Assignment 4

Modified Mon Feb 2 21:36:16 EST 2009

Contents

1. Background

In this assignment, we turn the job of allocating regsiters for our local variables over to the compiler, while still allowing ourselves to assign and reference register and frame variables explicitly. The compiler will use a graph-coloring register allocator.

Graph-coloring register allocation is based on the problem of coloring directly connected nodes of a graph different colors, just as we might color adjacent geographic regions on a map different colors. With a graph-coloring register allocator, the nodes of the graph are variables and registers, the links between nodes represent conflicts between variables and registers, and the colors are specific registers, e.g., `rax` or `r12`. So the idea is that we are trying to assign different registers to conflicting variables, but pack those that do not conflict together where possible so that we can locate as many variables in registers as we can.

A conflict between two variables means we cannot assign both to the same register, and a conflict between a variable and a register means we cannot assign the variable to the register. The first step of the algorithm is to determine where conflicts exist. We do this via live analysis.

1.1. Live Analysis

Two variables or a variable and a register *conflict* if:

a. one is in use (live) at some point where the other is assigned

b. the assignment isn't a simple move from one to the other

To see why this is the case, imagine that we are considering assigning two variables x and y to the same register. If, over the span of instructions where x is in use, there is no assignment to y, then we know that putting y in the same register will not wipe out x. If y is assigned within the span of instructions where x is in use, however, the assignment has the potential to wipe out the value in x, unless it happens that the value being stored in y is the same value as is already stored in x, which we generally know to be the case only if the assignment is a straight assignment of x to y.

A variable or register is in use, or *live*, at any given point, if the variable's value might yet be needed by the program. In general, this is an undecidable property, so we conservatively assume that the variable is live if we cannot prove that it is not live. The traditional conservative approximation, which we use, is that a variable is live at a given point if any reference to the variable occurs along any flow of control from the given point before the variable is *killed* (overwritten) by an intervening assignment to the variable.

Consider the following example.

```
(begin
  (set! a r8)
  (set! b fv0)
  (set! c (+ a 2))
  (if (< c 0) (nop) (set! c (+ c b)))
  (set! rax (+ c 1))
  (r15 rax rbp))
```

At the point where `a` is assigned, neither `b` nor `c` is live. The values in those variables, if any, cannot be used, since they are overwritten by assignments before any references occur. At the point where `b` is assigned, `a` is live, since it is referenced after that point, but `c` is not. At the point where `c` is assigned, `b` is live but `a` is not.

Since `a` is live where `b` is assigned, and it is not a straight assignment of `b` to `a`, `a` and `b` conflict. Similarly, `b` and `c` conflict, because `c` is assigned where `b` is live. But `a` and `c` do not conflict, since neither is live where the other is assigned. Thus, it is possible to put `a` and `c` in the same register, but not `a` and `b` or `b` and `c`.

Because the liveness of a variable is determined by whether it may yet be referenced, live analysis is performed backward along the flow of control from the leaves of a computation. We'll perform live analysis on one procedure at a time, so the leaves in our case are tail calls, which now list the set of locatons considered live at the point of the call. As the analysis works backward from the tail calls, it adds variables and registers when it sees references to them and removes variables and registers when it sees assignments to them. When operating on an `if` expression, it computes the live sets separately for the consequent and alternative. Because we don't know which will be executed, variables live in either should be considered live on exit from the test expression, so the natural approach is to union the two sets together before proceding upward along the flow of control in the test part. (We can do something a bit more clever than this, as we'll discuss later.)

Assuming the example above is the entire body of a procedure, live analysis proceeds from bottom to top as follows.

1. The live set is initialized to {`rax`,`rbp`}, which is the set of variables listed as live in the tail call.

	instruction	live after instruction	live after RHS
	`(set! a r8)`		
	`(set! b fv0)`		
	`(set! c (+ a 2))`		
	`(if (< c 0)`		
	`(nop)`		
	`(set! c (+ c b)))`		
	`(set! rax (+ c 1))`		
→	`(r15 rax rbp))`	{`rax`,`rbp`}	
conflicts for `a`: {}, `b`: {}, `c`: {}			

2. To this is added `r15`, since it is referenced by the tail call.

	instruction	live after instruction	live after RHS
	`(set! a r8)`		
	`(set! b fv0)`		
	`(set! c (+ a 2))`		
	`(if (< c 0)`		
	`    (nop)`		
	`    (set! c (+ c b)))`		
	`(set! rax (+ c 1))`	{rax,rbp,r15}	
→	`(r15 rax rbp))`	{rax,rbp}	
conflicts for a: {}, b: {}, c: {}			

3. The assignment to `rax` kills `rax`, removing it from the set of variables and registers live after the assignment. No conflicts are added at this point because the LHS is a register and no variables appear in the live set.

	instruction	live after instruction	live after RHS
	`(set! a r8)`		
	`(set! b fv0)`		
	`(set! c (+ a 2))`		
	`(if (< c 0)`		
	`    (nop)`		
	`    (set! c (+ c b)))`		
→	`(set! rax (+ c 1))`	{rax,rbp,r15}	{rbp,r15}
	`(r15 rax rbp))`	{rax,rbp}	
conflicts for a: {}, b: {}, c: {}			

4. Since the right-hand side of the assignment references `c`, we add `c` to the set. This set is used both for the consequent and alternative of the `if`.

	instruction	live after instruction	live after RHS
	`(set! a r8)`		
	`(set! b fv0)`		
	`(set! c (+ a 2))`		
	`(if (< c 0)`		
	`    (nop)`	{rbp,r15,c}	
	`    (set! c (+ c b)))`	{rbp,r15,c}	
→	`(set! rax (+ c 1))`	{rax,rbp,r15}	{rbp,r15}
	`(r15 rax rbp))`	{rax,rbp}	
conflicts for a: {}, b: {}, c: {}			

5. The assignment in the alternative kills `c`. `r15` and `rbp` are live here, where `c` is assigned, so we record that `c` conflicts with `r15` and `rbp`.

	instruction	live after instruction	live after RHS
	`(set! a r8)`		
	`(set! b fv0)`		
	`(set! c (+ a 2))`		
	`(if (< c 0)`		
	`    (nop)`	{rbp,r15,c}	
→	`    (set! c (+ c b)))`	{rbp,r15,c}	{rbp,r15}
	`(set! rax (+ c 1))`	{rax,rbp,r15}	{rbp,r15}
	`(r15 rax rbp))`	{rax,rbp}	
conflicts for a: {}, b: {}, c: {r15,rbp}			

6. The right-hand-side add `c` back, along with `b`.

	instruction	live after instruction	live after RHS
	`(set! a r8)`		
	`(set! b fv0)`		
	`(set! c (+ a 2))`		
	`(if (< c 0)`		
	`    (nop)`	{rbp,r15,c}	
	before alternative:	{rbp,r15,c,b}	
→	`    (set! c (+ c b)))`	{rbp,r15,c}	{rbp,r15}
	`(set! rax (+ c 1))`	{rax,rbp,r15}	{rbp,r15}
	`(r15 rax rbp))`	{rax,rbp}	
conflicts for `a`: {}, `b`: {}, `c`: {r15,rbp}			

7. The `nop` doesn't remove or add anything, so the set before the consequent is the same as the set after.

	instruction	live after instruction	live after RHS
	`(set! a r8)`		
	`(set! b fv0)`		
	`(set! c (+ a 2))`		
	`(if (< c 0)`		
	before consequent:	{rbp,r15,c}	
→	`    (nop)`	{rbp,r15,c}	
	before alternative:	{rbp,r15,c,b}	
	`    (set! c (+ c b)))`	{rbp,r15,c}	{rbp,r15}
	`(set! rax (+ c 1))`	{rax,rbp,r15}	{rbp,r15}
	`(r15 rax rbp))`	{rax,rbp}	
conflicts for `a`: {}, `b`: {}, `c`: {r15,rbp}			

8. The union of the consequent and alternative sets is used for `<` call.

	instruction	live after instruction	live after RHS
	`(set! a r8)`		
	`(set! b fv0)`		
	`(set! c (+ a 2))`		
	`(if (< c 0)`	{rbp,r15,c,b}	
→	*before consequent:*	{rbp,r15,c}	
	`    (nop)`	{rbp,r15,c}	
→	*before alternative:*	{rbp,r15,c,b}	
	`    (set! c (+ c b)))`	{rbp,r15,c}	{rbp,r15}
	`(set! rax (+ c 1))`	{rax,rbp,r15}	{rbp,r15}
	`(r15 rax rbp))`	{rax,rbp}	
conflicts for `a`: {}, `b`: {}, `c`: {r15,rbp}			

9. The predicate adds `c`, but `c` is already there, so the set does not change.

	instruction	live after instruction	live after RHS
	`(set! a r8)`		
	`(set! b fv0)`		
	`(set! c (+ a 2))`	{rbp,r15,c,b}	
→	`(if (< c 0)`	{rbp,r15,c,b}	
	`    (nop)`	{rbp,r15,c}	
	`    (set! c (+ c b)))`	{rbp,r15,c}	{rbp,r15}
	`(set! rax (+ c 1))`	{rax,rbp,r15}	{rbp,r15}
	`(r15 rax rbp))`	{rax,rbp}	
conflicts for `a`: {}, `b`: {}, `c`: {r15,rbp}			

10. The assignment kills `c`. `r15`, `rbp`, and `b` are live here, where `c` is assigned. We already know that `c` conflicts with `r15` and `rbp`, but we now also know that `c` conflicts with `b` and `b` conflicts with `c`.

	instruction	live after instruction	live after RHS
	`(set! a r8)`		
	`(set! b fv0)`		
→	`(set! c (+ a 2))`	{rbp,r15,c,b}	{rbp,r15,b}
	`(if (< c 0)`	{rbp,r15,c,b}	
	`(nop)`	{rbp,r15,c}	
	`(set! c (+ c b)))`	{rbp,r15,c}	{rbp,r15}
	`(set! rax (+ c 1))`	{rax,rbp,r15}	{rbp,r15}
	`(r15 rax rbp))`	{rax,rbp}	
conflicts for `a`: {}, `b`: {`c`}, `c`: {`b`,`r15`,`rbp`}			

11. The right-hand side adds `a`.

	instruction	live after instruction	live after RHS
	`(set! a r8)`		
	`(set! b fv0)`	{rbp,r15,b,a}	
→	`(set! c (+ a 2))`	{rbp,r15,c,b}	{rbp,r15,b}
	`(if (< c 0)`	{rbp,r15,c,b}	
	`(nop)`	{rbp,r15,c}	
	`(set! c (+ c b)))`	{rbp,r15,c}	{rbp,r15}
	`(set! rax (+ c 1))`	{rax,rbp,r15}	{rbp,r15}
	`(r15 rax rbp))`	{rax,rbp}	
conflicts for `a`: {}, `b`: {`c`}, `c`: {`b`,`r15`,`rbp`}			

12. The assignment kills `b`. `r15`, `rbp`, and `a` are live here, where `b` is assigned, so we record that `b` conflicts with `r15`, `rbp`, and `a` and that `a` conflicts with `b`.

	instruction	live after instruction	live after RHS
	`(set! a r8)`		
→	`(set! b fv0)`	{rbp,r15,b,a}	{rbp,r15,a}
	`(set! c (+ a 2))`	{rbp,r15,c,b}	{rbp,r15,b}
	`(if (< c 0)`	{rbp,r15,c,b}	
	`(nop)`	{rbp,r15,c}	
	`(set! c (+ c b)))`	{rbp,r15,c}	{rbp,r15}
	`(set! rax (+ c 1))`	{rax,rbp,r15}	{rbp,r15}
	`(r15 rax rbp))`	{rax,rbp}	
conflicts for `a`: {`b`}, `b`: {`r15`,`rbp`,`a`,`c`}, `c`: {`b`,`r15`,`rbp`}			

13. The right-hand side adds nothing, since we ignore frame variables.

	instruction	live after instruction	live after RHS
	`(set! a r8)`	{rbp,r15,a}	
→	`(set! b fv0)`	{rbp,r15,b,a}	{rbp,r15,a}
	`(set! c (+ a 2))`	{rbp,r15,c,b}	{rbp,r15,b}
	`(if (< c 0)`	{rbp,r15,c,b}	
	`(nop)`	{rbp,r15,c}	
	`(set! c (+ c b)))`	{rbp,r15,c}	{rbp,r15}
	`(set! rax (+ c 1))`	{rax,rbp,r15}	{rbp,r15}
	`(r15 rax rbp))`	{rax,rbp}	
conflicts for `a`: {`b`}, `b`: {`r15`,`rbp`,`a`,`c`}, `c`: {`b`,`r15`,`rbp`}			

14. The assignment kills `a`. `r15` and `rbp` are live here, where `a` is assigned, so we record that `a` conflicts with `r15` and `rbp`.

	instruction	live after instruction	live after RHS
→	`(set! a r8)`	{rbp,r15,a}	{rbp,r15}
	`(set! b fv0)`	{rbp,r15,b,a}	{rbp,r15,a}
	`(set! c (+ a 2))`	{rbp,r15,c,b}	{rbp,r15,b}
	`(if (< c 0)`	{rbp,r15,c,b}	
	`(nop)`	{rbp,r15,c}	
	`(set! c (+ c b)))`	{rbp,r15,c}	{rbp,r15}
	`(set! rax (+ c 1))`	{rax,rbp,r15}	{rbp,r15}
	`(r15 rax rbp))`	{rax,rbp}	
conflicts for `a`: {r15,rbp,b}, `b`: {r15,rbp,a,c}, `c`: {b,r15,rbp}			

15. The right-hand side adds `r8`, so the live set on entry contains `rbp`, `r15`, `r8`. If it contained anything other than registers, this would indicate a bug in our source program.

	instruction	live after instruction	live after RHS
	on entry	{rbp,r15,r8}	
→	`(set! a r8)`	{rbp,r15,a}	{rbp,r15}
	`(set! b fv0)`	{rbp,r15,b,a}	{rbp,r15,a}
	`(set! c (+ a 2))`	{rbp,r15,c,b}	{rbp,r15,b}
	`(if (< c 0)`	{rbp,r15,c,b}	
	`(nop)`	{rbp,r15,c}	
	`(set! c (+ c b)))`	{rbp,r15,c}	{rbp,r15}
	`(set! rax (+ c 1))`	{rax,rbp,r15}	{rbp,r15}
	`(r15 rax rbp))`	{rax,rbp}	
conflicts for `a`: {r15,rbp,b}, `b`: {r15,rbp,a,c}, `c`: {b,r15,rbp}			

The live analysis is now complete, and we have our final conflict sets:

`a`: {r15,b}
`b`: {c,r15,a}
`c`: {r15,b}

There are some important things to note:

- A variable never appears in its own conflict set. In order for it to appear, it would have to be assigned where it is live, but as soon as it is assigned, it becomes dead. So by using the live set in the interval between the computation of the right-hand side of the assignment and the assignment itself, we avoid having to do anything special to remove self conflicts. For example, when handling the assignment `(set! c (+ c b))`, we first remove `c` from the running live set, then record the conflicts between `c` and any registers and variables left in the live set, then `add c` back, along with `b`.
- We don't maintain conflict sets for registers. Registers are fixed; we can't, for instance, decide to locate `r8` in `rax`. So the register allocator will never need to ask with which other registers and variables a register conflicts.
- In the example above, it happens that `r8` is not live at the point where `a` is assigned to `r8`. Even if `r8` were live, we would not add `r8` to `a`'s conflict set, since the assignment is a straight move of one to the other.
- The set of conflicts we record at an assignment depends only on the left-hand-side variable and the set of variables that are live just before the assignment occurs. The set of variables referenced on the right-hand side are *irrelevant* when determining conflicts. So when we saw `(set! c (+ c b))` we *did not* add a conflict between `b` and `c`, though we would have if `b` had been live at that point.

- It is possible for a variable to be assigned at some point where it is not live. If this occurs, we could discard the assignment and ignore any variable and register references on the right-hand side of the assignment. If we do this, we'll usually end up with fewer conflicts. We don't bother however, and instead assume that some earlier, possibly optional, optimization pass has discarded useless assignments. In general, we don't want to complicate our required passes by performing optimizations, and we want to give users of our compiler the choice of whether optimizations should be enabled.

While simply unioning the consequent and alternative live sets before processing the predicate of an `if` expression will work, we can gather more precise live information in the presence of the boolean constants `(true)` and `(false)` by maintaining separate true and false live sets in predicate context. For all `if` expressions, we pass our predicate handler two live sets, a true live set (from the consequent expression) and a false live set (from the alternative expression). The handler propagates the two sets through to the consequent and alternative of any nested `if` expression and the last subexpression of any nested `begin` expression. If it finds `(true)` or `(false)`, it returns the true live set or the false live set accordingly. In the only other case, that of a predicate primitive call, it must union the two sets before processing the call, since the boolean value of the primitive call is unknown.

This pays off when variable references occur in dead code because the entire test expression is a constant, i.e., in the following cases.

```
(if (true) e1 e2)
(if (false) e1 e2)
```

In the first case, only those variables live on entry to e_1 will be shown live on entry to the `if` expression, while in the second case, only those variables live on entry to e_2 will be shown live on entry to the `if` expression,

If this were the only benefit, we would not bother, since we assume that such expressions would have been simplified by some earlier optimization pass, if desired. It also, however, pays off in more subtle cases like the one below.

```
(if (if e1
        (begin e2 (false))
        e3)
    e4
    e5)
```

Since e_4 cannot be reached from code that passes through e_2, the set of variables that are live after e_2 includes only those that are live going into e_5. By returning only the false live set for the occurrence of `(false)` just after e_2 we ensure that these are the only variables to make it through.

1.2. Register Assignment

The register assignment algorithm takes two inputs: a list of variables and a conflict table. It returns a list of register assignments for all or some of the variables. It is most easily described recursively as follows.

- If the list of variables is empty, return an empty list of register assignments.
- Pick a low-degree variable (either spillable or unspillable) from the list of variables, if one exists. Otherwise pick any spillable variable. A low-degree variable is one that conflicts with with fewer than k variables or registers in the current conflict table.
- Recur with the picked variable removed from the list of variables and the picked variable removed from the conflict lists of the other variables in the conflict table. (Thus, we expect the recursive call assign registers to a list one shorter with a conflict graph that possibly has fewer conflicts.) The recursive call should return a list of register assignments for (at least some of) the remaining variables.

- Attempt to select a register for the picked variable, avoiding any registers the picked variable conflicts with and any registers to which a conflicting variable is assigned in the list of register assignments returned by the recursive call. This step will succeed if a low-degree variable is picked, and may or may not succeed otherwise. If it succeeds, add the assignment to the list of register assignments and return the updated list. Otherwise return the non-updated list.

This algorithm is an adaptation of the optimistic register allocation described in "Improvements to graph coloring register allocation" (ACM TOPLAS 6:3, 1994) by Preston Briggs, et al.

If the list of register assignments returned by the register allocator contains an assignment for each variable in the original set of variables, the register allocator has succeeded. Otherwise, the register allocator has failed, and the set of variables for which the list *does not* contain assignments must be spilled, i.e., assigned frame locations. We are not handling frame allocation at present, so the register allocator should simply error out with an appropriate message if this happens.

1.3. Live Set Representation

A live set is most easily represented in Scheme as a list of variables and registers with no duplicates. There are more efficient representations for sets, and we'd want to use one of them in a production compiler, but lists will do for our purposes. Procedures for dealing with sets are in the latest version of helpers.ss: `set-cons`, which adds a single element to a set, `union`, which unions two or more sets, `intersect`, which intersects two or more sets, and `difference`, which returns the difference between two sets.

1.4. Conflict Graph Representation

A conflict graph is most easily represented in Scheme as an association list mapping each variable to a list of the variables and registers with which it conflicts. For example,

```
((a r15 rbp b)
 (b r15 rbp a c)
 (c b r15 rbp))
```

represents the conflict sets derived for the example in the preceding section. The represention is redundant in that conflicts between two variables are listed twice, once for each variable, but the efficiency with which the conflicts of a particular variable can be determined more than makes up for the cost of maintaining the redundant information.

The set operations can be used along with `assq` and `set-cdr!` to manipulate conflict graphs represented in this manner.

Again, we would probably choose a more efficient (and complicated) representation in a production compiler, but this representation will do for our purposes.

2. Scheme Subset 4

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

Program	⟶	`(letrec ([`*label* `(lambda () `*Body*`)]*)` *Body*`)`
Body	⟶	`(locals (`*uvar*`*)` *Tail*`)`
Tail	⟶	`(`*Triv* *Loc*`*)`
	|	`(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*

The only changes are (1) the `locate` form has been replaced with a `locals` form that lists unique variables but no locations, and (2) a list of locations *Loc*`*` is now included in each tail call.

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

Each *uvar* in a *Body* should appear exactly once in the `locals` list.

The grammar expressions are still limited by the constraints of the x86_64 target architecture, as described in the previous assignments.

3. Semantics

The `locals` form is like the `locate` form in that it declares a set of unique variables. It doesn't specify a set of locations, however, since it's the compiler's responsibility to assign locations to the variables in the locals list.

The locations *Loc*`*` listed in each call specify the set of locations presumed to be needed at the target of the jump. For a call representing a return, this will usually include the return-value register, `rax`. For a call representing a tail call, it will include the locations of the arguments. In both cases, `rbp` should also be listed, unless the entire program makes no use of frame variables. The order of the locations in the list is unimportant.

4. Things to do

To handle the new source language, we need to update the verifier and add three new passes: one to perform live analysis and build a *conflict graph*, a second to assign registers, and a third to discard the *Loc*`*` list included in each call. The three new passes follow `verify-scheme` and precede `finalize-locations`. 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 Scheme subset.

4.2. `uncover-register-conflict`

This pass inserts into the output a conflict graph listing for each *uvar* in the locals list a list of the other unique variables and registers with which it conflicts, i.e., with which it cannot share a register. It makes no other changes to the program.

A grammar for the output of this pass is shown below.

```
Program  ⟶  (letrec ([label (lambda () Body)]*) Body)
Body     ⟶  (locals (uvar*)
               (register-rconflict conflict-graph Tail))
Tail     ⟶  (Triv Loc*)
         |   (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 *conflict-graph* should be an association list mapping each variable in the `locals` list to a list of conflicting variables and registers, as described in Section 1.4.

Use the online compiler to generate example cases.

4.3. `assign-registers`

This pass attempts to assign register to each *uvar* in the `locals` list, using the register assignment algorithm described in Section 1.2. The set of available registers is given by the `registers` list defined in helpers.ss.

If successful, it records the register assignments in an output `locate` form and removes the `locals` and `register-conflict` forms. It makes no other changes to the program.

The grammar for the output of this pass is the almost same as the grammar for the source language of Assignment 3. The only differences are the *Loc** list included in each call, which will be discarded by the following pass, and the replacement of arbitrary locations *Loc* on the right-hand sides of `locate` bindings with registers.

```
Program  ⟶  (letrec ([label (lambda () Body)]*) Body)
Body     ⟶  (locate ([uvar reg]*) Tail)
Tail     ⟶  (Triv Loc*)
          |  (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
```

Use the online compiler to generate example cases.

4.4. `discard-call-live`

This pass discards the *Loc** list included in each call. The grammar for the output of this pass is the essentially same as the grammar for the source language of Assignment 3. The only difference is that the `locate` form right-hand sides are registers rather than *Locs* (registers or frame variables). This minor difference should not precipitate any changes to the that come after this one.

```
Program  ⟶  (letrec ([label (lambda () Body)]*) Body)
Body     ⟶  (locate ([uvar reg]*) 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
```

5. Boilerplate and Run-time Code

The boilerplate and run-time code does not change.

6. Testing

A small set of invalid and valid tests for this assignment have been posted in tests4.ss. You should make sure that your compiler passes work at least on this set of tests.

7. Coding Hints

Here are some quick hints:

- If you add conflicts to the conflict table destructively, i.e., use `set-cdr!` after finding the proper association, the helpers of `uncover-register-conflict` need to return just one value, the set of live variables. They do not need to return an expression, since the body code doesn't change. (The only change is the addition of the `register-conflict` form.)
- There's no need for `assign-registers` to venture into the code of a body; it uses only the `locals` list and the `register-conflict` table, and it doesn't make any changes to the code other than wrapping it with the `locate` form and discarding the other wrappers.
- While `discard-call-live` does need to venture into *Tail* expressions, it does not need to venture into *Effect* and *Pred* expressions.