Assignment 1

Contents

1. Background

During this course, we will write a compiler for a subset of Scheme that produces assembly language code for the 64-bit x86_64 architecture. Actually, we will do this each week of the semester, including this week, but the subset of Scheme will grow larger as the semester progresses. The compiler will also become more sophisticated in other ways, primarily to generate more efficient code.

The assignment for this week is to implement a compiler for the subset of Scheme given in Section 2. The compiler should consist of two passes:

- a verifier
- a code generator

The verifier should accept any Scheme value as an argument, but complain (with a descriptive message) if the value does not represent a program in the current subset of Scheme. The code generator should accept a program known to be in the current subset of Scheme and generate equivalent x86_64 code for the program. The generated code should be printed to the current output port.

The verifier should be called `verify-scheme`, and the code generator should be called `generate-x86-64`. You should be able to run them together on a Scheme program with a simple driver like the following:

```
(define driver
  (lambda (program)
    (with-output-to-file "t.s"
      (lambda ()
        (generate-x86-64 (verify-scheme program))))))
```

which leaves the output in the file `t.s`.

The structure of the code to be produced is given in Section 3. Section 4 gives the C code for a trivial run-time system that calls the generated Scheme program and prints its result, and Section 5 describes how to use gcc to assemble the generated code, compile the run-time system, link the two together, and run the resulting program. Section 6 and 7 give coding hints and instructions for testing the compiler.

What we need to know of the X86_64 architecture and instruction set is described in the X86_64 Primer.

2. Initial Scheme Subset

Here's the initial subset of Scheme we'll be handling:

Program	$\longrightarrow$	`(begin` *Statement*$^+$`)`
Statement	$\longrightarrow$	`(set!` Var_1 *int64*`)`
	|	`(set!` Var_1 Var_2`)`
	|	`(set!` Var_1 `(`*Binop* Var_1 *int32*`))`
	|	`(set!` Var_1 `(`*Binop* Var_1 Var_2`))`
Var	$\longrightarrow$	`rax` | `rcx` | `rdx` | `rbx` | `rbp` | `rsi` | `rdi`
	|	`r8` | `r9` | `r10` | `r11` | `r12` | `r13` | `r14` | `r15`
Binop	$\longrightarrow$	`+` | `-` | `*`

NB: In the final two statement forms, the LHS variable of the `set!` and the first operand of the binary operator must be the same variable, e.g., `(set! rdx (+ rdx r11))` is okay, but `(set! rdx (+ r11 rdx))` is not.

Int32 and *int64* are 32- and 64-bit exact integers, i.e., $-2^{31} \leq int32 \leq 2^{31} - 1$ and $-2^{63} \leq int64 \leq 2^{63} - 1$.

The restricted syntax for these operators, and the restricted choice of variable names is driven by the register and instruction set of the x86_64 architecture. We will soon lift both restrictions.

3. Boilerplate Code

The generated code must be enclosed in a wrapper that makes the entry point visible to the C run-time system and returns to C when the generated code has been run. The boilerplate code is as follows, where *generated code* denotes the hole to be filled by the generated code.

```
.globl _scheme_entry
_scheme_entry:
        generated code
        ret
```

The generated code must set `rax` to the final computed value. For example, the program:

```
(begin
  (set! rax 8)
  (set! rcx 3)
  (set! rax (- rax rcx)))
```

might produce the following assembly code:

```
.globl _scheme_entry
_scheme_entry:
        movq $8, %rax
        movq $3, %rcx
        subq %rcx, %rax
        ret
```

which returns the value 5 by leaving 5 in the `rax` register.

4. Run-time system

The run-time system consists of a main routine and a printer; both are trivial at this point.

```
#include <stdlib.h>
#include <stdio.h>
```

```
#ifdef __APPLE__ /* MacOS */
#define SCHEME_ENTRY scheme_entry
#else
#define SCHEME_ENTRY _scheme_entry
#endif

extern long SCHEME_ENTRY(void);

void print(long x) {
  printf("%ld\n", x);
}

int main(int argc, char **argv) {

 /* no arguments at this point */
  if (argc != 1) {
    fprintf(stderr, "usage: %s\n", argv[0]);
    exit(1);
  }

  print(SCHEME_ENTRY());

  return 0;
}
```

The special treatment of `scheme_entry` is necessary since MacOS implicitly adds an underscore onto `scheme_entry`, and we don't want it to think the name starts with two underscores.

Put the code for the run-time system in the file `runtime.c`. You can choose a different name, but if you do, adjust the invocation of the C compiler (described below) to reflect the different name.

5. Assembly instructions

You must be running a 64-bit x86_64 (AMD or Intel) Linux or MacOS system in order to build and run an executable program containing generated assembly code. If you do not have one of your own available, use `hulk.cs.indiana.edu`.

The generated assembly code must be assembled (translated) to machine code so that the processor can run it. Similarly, the C run-time code must be compiled to machine code. The two must be linked together to form a single executable. These three steps can be performed with a single invocation of the GNU C compiler, which knows how to handle assembly files (identified by the `.s` filename extension) as well as C files. Assuming that the generated assembly code (tucked inside the boilerplate code described in Section 3 is in the file `t.s` and that the run-time system is in the file `runtime.c`, typing the following command in the shell (bash or tcsh, for example) will do the job.

```
hulk% cc -m64 -o t t.s runtime.c
```

where `hulk%` is the shell's prompt.

The `-m64` option specifies that we are using the x86_64 instruction set, which may or may not be the default. The `-o` *filename* option specifies the name of the product executable file, which is `t` in this case. The remaining arguments are the files to be assembled or compiled.

The resulting executable file `t` can be run simply by typing `./t` in the shell as follows.

```
hulk% ./t
```

If you have `"."` in your path, you can type just `t` instead of `./t`.

So, for the example given in Section 3, a build-and-run interaction with the shell might look like the following.

```
hulk% cc -m64 -o t t.s runtime.c
hulk% ./t
5
```

Windows users: The Microsoft C compiler and assembler cannot be used for this class. The assembly syntax is different, and we aren't set up to test or grade the code in any syntax other than the one accepted by the Gnu tools. We tried installing the Cygwin tools, including the gcc compiler, on a 64-bit version of Windows, but the compiler installed by Cygwin does not support 64-bit compilation. If you're running a 64-bit version of Windows, you should be able to set up a Linux virtual machine, e.g., using vmware, but the easiest solution is to use hulk.cs.indiana.edu.

6. Coding hints

So that you can use helpers with names like `Statement` to reflect the grammar structure, we suggest you set the parameter `case-sensitive` to true by adding the following to the front of your file:

```
(case-sensitive #t)
```

The coding of `generate-x86-64` will go more smoothly if you make use of Chez Scheme's `format` and/or `printf` procedures. You can probably profit now and in future assignments by defining some helpers (procedures or syntactic forms) to format operands, emit instructions, and emit labels.

Although these passes are short enough to be written with `if` and `cond`, your code will be more clean and robust if you use `match`, which is defined in the file match.ss and described briefly in Using match.

Your compiler will run faster and you may get better feedback if you use optimize-level 2, by adding the following to the front of your file:

```
(optimize-level 2)
```

This tells the system that it can assume that primitive names like `cons` aren't going to be redefined or assigned. The downside is that if you do redefine or assign the name of a primitive, including via `trace`, it won't have any effect on your code.

7. Testing

A small set of short test programs, for both valid and invalid programs, appears in the file tests.ss. You should make sure that your compiler passes work at least on this set of tests.