Assignment 3

Contents

1. Background

Up until now, we've had to use variable names like `rax`, `r12`, and `fv3` in our code. These are hardly mneumonic. We've also had to avoid asking questions, or do so using a form of dynamically constructed jump table, as in the recursive factorial example in tests2.ss. This assignment partially address the former problem with the addition of a new `locate` form and fully addresses the latter with the addition of `if` expressions.

2. Scheme Subset 3

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

Program	⟶	`(letrec ([`*label* `(lambda () `*Body*`)]*)` *Body*`)`
Body	⟶	`(locate ([`*uvar Loc*`]*)` *Tail*`)`
Tail	⟶	`(`*Triv*`)`
	|	`(if` *Pred Tail Tail*`)`
	|	`(begin` *Effect** *Tail*`)`
Pred	⟶	`(true)`
	|	`(false)`
	|	`(`*relop Triv Triv*`)`
	|	`(if` *Pred Pred Pred*`)`
	|	`(begin` *Effect** *Pred*`)`
Effect	⟶	`(nop)`
	|	`(set!` *Var Triv*`)`
	|	`(set!` *Var* `(`*binop Triv Triv*`))`
	|	`(if` *Pred Effect Effect*`)`
	|	`(begin` *Effect** *Effect*`)`
Loc	⟶	*reg* | *fvar*
Var	⟶	*uvar* | *Loc*
Triv	⟶	*Var* | *int* | *label*

A unique variable *uvar* is a Scheme variable with a suffix similar to that of a label, except that the suffix is marked by "`.`" rather than of "`$`." That is, a unique variable 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`, or `destroy-instance!.50`. Within the same *Body*, each uvar must have a unique suffix but not necessarily a unique prefix, so `f.1` and `f.2` may be used in the same *Body*, but `f.1` and `g.1` cannot. Each uvar in a *Body* should be bound by exactly one of the `locate` bindings.

Registers (*reg*), frame variables (*fvar*), labels (*label*), integers (*int*), and binary operators (*binop*) are unchanged from the preceding subset.

The grammar expressions are still limited by the constraints of the x86_64 target architecture, as described in Assignment 2, with the additional similar constraints on the new relational operators implied by the architecture.

3. Semantics

The `locate` expressions in our new subset look rather like `let` expressions, but rather than create new locations, as `let` does, they create an alias from the LHS variable names to the RHS locations. In other words, they say where each variable is stored. Each variable must be assigned a single location, so it must appear at most once as an LHS of a given `locate` form. On the other hand, multiple variables may be assigned to the same location, i.e., both `x` and `y` can be located in register `rbx`. Of course, the programmer should ensure that this happens only when the variables are not live at the same time or when the variables are known to hold the same value where they are live at the same time. (A variable is *live* when it may yet be referenced.)

The `if` expressions in our new subset work just like `if` expressions in Scheme except that the set of predicate expressions is somewhat limited. A predicate can be the nullary operator `(true)`, the nullary operator `(false)`, a binary relational operator, an `if` expression whose then and else parts are predicates, or a `begin` expression whose last subexpression is a predicate.

A predicate cannot be a variable reference or a procedure call. The reason for this is simple: we want our intermediate language to be independent of the original source language and thus independent of any particular representation of true and false values. For us to make sense of a variable reference or procedure call in predicate position, we'd have to commit to such a representation.

4. Things to do

To handle the new source language, we need to update the verifier; add a new pass to replace each uvar occurrence with the corresponding location; update `expose-frame-var` to handle `if` expressions, predicate expressions, and `begin` expressions in effect context; add a new pass, `expose-basic-blocks`, that rewrites the program to restore the property that jumps appear only in tail context, and modify `flatten-program` and the code generator to handle conditional jumps. These new and modifed passes are described in Sections 4.1-4.6.

4.1. `verify-scheme`

This pass must be modified to reflect the structure of the new Scheme subset.

4.2. `finalize-locations`

This pass replaces each occurrence of a *uvar* in the body of each `locate` form with the corresponding `Loc`. It also discards the `locate` form. A grammar for the output of this pass is shown below.

Program	$\longrightarrow$	`(letrec ([`*label* `(lambda () `*Tail*`)]*)` *Tail*`)`
Tail	$\longrightarrow$	`(`*Triv*`)`
	|	`(if` *Pred Tail Tail*`)`
	|	`(begin` *Effect*`* `*Tail*`)`
Pred	$\longrightarrow$	`(true)`
	|	`(false)`
	|	`(`*relop Triv Triv*`)`
	|	`(if` *Pred Pred Pred*`)`
	|	`(begin` *Effect*`* `*Pred*`)`
Effect	$\longrightarrow$	`(nop)`
	|	`(set!` *Loc Triv*`)`
	|	`(set!` *Loc* `(`*binop Triv Triv*`))`
	|	`(if` *Pred Effect Effect*`)`
	|	`(begin` *Effect*`* `*Effect*`)`
Loc	$\longrightarrow$	*reg* | *fvar*
Triv	$\longrightarrow$	*Loc* | *int* | *label*

Example: The source program

```
(letrec ([f$1 (lambda ()
                (locate ([x.1 r8] [y.2 r9])
                  (if (if (= x.1 1) (true) (> y.2 1000))
                      (begin (set! rax y.2) (r15))
                      (begin
                        (set! y.2 (* y.2 2))
                        (set! rax x.1)
                        (set! rax (logand rax 1))
                        (if (= rax 0) (set! y.2 (+ y.2 1)) (nop))
                        (set! x.1 (sra x.1 1))
                        (f$1)))))])
  (locate () (begin (set! r8 3) (set! r9 10) (f$1))))
```

is converted to the following.

```
(letrec ([f$1 (lambda ()
                (if (if (= r8 1) (true) (> r9 1000))
                    (begin (set! rax r9) (r15))
                    (begin
                      (set! r9 (* r9 2))
                      (set! rax r8)
                      (set! rax (logand rax 1))
                      (if (= rax 0) (set! r9 (+ r9 1)) (nop))
                      (set! r8 (sra r8 1))
                      (f$1))))])
  (begin (set! r8 3) (set! r9 10) (f$1)))
```

This example is contrived to contain, in addition to a nonempty `locate` form, `if` expressions in all three contexts where they are valid: *Tail*, *Pred*, and *Effect*. We will use it as a running example for the new and significantly updated passes.

4.3. `expose-frame-var`

Depending on how you wrote this pass, it may need to be modified to reflect the structure of the grammar for the output of `finalize-locations`. It need *not* handle `locate` forms, which were discarded by the

preceding pass.

If you coded this pass as a simple tree walk that is independent of the grammatical structure, you should not need to make any changes.

4.4. `expose-basic-blocks`

The goal of this pass is to reduce the language to essentially the same language we had as input to `flatten-program` last week, with the addition of a limited form of `if` expressions in tail context, where they amount to two-way conditional jumps. Along the way, it must introduce new labels to handle the conditional control flow and bind these labels in the top-level `letrec` to procedures that represent the code to be executed at the target of each jump.

The following grammar describes the intermediate language of programs output from this pass.

Program	$\longrightarrow$	`(letrec ([`*label* `(lambda () `*Tail*`)]*)` *Tail*`)`
Tail	$\longrightarrow$	`(`*Triv*`)`
	|	`(if (`*relop Triv Triv*`) (,label) (,label))`
	|	`(begin `*Effect*`* `*Tail*`)`
Effect	$\longrightarrow$	`(set! `*Loc Triv*`)`
	|	`(set! `*Loc* `(`*binop Triv Triv*`))`
Loc	$\longrightarrow$	*reg* | *disp-opnd*
Triv	$\longrightarrow$	*Loc* | *int* | *label*

Compare this grammar with the previous one, and you'll see the following.

- *Tail* expressions have changed subtly in that the then and else parts of an `if` expression are no longer arbitrary *Tail* expressions but rather calls (jumps) to labels.
- *Pred* expressions are gone. The only surviving remnant is the *relop* call in the test part of an `if` expression.
- *Effect* expressions are simplified back down to the two forms of assignment.

We call this pass `expose-basic-blocks` because the pass effectively converts the incoming code containing arbitrarily nested `if` and `begin` expressions into code that contains only *basic blocks*, which are sequences of code entered only at the top and exited only at the bottom. There are several interesting optimizations that can be done with basic blocks, and we will undertake one or two before long.

Here's what the code for the running example program should look like after `expose-basic-blocks`:

```
(letrec ([f$1 (lambda () (if (= r8 1) (c$8) (a$9)))]
         [c$8 (lambda () (c$6))]
         [a$9 (lambda () (if (> r9 1000) (c$6) (a$7)))]
         [c$6 (lambda () (begin (set! rax r9) (r15)))]
         [a$7 (lambda ()
                (begin
                  (set! r9 (* r9 2))
                  (set! rax r8)
                  (set! rax (logand rax 1))
                  (if (= rax 0) (c$3) (a$4))))]
         [c$3 (lambda () (begin (set! r9 (+ r9 1)) (j$5)))]
         [a$4 (lambda () (j$5))]
         [j$5 (lambda () (begin (set! r8 (sra r8 1)) (f$1)))])
  (begin (set! r8 3) (set! r9 10) (f$1)))
```

Study this carefully and prove to yourself that it (a) performs the same computation as the input program and (b) is in the form required by the output grammar.

So we know what programs look like on input to this pass, and we know what they should look like on exit from the pass. The challenge, as always, is getting from the previous grammar to the new one while preserving the semantics of the input program. We will discuss this in lecture, but the following suggestions should help get you started.

- Each of the helpers for processing *Tail*, *Pred*, and *Effect* expressions may create new label bindings, so each needs to return a list of bindings as well as the output expression. The code can return these two values using the `values` procedure and receive them using either the `match` "cata" syntax or `let-values`. The labels should be created with the helpers.ss procedure `unique-label`.
- The *Pred* helper will need to be passed "true" and "false" labels. If the *Pred* helper encounters a call to a relational operator, it should generate a two-way conditional jump to the labels. It can do something much easier if it encounters `(true)` or `(false)`.
- The *Effect* helper will need to be passed a list of the output expressions that follow it so that, if the effect expression is an `if` expression, it can package up the code that follows and label it for use as a "join" point for the then and else parts of the `if` expression.

4.5. `flatten-program`

This pass must be extended to handle two-way conditional (`if`) expressions as well as unconditional jumps in tail context, converting them into the equivalent of assembly-language single-label compare-and-branch instructions. It should also be modified to take into account the label of the next letrec binding, if any, so that it does not produce unnecessary jumps.

The new jumps should be in one of the following two forms.

`(if (`*relop Triv Triv*`) (`*label*`))`
`(if (not (`*relop Triv Triv*`)) (`*label*`))`

`flatten-program` should choose which form to use based on the targets of the jump. If one is the same label as the label of the next `letrec` binding to be processed, `flatten-program` should produce a conditional jump to the other, introducing the `not` wrapper on the relational operator call if necessary, i.e., when jumping to the "false" label. If neither is the same as the label of the next `letrec` binding to be processed, it should produce a conditional jump to one followed by an absolute jump to the other. For example, `(if (< rax 3) (l$1) (l$2))` should produce

```
(if (< rax 3) (jump l$1))
```

if `l$2` is the label of the next `letrec` binding, and

```
(if (not (< rax 3)) (jump l$2))
```

if `l$1` is the label of the next `letrec` binding. It should produce either

```
(if (< rax 3) (jump l$1))
(jump l$2)
```

or

```
(if (not (< rax 3)) (jump l$2))
(jump l$1)
```

if neither `l$1` nor `l$2` is the label of the next `letrec` binding.

The code for handling unconditional jumps should also compare its jump target with the label of the next `letrec` binding to be processed. If it is the same, it should suppress the jump to avoid code sequences such as the following.

```
(jump l$7)
l$7
```

The label should not be suppressed, since it may be the target of a different jump.

In some cases, will result in two consecutive labels appearing in the output, and we can still end up with code like the following:

```
(jump l$7)
l$6
l$7
```

This is okay—we'll soon be adding an optimization pass that should prevent this situation from arising.

Example: The `flatten-program` output for the running example is shown below.

```
(code
  (set! r8 3)
  (set! r9 10)
  f$1
  (if (not (= r8 1)) (jump a$9))
  c$8
  (jump c$6)
  a$9
  (if (not (> r9 1000)) (jump a$7))
  c$6
  (set! rax r9)
  (jump r15)
  a$7
  (set! r9 (* r9 2))
  (set! rax r8)
  (set! rax (logand rax 1))
  (if (not (= rax 0)) (jump a$4))
  c$3
  (set! r9 (+ r9 1))
  (jump j$5)
  a$4
  j$5
  (set! r8 (sra r8 1))
  (jump f$1))
```

4.6. `generate-x86-64`

This pass must be modified to handle the conditional jump forms now produced by `flatten-program`, using the `cmpq` and conditional jump instructions `je`, `jne`, `jl`, `jle`, `jg`, and `jge`.

Example: The `flatten-program` output for the running example is shown below, without the boilerplate.

```
    movq $3, %r8
    movq $10, %r9
L1:
    cmpq $1, %r8
```

```
    jne L9
L8:
    jmp L6
L9:
    cmpq $1000, %r9
    jle L7
L6:
    movq %r9, %rax
    jmp *%r15
L7:
    imulq $2, %r9
    movq %r8, %rax
    andq $1, %rax
    cmpq $0, %rax
    jne L4
L3:
    addq $1, %r9
    jmp L5
L4:
L5:
    sarq $1, %r8
    jmp L1
```

5. Boilerplate and Run-time Code

The boilerplate and run-time code does not change.

6. Coding hints

Again, we strongly encourage you to use `match`. Take a look at the partial Assignment 2 solution for some ideas how to use `match`'s "cata" and extended quasiquote to work to avoid some of the explicit mapping and appending you may have done in your Assignment 2 solution.

7. Testing

A small set of invalid and valid tests for this assignment have been posted in tests3.ss. You should make sure that your compiler passes work at least on this set of tests.

We strongly encourage you to use the posted driver to automate your testing process.