Details on the PL/0 Code Generator
(`$Revision: 1.11 $`)

Gary T. Leavens
Leavens@ucf.edu

November 15, 2023

Abstract

This document gives some details about code generation for the PL/0 language.

1 Introduction

The fourth (and last) part of the project in COP 3402 is to build a code generator for part of PL/0. This document gives some details about how to do that and some hints.

2 What to Read

A good explanation of code generation is found in the book *Modern Compiler Implementation in Java* [1], in which we recommend reading chapters 6–12.

You might also want to read *Systems Software: Essential Concepts* [2] chapter 6.

3 Overview

The PL/0 language itself is described in the *PL/0 Manual*, which is available in the files section of Webcourses. The *PL/0 Manual* defines the grammar of the language and its semantics.

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

3.1 Inputs

The compiler will be passed a single file name on the command line, and the command line also include one of two options (which are described in Section 3.2).

The 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[1]; 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
```

[1]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.

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

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

```
make hw4-vmtest1.bof
```

3.2 Compiler Options

The compiler's main function (in the provided file `compiler_main.c`) understands two options, both of which produce normal output on the standard output stream (`stdout`), error output on standard error output stream (`stderr`), and do not create or affect the `.bof` file.

The `-l` option can be used to produce a list of tokens in the program file, which can be useful for debugging the lexer. The compiler stops after producing this list, without proceeding to parsing.

The `-u` option can be used to unparse the AST produced by the parser and stop after declaration checking (without generating code). This can be useful for understanding the AST of a program and for other kinds of debugging.

3.3 Running the VM

The output of the compiler in the binary object file can be used as input to the provided VM. The VM is located in the `vm` subdirectory of the provided files. This VM essentially the SRM from homework 1, with a few changes:

- Tracing is no longer the default, but by using the `-t` option on the command line one can start tracing the execution from the very beginning of a program's execution. The format of the tracing output has also been changed to include more of the runtime stack area.
- A PINT instruction was added to print an integer in decimal format, see the SRM manual in the `vm` subdirectory for details.

You can pass the binary object file that results from compilation, for example `hw4-vmtest1.bof`, to the VM, which is assumed to be named `vm/vm`, by running the following Unix command, with both the VM's standard output and error output sent to a file, in this case `hw4-vmtest1.myo`.

```
vm/vm hw4-vmtest1.bof > hw4-vmtest1.myo 2>&1
```

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

```
make hw4-vmtest1.myo
```

To see the VM's tracing output, you can either have the VM execute the instruction STRA or you can pass the `-t` option on the command line when running the VM, as in the following.

```
vm/vm -t hw4-vmtest1.bof > hw4-vmtest1.myto 2>&1
```

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

```
make hw4-vmtest1.myto
```

(That is, the Makefile uses the suffix `.myto` to produce tracing output into the `.myto` file.)

(The `make` command can also make the `.myo` or `.myto` file without you having to ask it to make the `.bof` file first, as make will automatically chain these commands together.)

3.4 Outputs

The normal output of the compiler, when no options are used, goes into the binary object file (with a `.bof` suffix). When the `-l` or `-u` options are used (see Section 3.2), the normal output goes to the standard output stream (`stdout`).

However, all of the compiler's error messages should go to the standard error output stream (`stderr`).

3.5 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.

3.6 A Simple Example

Consider the input in the file `hw4-gtest1.pl0`, (note that the suffix is lowercase 'P', lowercase 'L', and the numeral zero, i.e., '0') shown in Figure 1, which is included in the `hw4-tests.zip` file in the files section of Webcourses.

```
write 8.    # writes 8
```

Figure 1: The test file `hw4-gtest1.pl0`.

Compiling this `hw4-gtest1.pl0`, for example by using the command `make hw4-gtest1.bof`, produces a binary object file `hw4-gtest1.bof`. When run in the VM, for example by using the command `make hw4-gtest1.myo` this produces output consisting of the character 8, which matches the provided file `hw4-gtest1.out`, as shown in Figure 2.

```
8
```

Figure 2: Expected output (on stdout) from running the compiler on `hw4-gtest1.pl0`, and then running that binary object file on the VM.

3.7 Provided Driver and Tests

We provide a driver (which is in the provided file `compiler_main.c`) to run the tests.

Tests are found in the files named `hw4-*.pl0`. The expected output that results from running the binary object file the compiler produces for each test in the VM is found in a file named the same as the test input but with the suffix `.out`. For example, the expected output of the PL/0 file `hw4-gtest1.pl0` is in the file `hw4-gtest1.out`.

3.8 Checking Your Work

You can check your own compiler by running the tests using the Unix command on Eustis:

```
make check-outputs
```

Running the above command will generate files with the suffix `.myo`; for example your output from test `hw4-gtest3.pl0` will be put into `hw4-gtest3.myo`.

A Hints

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

A.1 Debugging Code Generation

It is often convenient to write our own PL/0 programs to test specific aspects of the code generator. The idea is to try to find what programs cause a problem and to isolate the cause of the problem; for example by checking the output of **`write`** statements in the PL/0 code, you can see what part of the `gen_code` implementation is likely to be the cause of the trouble. In some cases you made need to go into more detail (as described below), but writing your own (simple) test programs can greatly speed up debugging by helping you quickly refine your theory of what is going wrong.

Consult the *PL/0 Manual* (available in the files section on Webcourses) for the syntax of PL/0.

If the problem is in your compiler, then you can use such small test programs to trace your compiler's execution.

A.1.1 Debugging the Code Generated

The problem is likely to be in the code that your compiler is generating, but by looking at the generated code using small example programs, you can isolate problems to specific functions in your code generator (i.e., in `gen_code.c`).

If you suspect the problem is in the generated code, it is often helpful to see the assembly language form of the generated code. To see the assembly language form of the generated code you can either use the -p option of the VM (with a command like `vm/vm -p mytest.bof`) or you can use the provided disassembler, with a command like:

```
vm/disasm mytest.bof > mytest.asm 2>&1
```

The same thing can be accomplished more conveniently by using the command:

```
make mytest.asm
```

with the provided Makefile. (This command will automatically generate the `mytest.bof` file if needed.)

If you find that some code you thought you were generating is missing, check to make sure that your code generation functions are returning the missing code sequences. Problems can be caused by failing to assign the results of calling `code_seq_concat` and `code_seq_add_to_end`, because code sequences (type `code_seq`) are linked lists (and C passes arguments using call-by-value).

A.1.2 Tracing the VM's Execution of Generated Code

If you would like to see the details of how the VM is executing your code, then use the tracing option of the VM. You can do this by running your program using the command:

```
make mytest.myto
```

and then looking at how the VM is executing each instruction.

A.2 General Tips

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

An example that shows how to do much of the code generation is provided by the FLOAT language, which is available from the course's example code webpage.

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), is contained in the provided `unparser` and `scope_check` modules.

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.

A.3 Provided Files

Note that we are providing (in the `hw4-tests.zip` file in the files section on Webcourses several modules, including:

- The `code` module (see the provided files `code.h` and `code.c`), which has functions (such as `code_srl`) that can conveniently create each type of individual machine instruction (these will be returned as pointers to the type `code`) and functions that work on sequences of such code elements (with the type `code_seq`).
- The `id_use` module provides the type `id_use`, and (pointers to) `id_use` structures are put into the ASTs by declaration checking, so that they are available when generating code for an identifier usage. Note that an `id_use` struct provides access to the name's `id_attrs` using the provided function `id_use_get_attrs`.
- The `id_attrs` module provides the type `id_attrs` and functions that work with those attributes.
- The `ast` module defines the structure and information in the ASTs. The ASTs are essentially the same as in homework 3, but now have `id_use` pointers in all constant and variable identifier uses.

A.4 Gradual Development

When writing the code generator, it is useful to build the capabilities of the code generator gradually, so you can test as you make progress. To do that, start with simple examples such as nonterminals that have no productions that generate other nonterminals (like identifier expressions), and then use these to build up to more complex examples (by combining the generated code sequences for the simpler examples). In this way you can debug each part of the code generator as you proceed and use the recursive tree walk over the AST to combine the generated code sequences for simple examples into more complex code sequences for more complex examples.

The following might be a useful order to gradually build up to more complex examples (this is to some extent followed by the provided tests named `hw4-gtest*.pl0`).

- The **`skip`** statement (and basic program bookkeeping for creating a binary object file).
- The **`write`** statement and numeric literals (hint: implement and use the `literal_table` for numeric literals).
- The **`begin`** statement, so that a program can do more than one thing (such as two write statements).

- Constant declarations and identifier uses.
- Variable declarations.
- Assignment statements (which can use the kinds of expressions implemented already, numeric literals and identifiers).
- Binary expressions (such as `x + 1`), which can be used in both assignment and write statements.
- Conditions and if-statements.
- While loops.

As you gradually develop the code generator, it is often useful to use "stubs" in your coding, so that if an example turns out to call code generation for some nonterminal that is not yet implemented, you will know about that. A stub can be created for a function in C by having the body of the function call `bail_with_error` with an appropriate error message. For example, you might use the following stubs in `gen_code.c`:

```
// (Stub for:) Generate code for the procedure declarations
code_seq gen_code_proc_decls(proc_decls_t pds)
{
    bail_with_error("TODO: no implementation of gen_code_proc_decls yet!");
    return code_seq_empty();
}

// (Stub for:) Generate code for a procedure declaration
code_seq gen_code_proc_decl(proc_decl_t pd)
{
    bail_with_error("TODO: no implementation of gen_code_proc_decl yet!");
    return code_seq_empty();
}
```

References

[1] Andrew Appel and Jens Palsberg. *Modern Compiler Implementation in Java: Second Edition*. Cambridge, 2002.

[2] Euripides Montagne. *Systems Software: Essential Concepts*. Cognella Academic Publishing, 2021.