Assignment 2

Contents

1. Background

Our ultimate objective for the semester is a compiler for the subset of Scheme described by the grammar below. In the subset, constants are fixnums and booleans, and datums are fixnums, booleans, the empty list, pairs of datums and vectors of datums. Fixnums are integers in a limited range to be determined by our choice of representation. Primitives are the Scheme procedures `*`, `+`, `-`, `<`, `<=`, `=`, `>=`, `>`, `add1`, `sub1`, `zero?`, `boolean?`, `integer?`, `null?`, `pair?`, `procedure?`, `vector?`, `not`, `eq?`, `cons`, `car`, `cdr`, `set-car!`, `set-cdr!`, `make-vector`, `vector-length`, `vector-ref`, `vector-set!`, and `void`.

```
Expr  ⟶  constant
      |  (quote datum)
      |  var
      |  (set! var Expr)
      |  (if Expr Expr)
      |  (if Expr Expr Expr)
      |  (begin Expr Expr*)
      |  (lambda (var*) Expr Expr*)
      |  (let ([var Expr]*) Expr Expr*)
      |  (letrec ([var Expr]*) Expr Expr*)
      |  (primitive Expr*)
      |  (Expr Expr*)
```

In the first assignment, we wrote a compiler for a much smaller subset of Scheme that also happens, by design, to be a parenthesized version of a small assembly language. Aside from the source-program verifier, our Assignment 1 compiler consists of a code generator whose input is expressed in this parenthesized assembly language, so that the code generator is little more than an assembly-code formatter. This is exactly what a code generator should be. If we give more responsibility than this to the code generator, which is inherently tied to a particular machine, retargeting the compiler to a different machine will require us to rewrite more code than necessary. So, although our code generator will grow to handle a larger subset of x86_64 assembly code, it will retain the flavor of an assembly-code formatter. Since our input language will become less and less like assembly code as we generalize and expand it, we will need to add additional passes between the source-program verifier and code generator to reduce the language to parenthesized assembly code.

Our general approach for the semester will be to expand the subset of Scheme handled by the compiler, and sometimes to make the code generated by the compiler more efficient, via a process of successive refinement. Each week, we will add new passes to the compiler, augment existing passes, or both.

In this assignment, we begin this process by expanding the source language handled by our compilers a bit, with the goal of making it closer to the final Scheme subset described above. Another goal of the assignment is to expand the code generator so that it handles a larger subset of x86_64 assembly language. In a sense, this is a secondary goal because we would not bother to expand the code generator if it were not necessary to handle the larger language. On the other hand, the sooner we fill out the code generator, the sooner we can declare it complete and move onto higher-level tasks. So our expanded source language is designed with an eye toward feeding the code generator something close to the final version of parenthesized assembly language we'll need for it to handle in the end.

2. Scheme Subset 2

Here's a grammar for the augmented subset of Scheme we'll be handling this week.

Program	$\longrightarrow$	`(letrec ([`*label* `(lambda ()` *Tail*`)]*)` *Tail*`)`
Tail	$\longrightarrow$	`(`*Triv*`)`
	|	`(begin` *Effect** *Tail*`)`
Effect	$\longrightarrow$	`(set!` *Var* *Triv*`)`
	|	`(set!` *Var* `(`*Binop* *Triv* *Triv*`))`
Var	$\longrightarrow$	*reg* | *fvar*
Triv	$\longrightarrow$	*Var* | *int* | *label*

A register *reg* is a symbol naming one of the x86_64 registers `rcx`, `rdx`, `rbx`, `rbp`, `rsi`, `rdi`, |, `r8`, `r9`, `r10`, `r11`, `r12`, `r13`, `r14`, and `r15`.

A frame variable *fvar* is a symbol whose name is of the form `fv`*index*, where *index* is a nonempty sequence of digits with no unnecessary leading zeros, e.g., `fv0`, *fv3*, or *fv10*. The frame variables extend the original limited set of register variables with an effectively unbounded set of variables.

A label *label* is a symbol whose name is of the form *prefix*`$`*suffix*, where *suffix* is a nonempty sequence of digits with no unnecessary leading zeros, e.g., `f$0`, `minimal?$3`, `destroy-instance!$50`, or `$$$$17`. Each label must have a unique suffix but not necessarily a unique prefix, so `f$1` and `f$2` may be used in the same program, but `f$1` and `g$1` cannot. Each label in the program should be bound by exactly one of the `letrec` bindings.

An integer *int* is an exact integer.

A binary operator *binop* is one of `+`, `-`, `*`, `logand`, `logor`, and `sra`. Each of these except `sra` are ordinary Scheme primitives; `sra` stands for "shift right arithmetic," and a definition of it is given in helpers.ss.

The grammar expressions are still limited by the peculiar constraints of the x86_64 target architecture. In particular:

- In `(set!` *Var* `(`*op* $Triv_1$ $Triv_2$`))`, $Triv_1$ must be identical to *Var*.
- A label cannot serve as either operand of a binary operator.
- In `(set!` *Var* *Triv*`)`, if *Triv* is a label, *Var* must be a register. (This is because we need to use the `leaq` instruction, which requires the destination to be a register.)
- In `(set!` *Var* *Triv*`)`, if *Triv* is an integer n, either (a) $-2^{31} \leq n \leq 2^{31} - 1$ or (b) $-2^{63} \leq n \leq 2^{63} - 1$ and *Var* must be a register.
- For `(set!` *Var* `(*` *Triv* *Triv*`))`, *Var* must be a register.
- For `(set!` *Var* `(sra` $Triv_1$ $Triv_2$`))`, $Triv_2$ must be an exact integer k, $0 \leq k \leq 63$. Any other integer operand n of a binary operator must be an exact integer, $-2^{31} \leq n \leq 2^{31} - 1$.
- For `(`*Triv*`)`, *Triv* must not be an integer.

We choose not to encode these machine constraints in the grammar because we want to leave open the possibility that we may target other machines with different constraints in the future. In the long run, these constraints will affect only the code generator, which gets to assume the constraints, and an instruction selection pass, which enforces the constraints. For now, they also affect our verifier and, of course, the set of test programs we can write.

3. Semantics

The `lambda` expressions in our new subset are used to create procedures, but these are not quite the same as Scheme procedures. For one thing, the `lambda` expressions appear at the programs top level, like functions in a C program. For another, the procedures do not take explicit arguments (though they can receive values in registers and frame variables). Indeed, these procedures are merely labeled blocks of code, with the labels specified by the `letrec` form. This is exactly what we can express directly in assembly code, so we are not yet straying far from a parenthesized assembly language.

The calls in our new subset are limited as well, first because they have no argument expressions, and second because they appear only in tail position—so that all calls are tail calls. Tail calls are sometimes referred to as jumps with arguments. Our tail calls are jumps *without* arguments, i.e., simply jumps. Again, this is exactly what we can express directly in assembly code.

To handle frame variables, the run-time system and boilerplate code cooperate to set the `rbp` register to the base of a stack area, and to allow the Scheme code to return to the boilerplate, register `r15` (our "return address register") is set to the address of a `_scheme_exit` label to which each program must jump (via a tail call) to return back to the run-time system. (See Section 5.)

An interesting question is whether the language allows the expression of arbitrary computations, i.e,. whether it is Turing complete. In particular, can it express programs that loop indefinitely, then terminate based on some changing condition? Can it express, in some fashion, recursive procedures, despite the limitation that all calls are tail calls? We will address these questions in lecture.

4. Things to do

To handle the new source language, we need to update the verifier, write two new passes to reduce the source language to a parenthesized assembly language, and update the code generator. The two new passes are `expose-frame-var`, which converts the frame variables *fvar* into explicit memory operands, and `flatten-program`, which flattens the code into a straight sequence of labels and statements. These new and modifed passes are described in Sections 4.1-4.4.

4.1. `verify-scheme`

This pass must be modified to reflect the structure of the new subset of Scheme, as well as the machine constraints that restrict the syntax of the `set!` forms.

4.2. `expose-frame-var`

The job of `expose-frame-variable` is to convert occurrences of the the frame variables fv_0, fv_1, etc., into displacement mode operands (see the X86_64 Primer), with `rbp` as the base register and an offset based on the frame variable's *index*. Since our words are 64 bits, i.e., 8 bytes, the offset for fv_i should be $8i$, e.g., 0, 8, 16, etc., for fv_0, fv_1, fv_2, etc.

See Section 9 for a description of frames and frame variables and their place in the world.

The helpers.ss procedure `make-disp-opnd` should be used to construct displacement-mode operands. This procedure is a product of the `define-record` form for `disp-opnd`, along with the predicate `disp-opnd?` and the accessors `disp-opnd-reg`, and `disp-opnd-offset`.

Nothing else changes in this pass.

Example: The source program

```
(letrec ([f$1 (lambda ()
                (begin
                  (set! fv0 rax)
                  (set! rax (+ rax rax))
                  (set! rax (+ rax fv0))
                  (r15)))])
  (begin
    (set! rax 17)
    (f$1)))
```

is converted to

```
(letrec ([f$1 (lambda ()
                (begin
                  (set! #<disp rbp 0> rax)
                  (set! rax (+ rax rax))
                  (set! rax (+ rax #<disp rbp 0>))
                  (r15)))])
  (begin
    (set! rax 17)
    (f$1)))
```

where `#<disp` *reg offset*`>` is the printed syntax of a displacement-mode operand.

4.3. `flatten-program`

This pass flattens out the now slightly nested structure of our source language into one that more closely resembles assembly language, no `letrec`, no `begin` forms, calls turned into explicit jumps, and the names of procedures turned into label forms. It produces a single `code` form containing a sequence of labels, effect expressions, and jumps, with the code for the body of the `letrec` appearing first followed by the body of each `lambda` expression in turn, prefixed by its label.

Example: The `expose-frame-var` output

```
(letrec ([f$1 (lambda ()
                (begin
                  (set! #<disp rbp 0> rax)
                  (set! rax (+ rax rax))
                  (set! rax (+ rax #<disp rbp 0>))
                  (r15)))])
  (begin
    (set! rax 17)
    (f$1)))
```

is converted to the following.

```
(code
  (set! rax 17)
  (jump f$1)
  f$1
  (set! #<disp rbp 0> rax)
  (set! rax (+ rax rax))
  (set! rax (+ rax #<disp rbp 0>))
  (jump r15))
```

4.4. `generate-x86-64`

This pass must be modified to handle the `code` form in place of the `begin` form (which is trivial) and to add handling for labels, jumps, and the new operators `logand`, `logor`, and `sra`.

Example: The `flatten-program` output

```
(code
  (set! rax 17)
  (jump f$1)
  f$1
  (set! #<disp rbp 0> rax)
  (set! rax (+ rax rax))
  (set! rax (+ rax #<disp rbp 0>))
  (jump r15))
```

is converted to the following.

```
    movq $17, %rax
    jmp L1
L1:
    movq %rax, 0(%rbp)
    addq %rax, %rax
    addq 0(%rbp), %rax
    jmp *%r15
```

5. Boilerplate Code

The boilerplate has gotten more complicated, first because we realized there were some additional x86_64 registers we should have been preserving but weren't (`rbx` and `r12-r15`), second because `rbp` must be set to the base of the stack created by the run-time system (which it passes in the first C argument register, `rdi`), and third because the return-address must be placed in register `r15`.

```
    .globl _scheme_entry
_scheme_entry:
    pushq %rbx
    pushq %rbp
    pushq %r12
    pushq %r13
    pushq %r14
    pushq %r15
    movq %rdi, %rbp
    leaq _scheme_exit(%rip), %r15
    generated code
_scheme_exit:
    popq %r15
    popq %r14
    popq %r13
    popq %r12
    popq %rbp
    popq %rbx
    ret
```

6. Run-time system

The new run-time system is in the file runtime.c. It is more complicated than last week's run-time system because it takes care of creating storage for the stack and passing the address of the stack to the boilerplate code to be set up as the initial value of the `rbp` register. It also sets up a heap area and passes the base of the heap area as a second argument to the boilerplate code, which ignores it for now since we have no need for a heap yet.

7. Coding hints

Even if you did not use match for the first assignment, you should use it for this one. Check out Using match and the posted Assignment 1 solution a1.ss to help you get started.

We have posted a set of helpers in helpers.ss, including the `make-disp-opnd` procedure described above, register, frame-var, and label predicates, and our set of `emit` macros, which you'll probably find quite handy. The helpers are all documented at the top of the file.

8. Testing

A small set of invalid and valid tests for this assignment have been posted in tests2.ss. You should make sure that your compiler passes work at least on this set of tests.

The registers and a default set of frame variables (`fv0` through `fv100`) are predefined in helpers.ss to facilitate testing. The frame variables are set up as macros that manipulate entries in a vector representing the stack, based at the index stored in `rbp`. So, if `rbp` is set to zero, `fv0` accesses the first element of the stack, `fv1` the second, and so on.

9. More on frame variables

The 15 registers we have available are plenty for most procedures. After all, how many procedure actually have more than 15 variables in use at the same time? Go look at the last 50 procedures you wrote. Chances are, all of them could get by with 15 variables.

But what happens when a procedure does need more than 15 variables? What happens when a procedure calls a second procedure, and the second calls a third, and so on, and the entire chain together requires more than 15 variables? What happens if a procedure calls itself nontail-recursively to an indefinite depth? What happens if a procedure calls a library procedure and has no idea how many or which registers the library procedure uses?

The answer is that each procedure is given, when called, a bank of memory locations referred to as a *frame* and, to support recursion, these frames are stored on a *stack*. There are many ways this can work. For example, when called, a procedure might allocate all the space it might need by decrementing a stack pointer, then release the space when it returns by incrementing the stack pointer. Or it might allocate and release space on the stack as needed. In any case, the space occupied by a procedure at any given point in its execution is considered the procedure's frame at that point.

In our compiler, we will employ a *frame pointer* in place of a stack pointer, and the frame pointer will always point at the base of the procedure's frame. If the procedure makes a nontail call, it will make sure that any variable values it may need to use when the nontail call returns are stored in its frame, adjust the frame pointer to some point above the saved values, make the call, and finally adjust the frame pointer back to the base of its own frame when the call returns.

We'll have more to say about this process later. For now, the compiler does not need to support nontail calls, so we don't have to worry about adjusting the frame pointer. In essence, all we need is one single frame, because each procedure can reuse its caller's frame when called via a tail call.

We do, however, need to set aside one of our registers to use as a frame pointer, and for this we use the register intended by the machine's designer to be used for this purpose, the `rbp` register. We'll assume that the run-time system has provided space for the single frame we need, and that the boilerplate code has put the address of the base of the frame in the `bfp` register.