lexical input file must follow the -x flag.

Example Call: ./a.out -l -x lexicalInput.txt -a -v

-m | Explicitly sets the VM input file. Used to test the VM functionality independently

of the Lexer and Parser. The name of the vminput file must follow the -m flag.

Example Call: ./a.out -l -a -v -x lexicalInput.txt -m vmInput.txt

# Source Code Documentation

*A look at pl0.h*



// The following header serves as a reference doc. for all functionality

// within the implementation of a PL0 compiler found within.

//

// Authors:

// Manuel Govea

// Christian Whitted

//

// Course:

// COP3402C - Summer 2018

// Prof. Montagne

//

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <math.h>

#ifndef PL0

/\* VM Constants \*/

#define MAX\_STACK\_HEIGHT 2000 // Maximum number of ARs (Activation Records) in STACK

#define MAX\_CODE\_LENGTH 500 // Maximum number of Instructions held by the IR Register

#define MAX\_LEXI\_LEVEL 3 // Number of Lexical Levels

#define MAX\_REG 16 // Max

/\*Lexer Constants \*/

#define MAX\_LENGTH\_IDENTIFIER 11 // Maximum length of an ID in PL0

#define MAX\_LENGTH\_INTEGER 5 // Maximum length of an integer in PL0

#define SIZE\_ALPHABET 26 // Size of Keyword Trie (Num. allowable values)

/\* Parser Constants\*/

#define MAX\_SL\_LENGTH 500

#define OUTPUT\_FILE "out.txt"

/\*Debug Flag\*/

#define DEBUG 0

// COMPILER OUTPUT FILE

FILE \*output;

//----------------------------------//

// Lexer Globals //

//----------------------------------//

/\* Enum Declaration - Lexer

\* Symbolic representation of the Token types identified by the

\* lexical analyzer.

\* \*/

typedef enum {

nulsym = 1,

identsym,

numbersym,

plussym,

minussym,

multsym,

slashsym,

oddsym,

eqsym,

neqsym,

lessym,

leqsym,

gtrsym,

geqsym,

lparentsym,

rparentsym,

commasym,

semicolonsym,

periodsym,

becomessym,

beginsym,

endsym,

ifsym,

thensym,

whilesym,

dosym,

callsym,

constsym,

varsym,

procsym,

writesym,

readsym,

elsesym

} token\_type;

// Lexeme Table - Linked List Implementation

typedef struct TableEntry

{

char word[MAX\_LENGTH\_IDENTIFIER + 1];

int ID;

struct TableEntry \*next;

} TableEntry;

TableEntry \*tableHead = NULL, \*tableTail = NULL;

// Keyword Trie - Lexer

typedef struct TrieNode

{

struct TrieNode \*children[SIZE\_ALPHABET];

int keyValue;

} TrieNode;

//----------------------------------//

// Parser globals //

//----------------------------------//

// Struct representation of a Symbol within the Symbol Table

typedef struct Symbol

{

int kind; // const = 1, var = 2

char name [11]; // Name of var/const (length 10. name[11] = \0)

int value; // Value of var/const

int level; // Lexicographical level

int address; // Offset from base address

int mark; // is it accessible? 0 = yes, 1 = no

} Symbol;

TableEntry \*token = NULL; // Current Token to be Parsed

Symbol toBeInserted; // Symbol to be inserted (SELF DOCUMENTATION FTW!)

Symbol \*badSearch = NULL; // check to determine if you found the symbol you where looking for

Symbol symbolTable[MAX\_SL\_LENGTH]; // Symbol Table

int lastIndexOfST = 0; // Keeps track of the last element in the symbol table

int errHandle = 0; // Simple Err Handle Value.

int currLexical = -1; // Lexical Level of the current Program.

int varLexical = 0; // Lexical Level of the Variable we are retrieving from the symbol table

int searchLexical = 0; // |varLexical - currlexical| --> Lexical Level to store or load a word into stack

int curInsertionOffset = 4;// M - from VM. Used to Store things into the symbol table

int currRegPos = 0; // Current Registry. Load and store into this current position.

int currPC = 0; // Current index of IR[] for code generation

//----------------------------------//

// pm/0 (vm) globals //

//----------------------------------//

// Instruction - VM

typedef struct Instruction

{

int OP;

int REG;

int L;

int M;

} Instruction;

/\* VM Registers & Stack \*/

Instruction IR [MAX\_CODE\_LENGTH]; // Instruction Register

int STACK [MAX\_STACK\_HEIGHT]; // Stack

int REG [MAX\_REG]; // Registers 0 - 15 (16)

int SP = 0; // Stack Pointer

int BP = 1; // Base pointer

int PC = 0; // Program Counter

int PPC = 0; // Previous Program Counter (printing purposes)

int Halt = 0; // End Of Program Flag

int numAR = 0; // Number of activation records currently open

//----------------------------------//

// lexer function declarations //

//----------------------------------//

// Invokes the Lexical Analyzer for the passed file.

int lexer (char\* filename, int printFlag);

// Returns 1 for invalid symbols. 0 otherwise.

int invalidSymbol(char c);

// Error Handler. Prints error to screen and exits PL0 Compiler.

void handleError(int err);

// Helper Function: Inserts a Lexeme (Word and Identifier) into the Lexeme Table

TableEntry \*insertTableEntry(TableEntry \*tail, char \*word, int id);

// Helper Function: Returns a head node for the Keyword Trie.

TrieNode \*getTrieNode();

// Helper Function: Inserts a Keyword into the Static keyword trie.

void insertKeyword(TrieNode \*head, char \*key, int enumValue);

// Returns the (int) token-type of a given "word" if applicable.

int searchKeyword(TrieNode \*head, char \*key);

// Helper Function: Initializes the Keyword Trie

void initKeywords(TrieNode \*head);

// Heler Function: Calls printSource, printLexemeTable, and printList. Closes open FILE(s).

void wrapUp(TableEntry \*head, FILE \*src, FILE \*out);

// Prints the Source Program to stdout

void printSource(FILE \*src, FILE \*out);

// Prints the Lexeme Table to stdout

void printLexemeTable(TableEntry \*head, FILE \*out);

// Prints the (Raw) Lexeme List to stdout

void printLexemeList(TableEntry \*head, FILE \*out);

//----------------------------------//

// Parser Function Declarations //

//----------------------------------//

// TODO: Add Function Descriptions

// Invoques the parser

int parser(int printFlag);

int program(); //--------------------------------------------//

int block(); // //

int statement(); // Set of functions called recursively to //

int condition(); // check the grammar, build the symbol table, //

int expression(); // and generate the instructions for the VM //

int term(); // //

int factor(); //--------------------------------------------//

// Adds a new symbold to the Symbol Table

int addToTable(Symbol simbol);

// Looks for a symbol in the Symbol Table

Symbol \*lookUp(char \*symbol);

// update the address of a procedure

void updateAddress(char \*name, int currPC);

// mark as not accessible

void markUnusable(char\* name);

// Generates a new instruction

int gen(int OP, int REG, int L, int M);

// Prints the instruction register

void printIR();

//----------------------------------//

// VM Function Declarations //

//----------------------------------//

/\* VM - Function Definitions \*/

// Invoke the PM/0 (PL/0) Virtual Machine

int VM(char \*filename, int printFlag);

// Helper Function: Initilializes the PM/0 Virtual Machine.

// -Initializes STACK and REG to 0

// -Read Instructs from file if filename is not null

void initializeVM(char \*fileName);

// Parses and Executes the given INSTRUCTION

void instDecode(Instruction inst);

// Prints (dumps) the current state of the VM to stdout

void DumpVM();

// Helper Function: Provided by Professor MOtagne - Base Incrementer Func.

int base(int l, int base);

// Helper Function: Given by Professor MOtagne - Prints the current state of the stack

void printStack(int sp, int bp, int \*stack, int numAR);

// Helper Function: Returns the word (string literal) given an applicalbe OP-CODE (integer)

char \*parseOP(int i);

#define PL0

#endif