Details on the SPL Code Generator
(`$Revision: 1.16 $`)

Gary T. Leavens
Leavens@ucf.edu

November 13, 2024

Abstract

This document gives some details about code generation for the SPL language.

1 Introduction

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

In terms of modules you need to implement:

1. A `gen_code` module that generates BOF files for the VM from SPL programs (by walking over the AST). The functions `gen_code_initialize` and `gen_code_program` are called from the compiler's main (in `compiler_main.c`).

2. A `literal_table` module, which the `gen_code` module would use to find the values of numeric literals used in the program.

Both of these will be discussed in class.

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 SPL language itself is described in the *SPL Manual*, which is available in the files section of Webcourses. The *SPL 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 SPL program to be compiled. Note that this input program file is not necessarily legal according to the semantics of SPL[1]; for example it might do a division by 0. For example, if the file name argument is `hw4-vmtest1.spl` (and both the compiler executable, `./compiler`, and the file `hw4-vmtest1.spl` are in the current directory), then the following command line (given to the shell on Eustis)

```
./compiler hw4-vmtest1.spl
```

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

The same thing can also be accomplished using the `make` command with the provided `Makefile` 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 SSM from homework 1, with a few changes:

- Tracing the VM's execution is no longer the default for the VM. Thus the `.myo` files produced by running the compiled code in the VM do not contain tracing output (unless the compiled code uses the STRA instruction).

  However, by using the VM's `-t` option one can start tracing the execution from the very beginning of a program's execution. This can also be done by using the Unix make command to make the corresponding `.myt` file, which compiles the `.spl` source file and then runs the resulting `.bof` file in the VM with the `-t` option.

  The format of the tracing output has also been changed to include more of the runtime stack area (in this VM the tracing shows the non-zero words from the address in the stack pointer (SP) register to the original stack bottom address given in the BOF file).

[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.

- A PINT instruction was added to print an integer in decimal format, see the *SSM Manual* in the `vm` subdirectory for details. (This instruction is handled in all the relevant modules.)
- Similarly, a CPR new instruction was added; this instruction copies a value from one register to another and is also detailed in the *SSM Manual*.

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.myt
```

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

(Note that 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. The `make` command should automatically remake the `.bof` files when the compiler is changed, but you can always use the command `make cleanall` to start over, by removing the executables and the BOF files.)

3.4 Outputs

The normal output of the compiler, when no options are used, goes into the binary object file (with a `.bof` suffix); for example, if the source code is in the file `test.spl` then the output would go into the file `test.bof`.

However, when the `-l` or `-u` options are used (see Section 3.2), the normal output goes to the standard output stream (`stdout`) and no BOF file is created.

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.

The compiled code should also exit with a zero error code when it terminates normally; thus you should be sure that your compiler adds an EXIT instruction with the code 0 to the end of the compiled code.

3.6 A Simple Example

Consider the input in the file `hw4-gtest1.spl`, shown in Figure 1, which is included in the `hw4-tests.zip` file in the files section of Webcourses.

Compiling this `hw4-gtest1.spl`, 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.

```
begin
  print 8   % prints 8
end.
```

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

```
8
```

Figure 2: Expected output (on stdout) from running the compiler on `hw4-gtest1.spl`, 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 hw4-tests.zip file with file names of the form `hw4-*.spl`. The expected output that results from running the binary object file the compiler produces for each test in the VM (without tracing) is found in a file named the same as the test input but with the suffix `.out`. For example, the expected output of the SPL file `hw4-gtest1.spl` 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.spl` will be put into `hw4-gtest3.myo`. These `.myo` files will be compared against the expected outputs in the corresponding `.out` files. Note that these comparisons do not involve tracing.

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 (very small) SPL 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 **`print`** statements in the SPL 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 *SPL Manual* (available in the files section on Webcourses) for the syntax of SPL.

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 the relevant code generation functions are linking those missing code sequences into their result and that they are returning the code sequence expected. Such problems can be caused by failing to use code_seq_concat (or code_seq_add_to_end), as the C compiler does not seem to warn you about calling a function whose result is ignored.

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.myt
```

and then looking at how the VM is executing each instruction in the mytest.myt file.

A.2 General Tips

Recursion is your friend (again); you can use the structure of the code in the provided unparser or scope_check modules for inspiration and examples of how to write a recursive walk over the ASTs in 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.3 Provided Files

In addition to the compiler_main.c file, we are providing several modules. These are found in the hw4-tests.zip file in the hw4 folder of the Files section on Webcourses. The provided modules include:

- The code module (in the provided files code.h and code.c), which has functions (with names based on the VM instructions, such as code_add and code_sub) that can create each type of VM instruction (these will be returned as pointers to the type code).
- The code_seq module (in code_seq.h and code_seq.c), which defines the code_seq type for sequences of VM instructions and functions to query and manipulate such code sequences.
- The code_utils module (in code_utils.h and code_utils.c), which defines functions that return useful code sequences, such as sequences for saving and restoring registers, and for calculating the frame pointer needed to address a variable or constant in a surrounding scope.

- The scope_check module (in scope_check.h and scope_check.c), which does declaration checking (as in homework 3) and also decorates the AST with id_use pointers, to give the code generator access to information about identifier uses (including each identifier's attributes). (The scope_check module uses the provided symtab module, which itself uses the provided scope module.)
- The id_use module provides the type id_use. Pointers to id_use structures are put into the ASTs during declaration checking (i.e., by scope_check.c), so that they are available when generating code for an identifier use. Note that an id_use structure 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 lexical_address module, which provides functions for dealing with lexical_address data structures.
- The file_location module, which provides functions for dealing with file_location data structures.
- 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 associated with all constant and variable identifier uses.
- The unparser module, which shows how to walk over the ASTs and can be useful for debugging (or error messages).
- The regname module, which defines macros for the important named registers (GP, FP, SP, and RA).
- The utilities module, which provides functions to print error messages and debugging information.
- The file spl_lexer.l, and the spl_lexer, lexer_utilities, and lexer modules, the do lexical analysis.
- The file spl.y, and the parser module, which provide parsing for the compiler.
- The bof and instruction modules, which deal with binary object files and VM instructions.

We also provide several files generated by flex and bison as well as a file of character inputs for testing purposes (char-inputs.txt).

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*.spl).

- An empty block (where you can see how the basic program bookkeeping for creating a binary object file works).
- The **print** 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 print 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 print 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.