Assignment 9

Contents

1. Background

In this assignment, we start the language-dependent portion of our compiler, i.e., the portion of the compiler that converts Scheme programs to UIL programs. Our first task is a simple one: implement a Scheme subset that is much like UIL but features `let` in place of `locals` and `set!`. We convert programs in the new subset to UIL by writing two passes, one that records the set of let-bound variables in a `locals` form and another that replaces `let` expressions with `set!` expressions.

2. Scheme Subset 9

Here is a grammar for the subset of Scheme we'll be handling this week:

```
Program  ⟶  (letrec ([label (lambda (uvar*) Tail)]*) Tail)
Tail     ⟶  Triv
          |  (binop Value Value)
          |  (alloc Value)
          |  (Value Value*)
          |  (if Pred Tail Tail)
          |  (begin Effect* Tail)
          |  (let ([uvar Value]*) Tail)
Pred     ⟶  (true)
          |  (false)
          |  (relop Value Value)
          |  (if Pred Pred Pred)
          |  (begin Effect* Pred)
          |  (let ([uvar Value]*) Pred)
Effect   ⟶  (nop)
          |  (mset! Value Value Value)
          |  (Value Value*)
          |  (if Pred Effect Effect)
          |  (begin Effect* Effect)
          |  (let ([uvar Value]*) Effect)
Value    ⟶  Triv
          |  (binop Value Value)
          |  (alloc Value)
          |  (Value Value*)
          |  (if Pred Value Value)
          |  (begin Effect* Value)
          |  (let ([uvar Value]*) Value)
Triv     ⟶  uvar | int | label
```

Unique variables (*uvar*), labels (*label*), integers (*int*), binary operators (*binop*), and relational operators (*relop*) are the same as in UIL. The machine constraints on integer values are also carried over from UIL.

Within a *Program*, each *label* bound by the `letrec` expression must have a unique suffix, and each *uvar* bound by a `lambda` or `let` expression must have a unique suffix. (This is more restrictive than UIL, which allows each a *uvar* in one *Body* to have the same suffix as a *uvar* in another *Body*.)

3. Things to do

To handle the new source language, we need to add three new passes:

- `verify-scheme`,
- `uncover-locals`, and
- `remove-let`.

We will keep `verify-uil` in our compiler as a check on the language-dependent front-end, so the first four passes of our compiler will be `verify-scheme`, `uncover-locals`, `remove-let`, and `verify-uil`.

The new passes are described in sections 3.1 through 3.3.

3.1. `verify-scheme`

This pass is similar to `verify-uil` but must reflect the changes in the language, i.e., the addition of `let` and the removal of `locals` and `set!`.

3.2. `uncover-locals`

This pass scans through each `lambda` or `letrec` body to find all variables bound by `let` expressions within the body and records these variables in a `locals` form wrapped around the body.

The only change to the grammar is the introduction of the `locals` form.

Program	$\longrightarrow$	`(letrec ([`*label* `(lambda (`*uvar*`*)` *Body*`)]*)` *Body*`)`
Body	$\longrightarrow$	`(locals (`*uvar*`*)` *Tail*`)`

Since the code within the body does not change, each of the grammar helpers (e.g., `Tail`) can simply return the list of unique variables found on the left-hand sides of `let` expressions within the body.

3.3. `remove-let`

This pass replaces each `let` expression in the input program with `set!` expressions, i.e., performs a conversion like the following:

`(let ([`*x e*`] ...)` *body*`)` $\rightarrow$ `(begin (set!` *x e*`) ...` *new-body*`)`

where *new-body* is the result of recursively processing *body*.

Since the order in which the right-hand sides of a `let` are evaluated is unspecified, you might give some thought to how you might reorder the assignments produced by the transformation above to make the final assembly code shorter and/or more efficient. For example, by reordering the assignments, you might be able to produce a program in which fewer variables are live across a call and thus must be saved in the frame.

The output of this pass is in UIL, a grammar for which is given in Assignment 8.

4. Boilerplate and Run-time Code

The boilerplate and run-time code does not change.

5. Testing

A small set of invalid and valid tests for this assignment have been posted in tests9.ss. You should make sure that your compiler passes work at least on this set of tests.

6. Coding Hints

Before starting, study the output of the online compiler for several examples.

Use `make-begin` in `remove-let` to avoid nested `begin` expressions.