Homework 4: Code Generation for Full PL/0

See Webcourses and the syllabus for due dates.

1 Purpose

In this homework your team [Collaborate] will implement a code generator for the full PL/0 language, including nested procedures [UseConcepts] [Build]. The PL/0 language is defined in Section 6 below.

2 Directions

For this homework, we are providing several files in the `hw4-tests.zip` file in the course homeworks directory. These files include our stack VM (which is similar but not identical to that used in homework 1), which is placed in a subdirectory called `vm`, along with the VM's documentation. We are providing a compiler front end (including a lexer and parser) and a static analysis ("middle end") that produces enhanced ASTs as a kind of IR. We also provide a code module that defines a type `code_seq` that represents sequences of machine code instructions. Do not change any of these provided files (except for `gen_code.c`, which is explained below), as many are used in our testing.

Your task in this homework is to generate code for PL/0 programs that works correctly on the provided VM. We are also providing a main program in the file `compiler_main.c`, which interfaces with a `gen_code` module for code generation. We have provided the file `gen_code.h` and an outline of `gen_code.c` for you to edit. In `gen_code.c` most of the functions are stubs for which you will need to write implementations. When you see the body of a function that looks like the following

```
// Replace the following with your implementation
bail_with_error("... not implemented yet!");
return code_seq_empty();
```

that means that the function is a stub that you must write code for. You should not change the organization of the `gen_code.c` file, and should only provide implementations by replacing code like the above with code that works properly. We are also providing a `code` module (in the files `code.h` and `code.c`), which can be used to help with code generation (as described in class).

For the implementation, your code must be written in 2017 ANSI standard C and must compile with gcc and run correctly on Eustis, when compiled with the `-std=c17` flag.[1] We recommend using the gcc flags `-std=c17 -Wall` and fixing all warnings before turning in this assignment.

Note that we will randomly ask questions of students in the team to ensure that all team members understand their solution; there will be penalty of up to 10 points (deducted from all team members' scores for that assignment) if some team member does not understand some part of the solution to an assignment.

For this homework, you are *not* allowed to submit code generated by automatic compiler generator tools (such as LLVM).

In addition to your code, you must also submit the output of our provided tests named `hw4-vmtest*.pl0`, which the provided Makefile places in similarly named files with a `.myvo` extension.

Grading will be done as follows

[1] See this course's resources page for information on how to access Eustis.

1. (25 points) Generating code for constant and variable declarations.
2. (75 points) Generating code for expressions and atomic statements (those without any contained statements).
3. (50 points) Generating code for the remaining statements (begin, if-then-else, and while).
4. (25 points) Generating code for procedures and procedure calls (and returns).
5. (25 points) Generating code for procedures that includes nested procedures.

Note that the provided tests named `hw4-vmtest*.pl0` are syntactically legal and should not have parse errors (or declaration errors). They are arranged in sequence according to the rubric above, but it is often helpful to write your own tests to facilitate debugging.

3 What to Turn In

Your team must submit on Webcourses a single zip file containing your code generator. (That is, you only need to submit one zip file in total for this homework, not one for each part of the rubric above.) This zip file must include:

1. A plain text file, called `sources.txt`, that names all the `.c` source files used to implement your compiler. (The file names in `sources.txt` should be separated by either blanks or newlines.) This should include all the provided `.c` files. For example, your `sources.txt` file might look like the following:

   ```
   ast.c code.c compiler_main.c file_location.c gen_code.c id_attrs.c
   id_use.c instruction.c label.c lexer.c lexer_output.c lexical_address.c
   parser.c proc_holder.c reserved.c scope.c scope_check.c symtab.c token.c
   unparser.c utilities.c
   ```

   (The order of these names should not matter if you include the header files in each `.c` file that declare all the names used in that `.c` file.)

2. Each source file that is needed to compile both your compiler and VM with `gcc -std=c17` on Eustis, including all needed header files.
3. The output files that result from running our tests. These are the `.myvo` files created by the provided `Makefile`.

You can use the Unix command

```
make submission.zip
```

on Eustis to create a zip file that has all these files in it, after you have created your `sources.txt` file and run our tests (using the command `make check-outputs`) to create the `.myvo` files.

We will take points off for not passing the tests and some points off for: code that does not work properly, duplicated code, code with extra unnecessary cases, or code that is excessively hard to follow. Avoid duplicating code by using helping functions, or library functions. It is a good idea to check your code for these problems before submitting.

Don't hesitate to contact the staff if you are stuck at some point. Your code should compile properly; if it doesn't, then you probably should keep working on it. Email the staff with your code file if you need help getting it to compile or have trouble understanding error messages. If you don't have time to get your code to compile, at least tell us that you didn't get it to compile in your submission.

4 OS Interface

The following subsections specify the interface between the Unix operating system (as on Eustis) and the parser as a program.

4.1 Inputs

The compiler will be passed a single file name on the command line. This file name is the name of a file that contains the input PL/0 program to be compiled. Note that this input program file is not necessarily legal according to the semantics of PL/0[2]; for example it might do a division by 0. For example, if the file name argument is `hw4-vmtest1.pl0` (and both the compiler executable, `./compiler`, and the file `hw4-vmtest1.pl0` are in the current directory), then the following command line (given to the shell on Eustis)

```
./compiler hw4-vmtest1.pl0 > hw4-vmtest1.myvi
```

will run your compiler on the program in `hw4-vmtest1.pl0` and put the generated machine code into the file `hw4-vmtest1.myvi`.

The same thing can also be accomplished using the `make` command on Unix:

```
make hw4-vmtest1.myvi
```

4.2 Running the VM

The output of the compiler can be used as input to the provided VM. You can pass the file `hw4-vmtest1.myvi` to the VM, which is assumed to be named `vm/vm`, by running the Unix command, which both the VM's standard output and error output to `hw4-vmtest1.myvo`.

```
vm/vm hw4-vmtest1.myvi > hw4-vmtest1.myvo 2>&1
```

The same thing can also be accomplished using the `make` command on Unix:

```
make hw4-vmtest1.myvo
```

The output of the VM, which consists of a listing of the machine code program given to it and an execution trace, is placed in `hw4-vmtest1.myvo` by these commands.

(The `make` command can also make the `.myvo` file without you having to ask it to make the `.myvi` file first, as it will chain these commands together.)

4.3 Outputs

All of the compiler's error messages must be sent to standard error output (`stderr`).

4.4 Exit Code

When the compiler finishes without detecting any errors, it should exit with a zero error code; otherwise it should exit with a non-zero exit code.

[2]The compiler's front end and static analysis phases can also handle inputs that do not conform to the language, but our tests should not have such problems.

5 What Must be Done

For this assignment you need to: implement the stubs in `gen_code.c` and any other code needed to make code generation work correctly.

6 PL/0

The language you will be compiling for this homework is (full) PL/0, which includes procedures and the **`call`** statement. Note that in (full) PL/0 the "`procedure`" and "`call`" are reserved words.

6.1 Syntax

The context-free grammar for (full) PL/0 is defined in Figure 1 and its lexical grammar is defined in Figure 2.

Note that the context-free grammar of PL/0 is essentially the same as in homework 3, with the addition of (nested) procedure declarations and the **`call`** statement. The lexical grammar is unchanged from homework 2.

6.2 ASTs and Abstract Syntax

The type for abstract syntax trees (ASTs) is defined in the provided files `ast.h`, with helping functions in `ast.c`.

The file `ast.h` declares a type named `AST` and a type `AST_list`. The type `AST_list` is a (linked) list of ASTs.

The `AST` type is essentially the same as in homework 3, however, new ASTs have been added for procedure declarations and the call statement and the AST for programs (blocks) has a new field that holds the list of procedure declaration ASTs. (Note that the AST type for programs is reused as the AST type for blocks.)

The other major change to the ASTs is that uses of identifiers (in assignment statements, call statements, read statements, and identifier expressions) are replaced by a new type of AST node (with the type tag `ident_ast`) that contains the data subfield `ident` (of type `ident_t`). Where previously the AST in such places contained just (a pointer to) the string giving the identifier's name, the new AST that replaces it contains both that string (name) and a (pointer to an) `id_use` struct of type `id_use`. The type `id_use` is defined in the `id_use` module (files `id_use.h` and `id_use.c`). An `id_use` struct contains the `id_attrs` for that identifier (as found during static analysis) and also the number of scopes outward from the current scope that one needs to traverse to reach the scope where the name was declared. This is sufficient to construct the lexical address of the identifier used.

The abstract syntax for PL/0, which may be useful in working with the ASTs is given in Figure 3.

A good example of how to write a tree walk over the ASTs is given in the provided file `unparser.c`.

⟨program⟩ ::= ⟨block⟩ **.**

⟨block⟩ ::= ⟨const-decls⟩ ⟨var-decls⟩ ⟨proc-decls⟩ ⟨stmt⟩

⟨const-decls⟩ ::= **{**⟨const-decl⟩**}**
⟨const-decl⟩ ::= **const** ⟨const-def⟩ **{**⟨comma-const-def⟩**}** **;**
⟨const-def⟩ ::= ⟨ident⟩ = ⟨number⟩
⟨comma-const-def⟩ ::= **,** ⟨const-def⟩

⟨var-decls⟩ ::= **{**⟨var-decl⟩**}**
⟨var-decl⟩ ::= **var** ⟨idents⟩ **;**
⟨idents⟩ ::= ⟨ident⟩ **{**⟨comma-ident⟩**}**
⟨comma-ident⟩ ::= **,** ⟨ident⟩

⟨proc-decls⟩ ::= **{**⟨proc-decl⟩**}**
⟨proc-decl⟩ ::= **procedure** ⟨ident⟩ **;** ⟨block⟩ **;**

⟨stmt⟩ ::= ⟨ident⟩ **:=** ⟨expr⟩
| **call** ⟨ident⟩
| **begin** ⟨stmt⟩ **{**⟨semi-stmt⟩**}** **end**
| **if** ⟨condition⟩ **then** ⟨stmt⟩ **else** ⟨stmt⟩
| **while** ⟨condition⟩ **do** ⟨stmt⟩
| **read** ⟨ident⟩
| **write** ⟨expr⟩
| **skip**
⟨semi-stmt⟩ ::= **;** ⟨stmt⟩
⟨empty⟩ ::=

⟨condition⟩ ::= **odd** ⟨expr⟩
| ⟨expr⟩ ⟨rel-op⟩ ⟨expr⟩
⟨rel-op⟩ ::= = | <> | < | <= | > | >=

⟨expr⟩ ::= ⟨term⟩ **{**⟨add-sub-term⟩**}**
⟨add-sub-term⟩ ::= ⟨add-sub⟩ ⟨term⟩
⟨add-sub⟩ ::= ⟨plus⟩ | ⟨minus⟩
⟨term⟩ ::= ⟨factor⟩ **{**⟨mult-div-factor⟩**}**
⟨mult-div-factor⟩ ::= ⟨mult-div⟩ ⟨factor⟩
⟨mult-div⟩ ::= ⟨mult⟩ | ⟨div⟩
⟨factor⟩ ::= ⟨ident⟩ | ⟨sign⟩ ⟨number⟩ | (⟨expr⟩)
⟨sign⟩ ::= ⟨plus⟩ | ⟨minus⟩ | ⟨empty⟩

Figure 1: Context-free grammar for the concrete syntax of (full) PL/0. The grammar uses a `terminal font` for terminal symbols, and a **`bold terminal font`** for reserved words. As in EBNF, curly brackets $\{x\}$ means an arbitrary number of (i.e., 0 or more) repetitions of x. Note that curly braces are not terminal symbols in this grammar. Also note that an **`else`** clause is required in each if-statement.

⟨ident⟩ ::= ⟨letter⟩ **{**⟨letter-or-digit⟩**}**
⟨letter⟩ ::= `a` | `b` | `...` | `y` | `z` | `A` | `B` | `...` | `Y` | `Z`
⟨number⟩ ::= ⟨digit⟩ **{**⟨digit⟩**}**
⟨digit⟩ ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
⟨letter-or-digit⟩ ::= ⟨letter⟩ | ⟨digit⟩
⟨plus⟩ ::= `+`
⟨minus⟩ ::= `-`
⟨mult⟩ ::= `*`
⟨div⟩ ::= `/`

⟨ignored⟩ ::= ⟨blank⟩ | ⟨tab⟩ | ⟨vt⟩ | ⟨formfeed⟩ | ⟨eol⟩ | ⟨comment⟩
⟨blank⟩ ::= "A space character (ASCII 32)"
⟨tab⟩ ::= "A horizontal tab character (ASCII 9)"
⟨vt⟩ ::= "A vertical tab character (ASCII 11)"
⟨formfeed⟩ ::= "A formfeed character (ASCII 12)"
⟨newline⟩ ::= "A newline character (ASCII 10)"
⟨cr⟩ ::= "A carriage return character (ASCII 13)"
⟨eol⟩ ::= ⟨newline⟩ | ⟨cr⟩ ⟨newline⟩
⟨comment⟩ ::= ⟨pound-sign⟩ **{**⟨non-nl⟩**}** ⟨newline⟩
⟨pound-sign⟩ ::= #
⟨non-nl⟩ ::= "Any character except a newline"

Figure 2: Lexical grammar of PL/0. The grammar uses a `terminal font` for terminal symbols. Note that all ASCII letters (a-z and A-Z) are included in the production for ⟨letter⟩. Again, curly brackets **{**x**}** means an arbitrary number of (i.e., 0 or more) repetitions of x. Note that curly braces are not terminal symbols in this grammar. Some character classes are described in English, these are described in a Roman font between double quotation marks (" and "). Note that all characters matched by the nonterminal ⟨ignored⟩ are ignored by the lexer (except that each instance of ⟨eol⟩ ends a line).

⟨program⟩ ::= **{**⟨const-decl⟩**}** **{**⟨var-decl⟩**}** **{**⟨proc-decl⟩**}** ⟨stmt⟩
⟨const-decl⟩ ::= **`const`** ⟨name⟩ = ⟨number⟩
⟨var-decl⟩ ::= **`var`** ⟨name⟩
⟨proc-decl⟩ ::= **`procedure`** ⟨name⟩ ⟨program⟩
⟨stmt⟩ ::= ⟨ident⟩ **`:=`** ⟨expr⟩
| **`call`** ⟨ident⟩
| **`begin`** ⟨stmt⟩ **{**⟨stmt⟩**}**
| **`if`** ⟨condition⟩ **`then`** ⟨stmt⟩ **`else`** ⟨stmt⟩
| **`while`** ⟨condition⟩ **`do`** ⟨stmt⟩
| **`read`** ⟨ident⟩
| **`write`** ⟨expr⟩
| **`skip`**
⟨ident⟩ ::= ⟨name⟩ ⟨id-use⟩
⟨id-use⟩ ::= ⟨attrs⟩ ⟨levelsOutward⟩
⟨attrs⟩ ::= ⟨file-location⟩ ⟨kind⟩ ⟨loc-offset⟩ ⟨label⟩
⟨kind⟩ ::= **`constant`** | **`variable`** | **`procedure`**
⟨label⟩ ::= ⟨unset-label⟩ | ⟨set-label⟩
⟨unset-label⟩ ::= **`unset-label`** *`false`*
⟨set-label⟩ ::= **`set-label`** ⟨address⟩
⟨condition⟩ ::= **`odd`** ⟨expr⟩ | ⟨expr⟩ ⟨rel-op⟩ ⟨expr⟩
⟨rel-op⟩ ::= `=` | `<>` | `<` | `<=` | `>` | `>=`
⟨expr⟩ ::= ⟨expr⟩ ⟨bin-arith-op⟩ ⟨expr⟩ | ⟨ident⟩ | ⟨number⟩
⟨bin-arith-op⟩ **`::=`** `+` | `-` | `*` | `/`

Figure 3: Enhanced Abstract Syntax for PL/0 that serves as a description of the enhanced ASTs (or IR). Note that the abstract syntax of a ⟨program⟩ is also used as the abstract syntax of blocks (from the concrete syntax). The nonterminal ⟨name⟩ is a string that represents a identifier that is being declared. However, uses of identifiers are represented by ⟨ident⟩ in the abstract syntax, and contain more information. The nonterminal ⟨levelsOutward⟩ is an unsigned integer. The nonterminal ⟨file-location⟩ consists of a file name, and two unsigned integers (the line and column numbers). The nonterminal ⟨kind⟩ represents the type `id_kind` defined in the `id_attrs.h` file. Labels are data structures that are either unset or set and contain an address; the nonterminal ⟨address⟩ is an unsigned short integer representing an index in the VM's code array. Here the curly brackets, as in **{**x**}**, means a possibly empty list of x.

6.3 Semantics

This subsection describes the semantics of PL/0.

Nonterminals discussed in this subsection refer to the nonterminals in the context-free grammar of PL/0's concrete syntax, as defined in Figure 1.

A ⟨program⟩ consists of zero-or-more constant declarations (⟨const-decls⟩), zero-or-more variable declarations (⟨var-decls⟩), and zero-or-more procedure declarations (⟨proc-decls⟩), followed by a statement.

In PL/0, all constants and variables denote (short) integers. The execution of a program declares the named constants, variables, and procedures, and initializes the constants and variables. Then it runs the statement.

A *scope* in PL/0 is an area of program text that extends from the beginning of a ⟨program⟩ or ⟨block⟩ to its end. In PL/0 there are nested scopes, as a program or block contains blocks for procedures declared within it; any declarations of ⟨ident⟩s in a nested block that are also declared in a surrounding block cause a hole in that identifier's scope that is as big as the scope in which the inner declaration appears. However, it is an error if an ⟨ident⟩ is declared more than once in the same scope, as either a constant, a variable, or a procedure.

6.3.1 Constant Declarations

The nonterminal ⟨const-decls⟩ specifies zero or more constant declarations.

Each constant declaration, of the form ⟨ident⟩ = ⟨number⟩, declares that ⟨ident⟩ is a (short) integer constant that is initialized to the value given by ⟨number⟩. The scope of such a constant declaration is the area of the program's text that follows the declaration.

It is an error for an ⟨ident⟩ to be declared as a constant more than once. It is an error for the program to use the a declared constant's ⟨ident⟩ on the left hand side of an assignment statement or in a read statement.

6.3.2 Variable Declarations

The nonterminal ⟨var-decls⟩ specifies zero or more variable declarations.

Each variable declaration, of the form ⟨ident⟩, declares that ⟨ident⟩ is a (short) integer variable that is initialized to the value 0.

It is an error for an ⟨ident⟩ to be declared as a variable if it has already been declared as a constant or as a variable in the same scope.

Unlike constants, variable names may appear on the left hand side of an assignment statement or in a read statement.

6.3.3 Procedure Declarations

The nonterminal ⟨proc-decls⟩ specifies zero or more procedure declarations.

Each procedure declaration, of the form **`procedure`** ⟨ident⟩ **`;`** ⟨block⟩ **`;`** declares that ⟨ident⟩ is a procedure that when run executes the given ⟨block⟩; that is, it declares and initializes the constants and variables declared in the ⟨block⟩ and declares the block's procedures and then runs the statement in the ⟨block⟩. Therefore, a procedure executes as if it were a program, although it may use identifiers declared in a surrounding scope.

It is an error for an ⟨ident⟩ to be declared as a procedure if it has already been declared as a constant, variable, or procedure in the same scope.

Procedure names may not be used on the left hand side of an assignment statement nor may they be used in a read statement.

6.3.4 Statements

A ⟨block⟩ contains a single statement (⟨stmt⟩) that is run when the block is executed. When a program is run, its block is executed, and thus the statement in that block starts executing.

Assignment Statement An assignment statement has the form ⟨ident⟩ `:=` ⟨expr⟩. It evaluates the expression ⟨expr⟩ to obtain a value and then it assigns it to the variable named by ⟨ident⟩. Thus, immediately after the execution of this statement, the value of the variable ⟨ident⟩ is the value of ⟨expr⟩.

It is an error if the left hand side ⟨ident⟩ has not been declared as a variable. (Note that the evaluation of the ⟨expr⟩ may produce runtime errors.)

Call Statement A call of the form **`call`** ⟨ident⟩ executes the ⟨block⟩ declared by the procedure named ⟨ident⟩. (Therefore, it allocates space for the constants and variables declared in that procedure's ⟨block⟩, initializes them, and then executes that ⟨block⟩'s statement.)

It is an error if the ⟨ident⟩ has not been declared as a procedure.

Since procedures in PL/0 do not have formal parameters and do not return results, one can only pass arguments to a procedure and return results using variables that are global to that procedure.

Begin Statement A begin statement has the form **`begin`** $S_1; S_2; \ldots; S_n$ **`end`** (where $n \geq 1$) and is executed by first executing statement S_1, then if S_1 finishes without encountering an error S_2 is executed, and so on, in sequence. Any run-time errors encountered cause the entire compound statement's execution to terminate with that error.

Conditional Statement A conditional statement has the form **`if`** C **`then`** S_1 **`else`** S_2 and is executed by first evaluating the condition C. When C evaluates to true, then S_1 is executed; otherwise, if C evaluates to false (i.e., if it does not encounter an error), then S_2 is executed.

Note that in the concrete syntax there are no parentheses around the condition.

While Statement A while statement has the form **`while`** C **`do`** S and is executed by first evaluating the condition C. If C evaluates to false, then S is not executed and the while statement finishes its execution. When C evaluates to true, then S is executed, followed by the execution of **`while`** C **`do`** S again. Note that C is evaluated each time, not just once.

Again, in the concrete syntax there are no parentheses around the condition.

Read Statement A read statement of the form **`read`** x, where x is a declared variable identifier, reads a single character from standard input and puts its ASCII value into the variable x. The value of x will be set to -1 if an end-of-file or an error is encountered on standard input.

It is an error if x has not been previously declared as a variable.

Write Statement A write statement of the form **`write`** e, first evaluates the expression e, and if that expression yields a value in the range 0 to 255, then it writes that value to standard output as an ASCII character. Otherwise, if e yields a value outside the range 0 to 255 (i.e., a value less than 0 or greater than 255), then an error occurs.

Skip Statement A skip statement of the form **`skip`** does nothing and does not change the program's state.

6.3.5 Conditions

A ⟨condition⟩ is an expression that has a Boolean value: either true or false.

Odd Condition A ⟨condition⟩ of the form **odd** e first evaluates the expression e. If the value of e is an odd integer (i.e., it is equal to 1 modulo 2), then the value of the condition is true. If the value of e is even, then the value of the condition is false.

Relational Conditions A ⟨condition⟩ of the form $e_1\ r\ e_2$ first evaluates e_1 and then e_2, obtaining integer values v_1 and v_2, respectively. (If either evaluation encounters an error, then the condition as a whole encounters that error.) Then it compares v_1 to v_2 according to the relational operator r, as follows:

- if r is =, then the condition's value is true when v_1 is equal to v_2, and false otherwise.
- if r is <>, then the condition's value is true when v_1 is not equal to v_2, and false when they are equal.
- if r is <, then the condition's value is true when v_1 is strictly less than v_2, and false otherwise.
- if r is <=, then the condition's value is true when v_1 is less than or equal to v_2, and false when $v_1 > v_2$.
- if r is >, then the condition's value is true when v_1 is strictly greater than v_2, and false otherwise.
- if r is >=, then the condition's value is true when v_1 is greater than or equal to v_2, and false when $v_1 < v_2$.

6.4 Expressions

An ⟨expr⟩ of the form $e_1\ o\ e_2$ first evaluates e_1 and then e_2, obtaining integer values v_1 and v_2, respectively. (If either evaluation encounters an error, then the expression as a whole encounters that error.) Then it combines v_1 and v_2 according to the operator o, as follows:

- An expression of the form e_1+e_2 (i.e., a binary operator expression where the operator o is +) yields the value of $v_1 + v_2$, according to the semantics of the type **short int** in C.
- An expression of the form e_1−e_2 yields the value of $v_1 - v_2$, according to the semantics of the type **short int** in C.
- An expression of the form $e_1 \star e_2$ yields the value of $v_1 \times v_2$, according to the semantics of the type **short int** in C.
- An expression of the form e_1 / e_2 yields the value of v_1/v_2, according to the semantics of the type **short int** in C. The expression encounters an error if v_2 is zero.

There are also a few other cases of expressions that do not involve binary operators. These have the following semantics:

- An identifier expression, of the form x, has as its value the value of the integer stored in the constant or variable named x whose declaration is found in the closest syntactically surrounding scope.

  It is an error if x has not been previously declared as a constant or variable.

- An expression of the form sn, where s is a ⟨sign⟩ and n is a ⟨number⟩ yields the value of the base 10 literal n if the sign s is + or ⟨empty⟩. However, if the sign s is −, then the value is the negated value of the base 10 literal n according to the semantics of the type **`short int`** in C.

  Note that there is no AST for negating a number, since the AST can hold the negation; thus the negated value is simply stored as a number AST.

- An expression of the form `(`e`)` yields the value of the expression e.

6.5 Simple Example of Inputs and Outputs

Consider the following input in the file `hw4-vmtest2.pl0`, (note that the suffix is lowercase 'P', lowercase 'L', and the numeral zero, i.e., '0') which is included in the `hw4-tests.zip` file in the course homeworks directory.

```
# $Id: hw4-vmtest2.pl0,v 1.1 2023/03/20 21:23:14 leavens Exp $
var x;
x := 5.
```

When this is compiled and the resulting output is used as input to the VM, this should produce the expected output found in the following file (`hw4-vmtest2.vmo`).

```
Addr  OP    M
0     INC   3
1     INC   1
2     PBP   0
3     LIT   5
4     STO   3
5     HLT   0
Tracing ...
PC: 0 BP: 0 SP: 0
stack:
==> addr: 0     INC   3
PC: 1 BP: 0 SP: 3
stack: S[0]: 0 S[1]: 0 S[2]: 0
==> addr: 1     INC   1
PC: 2 BP: 0 SP: 4
stack: S[0]: 0 S[1]: 0 S[2]: 0 S[3]: 0
==> addr: 2     PBP   0
PC: 3 BP: 0 SP: 5
stack: S[0]: 0 S[1]: 0 S[2]: 0 S[3]: 0 S[4]: 0
==> addr: 3     LIT   5
PC: 4 BP: 0 SP: 6
stack: S[0]: 0 S[1]: 0 S[2]: 0 S[3]: 0 S[4]: 0 S[5]: 5
==> addr: 4     STO   3
PC: 5 BP: 0 SP: 4
stack: S[0]: 0 S[1]: 0 S[2]: 0 S[3]: 5
==> addr: 5     HLT   0
PC: 6 BP: 0 SP: 4
stack: S[0]: 0 S[1]: 0 S[2]: 0 S[3]: 5
```

6.6 Checking Your Work

You can check your own compiler by running the tests using the following Unix shell command on Eustis, which uses the `Makefile` from the `hw4-tests.zip` file in the course homeworks directory.

```
make check-outputs
```

Running the above command will generate files with the suffixes `.myvi` and `.myvo`; for example the compiler's output from the test file `hw4-vmtest3.pl0` will be put in `hw4-vmtest3.myvi` and the output from running that through the VM will be put into `hw4-errtest3.myvo`.

A Hints

We will give more hints in the class's lecture and lab sections.

We have often found it convenient to write our own PL/0 programs to test specific aspects of the compiler. The idea is to try to find what programs cause a problem and to isolate the cause of the problem; by writing your own (simple) programs and tracing the compiler's execution and outputs for those programs, you can isolate problems to specific functions in your code generator (`gen_code.c`).

(As an aid to writing PL/0 programs, the `compiler_main.c` file is written to understand the options `-l` or `-u`; these can be used before the file name on the command line. The `-l` option causes the compiler to print a token listing from the file given, as in homework 2. The `-u` option causes the compiler to print an unparsed version of the file given, provided that there were no syntax errors, as in homework 3.)

Recursion is your friend again and is assumed in the structure of `gen_code.c`. Write code trusting that the functions called work properly and concentrate on understanding what each function is responsible for doing.

Note that we are providing (in the `hw4-tests.zip` file in the course homeworks directory) both a declaration of the relevant types involved in working with code (in the `code` module , found in files `code.h` and `code.c`). The `id_use` module provides the type `id_use` and the `id_attrs` module provides the type `id_attrs` and functions that work with those. The `code` module provides not only the `code` type, but also the `code_seq` type. Also very useful is the `ast` module, which has several functions to create and return (pointers to dynamically allocated) ASTs and lists of ASTs.

For a good example of how to do a tree walk on the ASTs (e.g., to build a symbol and check declarations and identifier uses), see the provided files `unparser.c` and `scope_check.c`.

To find a name in lots of source code, from the Unix command line (or from the MacOS terminal app) you can use the command `grep`, as in the following, which searches all of your files ending in `.c` for the string `kind`:

```
grep 'kind' *.c
```

IDEs and the Windows explorer provide similar commands to search files. You can also use `findstr` in Windows or `Select-String` in the Windows PowerShell.