

Module 10: Closure Conversion

Introduction

This week you'll finish your UFT by translating pure, higher-order Scheme into first-order Scheme.

- *What am I doing?*
 - Implement `lambda` by adding *closure conversion* to your UFT. You'll implement a limited form that does not support mutation of variables captured in closures.
- *Why am I doing it?*
 - You'll learn how first-class, nested functions are implemented—and you may be pleasantly surprised how simple they can be. And you'll learn a realistic technique that scales perfectly well to industrial compilers.
 - You'll skip mutation because putting mutable data in closures rarely makes sense. Mutable data can be dealt with through a variety of techniques, but the easy one is to implement an additional pass that does a little static analysis, and moves any captured mutable variables to the heap. You can implement such a pass for depth points.
 - Closure conversion provides a classic case study in the most general way a new feature can be added to a virtual-machine system: you have new syntax (`lambda`), new VM support (a closure type and three supporting instructions), a new compiler pass (closure conversion), and small updates to existing compiler passes (K-normalization and code generation).
- *How?*
 - Before lab, you'll read about how closures work: each `lambda` is turned into a record allocation. And you'll see how closure records can be simulated in vScheme by embedding them as lists.
 - In lab you'll closure-convert a couple of `lambdas` by hand, and you'll implement the embedding you read about before lab.

- After lab you’ll build the closure-conversion pass and the necessary extensions throughout the UFT and SVM system. Nothing here is too challenging, but as is typical for a language extension, you’ll touch a lot of code.
- At the end of the week, you’ll submit a translator that includes pass `ho-cl` (for debugging) as well as `ho-kn`, `ho-vs`, and `ho-vo` (to run the code). Your system (translator plus VM) will be able to run any mutation-free Scheme code from COMP 105.

The module step by step

Before lab: New code, free variables, and closures

Since our preparation time is unusually short, I’ve tried to cut the lab prep to a minimum.

- (1) *If necessary, refresh your memory on free variables.* On Slack, I’ve posted three pages from a book chapter on free and bound variables. Before lab, have a look at the first three paragraphs and the examples of free variables.

If you’ve done the “Improving closures” problem in COMP 105, you might find that worth reviewing as well. It will help you understand the ideas, but there’s nothing specific there that you need to review before lab.

If you already understand and remember the ideas of free variables, you can skip this step.

- (2) *Read about closure conversion.* Read the handout How Closures Work. Before lab, the key sections are the introduction, how it works, and embedding.)

This the key step.

- (3) *Download updates.* Update your git repository in two steps:
 - A. Be sure all your own code is committed. Confirm using `git status`.
 - B. Update your working tree by running `git pull` (or you might possibly need `git pull origin master`). I apologize, but **you will get merge conflicts** in `uft.sml`. You can try resolving them in favor of the upstream repo, but don’t lose changes you made in previous modules. If in doubt, ask course staff for help.

Verify that your UFT builds with the new code.

If you don’t get to this step, you’ll be able to complete the first part of the lab but not the second part.

Lab: Closure-conversion highlights

This lab will give you the highlights of closure conversion, as well as all of the *ideas* you need to complete the module. A lot of details will follow, but if you have a productive lab, the details won't give you trouble.

The idea of closure conversion is to **turn lambda into a data structure**. (A data structure is something we already know how to implement.) In lab, we'll work with the two lambdas in this version of Quicksort:¹

```
(define o (f g) (lambda (x) (f (g x))))

(define qsort (xs)
  (if (null? xs)
      '()
      (let* ([pivot (car xs)]
             [rest (cdr xs)]
             [right? (lambda (n) (> n pivot))]
             [left? (o not right?)])
        (append (qsort (filter left? rest))
                (cons pivot (qsort (filter right? rest)))))))
```

You'll closure-convert this code by hand. To run the converted code, you'll need a test case.

[Click for a test case]

```
(define iota^ (n)
  ; return reversed list of natural numbers 1..n
  (if (= n 0) '() (cons n (iota^ (- n 1)))))

(check-expect
 (qsort '(65 15 87 42 62 45 6 81 53 34 33 82 79 7 17 39 71 18 98 92 77 41 51 16 86 30 49 10 4 68 35 52 69 12 85 3)
 (reverse (iota^ 100)))
```

Let's get started!

- (4) *Confirm that vscheme behaves as expected.* Compile the vscheme interpreter by running `make mosml` in your `src/vscheme` directory.

Capture `qsort` and its test case from the web browser and put them in file `qsort.scm`.

Confirm that the test passes:

```
$ vscheme < qsort.scm
The only test passes.
```

Make a copy of `qsort.scm` in file `qsort-orig.scm`.

¹Quicksort is used for demonstration purposes only. Quicksort is great for sorting a mutable array. An immutable list, not so much.

(5) *Closure-convert Quicksort by hand.* The closure handout describes three species of variables: *global*, *local*, and *free*. Start by identifying these species in the two `lambda` expressions in file `qsort.scm`.

A. The `lambda` in the `o` function contains names `f`, `g`, and `x`. Classify each of these names as *global*, *free*, or *local*.

B. The `lambda` in the `qsort` function contains names `>`, `n`, and `pivot`. Classify each of these names as *global*, *free*, or *local*.

C. Now rewrite each of these `lambda` expressions to closure-convert the code by hand:

- Each `lambda` becomes a closure record.
- In the vScheme embedding (which is what you are writing), the closure record is represented by a list with the revised `lambda` at the head, followed by all the captured variables.
- The revised `lambda` acquires a new, first parameter `$closure`, and internally, every reference to a free variable is replaced by code that extracts the variable's value from the `$closure` record.

The extraction may be implemented using only `car` and `cdr`, or for readability, you may use the predefined vScheme functions `nth` or `CAPTURED-IN`.

Implementations of `nth` and `CAPTURED-IN`

```
(define nth (n xs)
  (if (= n 0)
      (car xs)
      (nth (- n 1) (cdr xs))))

(define CAPTURED-IN (i xs) (nth (+ i 1) xs))
```

If you're not certain what the embedded code is supposed to look like, have another look at the embedding of `flip` in the closures handout.

D. Run your edited `qsort.scm` with the `vscheme` interpreter and confirm that the test still passes:

```
$ vscheme < qsort.scm
The only test passes.
```

If there are issues calling a non-function, be sure you updated and compiled your `vscheme` in step (3) and (4).

(6) *Embed closure-converted code.* Now shift your attention the UFT. This part of lab will solidify your understanding of closure representation.

In file `clscheme.sml`, write the embedding for the `C.CAPTURED` and `C.CLOSURE` forms. This is the embedding you just did by hand.

To generate Scheme code, you can generate `APPLY` nodes that call predefined `vScheme` functions like `cons`, `nth`, or `CAPTURED-IN`. Some common cases of such `APPLY` nodes are predefined for you in the `SU` module. This abbreviation refers to module `VSchemeUtils`, which is defined in file `vscheme.sml`.

Verify that your UFT builds.

After lab

The rest of the module should proceed in an orderly, methodical way. The module requires you to touch a lot of different files, so pace yourself. If you wait until Sunday, you'll have a rough time.

You'll proceed downward in stages: implement the closure conversion you've done by hand; K-normalize closures and references to captured variables; generate code for those K-normal forms; and implement new instructions to support the run-time representation of closures. As you'll see, I recommend doing the instructions *before* the code generator—that way you'll know exactly what you're targeting.

In the last part of the module, you'll visit the entire stack again, this time to add mutual recursion via `letrec`.

Closure-converting non-recursive lambdas

Closure conversion is small compared to the infrastructure needed to support it. I'm afraid this experience is typical of adding a new feature to a language.

- (7) *Change the type of your K-normalizer.* Your K-normalizer will eventually have to deal with two new forms; that's step (13). For now, go to file `knormalize.sml` and make the following changes:
 - Globally search for `FirstOrderScheme` and replace it with `ClosedScheme`.
 - Extend your K-normalizer function `exp` with three new pattern matches: `F.CLOSURE (lambda, captured)`, `F.CAPTURED i`, and `F.LETREC (bindings, body)`. For now, each right-hand side should call `Impossible.exercise "CLOSURE"` or something similar.
- (8) *Complete the UFT driver.* Confirm that your `git pull` has given you three new materializers in file `uft.sml`: `H0X_of`, `H0_of`, and `CL_of`. Then make these changes:
 - Change your function `KN_reg_of` to use the `CL_of` materializer instead of your old `F0_of` materializer.
 - Add these cases to your `translate` function:

```
| CL => CL_of      inLang >>> ! (emitCL outfile)
```

```
| H0 => H0_of      inLang >>> ! (emitH0 outfile)
| H0X => H0X_of    inLang >>> ! (emitH0 outfile)
```

Because H0 and H0X have the same internal representation, they share an emitter.

- Remove the wildcard case from your `translate` function.

Confirm that your UFT compiles.

- (9) *Identify free variables.* In file `closure-convert.sml`, define function `free` of type `X.exp -> X.name S.set`. (You can confirm the type by putting `val _ = free : X.exp -> X.name S.set` *after* the definition of `free`.) The `S` is an abbreviation for the `Set` module defined in file `set.sml`, which includes such operations as set union, set difference, and so on.

The free variables of an expression are defined in a set of proof rules in the book excerpt I posted on Slack. Those proof rules are expressed in terms of set membership, and what you need is a set of names. So the rules require translation:

- If there is a single way to prove membership, as in the `X.LOCAL` case, that's going to translate to a singleton set.
- If there are multiple ways to prove membership, as in the cases for `X.IF`, `X.BEGIN`, `X.FUNCALL`, `X.PRIMCALL`, and several others, that's going to translate using function `S.union`'.
- If there's a premise that says $y \notin A$, for some set A , that's going to translate using function `S.diff`.

The tricky cases are `X.LET` and `X.LETREC`. And `X.GLOBAL`—in the jargon of closure conversion, a global variable is not a free variable.

- (10) *Implement the core cases of closure conversion.* In file `closure-convert.sml`, build out internal functions `closure` and `exp`.

Begin with `exp`. This function is going to be a lot like the disambiguator you wrote in module 5: most cases are structural, and the real action is limited to just a few key forms. In closure conversion, those key forms are `lambdas` and references to local variables.

- When you get to the `X.LAMBDA` case, call internal function `closure`. Leave `closure` unimplemented until you get the other code to type-check.
- An attempt to read a local variable either stays as is or gets turned into `CAPTURE`. An attempt to *write* a local variable either stays as is or triggers an assertion failure (`Impossible.impossible`): if code tries to write a captured variable, there is a bug elsewhere in the UFT.
- In the `X.LETX` case, match only the `X.LET` form. Postpone `X.LETREC` until step (23). For now, use `Impossible.exercise`.

Keep in mind that the disambiguation pass you implemented in module 5 distinguishes global variables from other species of variables, but it does *not* distinguish *local* variables from *free* variables—both are represented using the value constructor `X.LOCAL`. To distinguish local variables from free variables, you’ll have to use the parameters passed to function `closeExp`.

Confirm that your UFT builds—that is, your code typechecks.

Next implement function `closure`. This function converts a higher-order `lambda` into a closure. The environments require careful attention:

- The names that are captured by the new closure are the free names of the `lambda`—or if you prefer, the names that are free in the body but are not formal parameters. Set functions `S.elms` and `S.ofList` may be useful here.

The names that are captured are not necessarily related to the `captured` list passed to `closeExp`. The body of the `lambda` is closed with respect to these names, *not* with respect to the `captured` names passed to `closeExp`.

- If a name on the free list is itself captured in the outer context—that is, if it *does* appear on the `captured` list passed to `closeExp`—then it has to go into the closure using the `C.CAPTURED` form, *not* as a local variable. That’s why in Closed Scheme the captured variables in a `closure` are represented as an `exp list`, not a `name list`. The easiest way to manage this conversion is to get `closeExp` `captured` to do it for you recursively.

Verify that your UFT builds.

Whoops! You need to close the definition forms as well (function `close`). These forms get closed in the same way as expression forms, except that a definition never occurs inside a `let` or a `lambda`, so it does not capture any variables.

- (11) *Test closure conversion.* Try out your closure conversion on the copy of Quicksort you made in step (4):

```
uft ho-cl qsort-orig.scm | less
```

The results may be a little hard to read, but if you called `CAPTURED-IN` in your embedding in step (6), you should be able to spot the uses and decide if they are OK.

If things look good, confirm that you can run the closure-converted, re-embedded code:

```
$ uft ho-cl qsort-orig.scm | vscheme
```

The only test passed.

Ripples downstream: K-normalization

You have now completed the main intellectual work of the module. The rest of the module requires you to propagate the `CAPTURED` and `CLOSURE` forms downstream—and eventually to deal with recursion.

(12) *Add capture and closure forms to K-normal form.* In your `knormal.sml` file, **define syntactic forms** for a captured variable and a closure.

- A captured variable has exactly the same form as in Closed Scheme: it's defined by a small integer slot number that is known at compile time.
- A closure is represented almost as it is in Closed Scheme, except the value of every captured variable must be in a register. My code uses ML's `withtype` modifier to define a type synonym:

```
withtype 'a closure = ('a list * 'a exp) * 'a list
  (* (funcode, registers holding values of captured variables) *)
```

I also define

```
type 'a funcode = 'a list * 'a exp (* lambda with no free names *)
```

The closure itself has to be combined with a `let` form: in addition to the `'a closure`, it must have a left-hand side of type `'a` and a body of type `'a exp`. I call this form `LET_CLOSURE`

Add these forms to your embedding in file `embedkn.sml`. You can clone and modify the embedding you defined in `clscheme.sml`. And add them to your renamer in file `knrename.sml`.

To get this code to compile, you will have to extend your code generator with two new cases. For now, leave them as calls to `Impossible.exercise`.

(13) *K-normalize captured and closure forms.* Extend your K-normalization function to handle the two new cases:

- The `CAPTURED` case doesn't have any subexpression, so it requires almost no thought.
- In a `CLOSURE`, if the list of captured variables is empty, then the closure can be K-normalized just as a `funcode` without any free names. This little optimization is worth doing because it makes the code both faster and easier to read.

If the list of captured variables is *not* empty, then a temporary register `t` has to be allocated to hold the closure while it is being initialized, and the captured variables ~~first~~ have to be put in registers, and then a K-normal form closure can be built by K-normalizing the `funcode` part. Finally you can create a `LET_CLOSURE` form by putting `t` in the

body. That is, the K-normal form of a closure C is not merely a C' ; it has the form `let t = C' in t`.

To put the captured variables in registers, use `nbRegsWith (exp rho)` from the K-normalization module. The continuation allocates the closure.

The conversion of a function has to be done twice, and you'll do it a third time when you do `LETREC`, so it's worth defining a helper function. Mine is

```
val funcode : F.funcode -> reg K.funcode
```

This function does almost the same thing as the code you already have to K-normalize the `F.DEFINE` form: the formal parameters go into an environment, and every nonzero register that is not used to hold a formal parameter is available. The only difference from `F.DEFINE` is that in a `lambda` expression, the function has no name, so the environment does not bind a function name to register 0.

Note that a `funcode` closure refers only to its formal parameters by name; during the closure-conversion process, every reference to a captured variable is replaced by an expression of the form `F.CAPTURED i`.

- (14) *Test K-normalization.* For an eyeball test, I would try to K-normalize the predefined functions `o` and `curry`.

```
echo '(lambda (f g) (lambda (x) (f (g x))))' | uft ho-kn
echo '(lambda (f) (lambda (x) (lambda (y) (f x y))))' | uft ho-kn
```

For a full test, you should be able to K-normalize `qsort-orig.scm` and then run the embedded code in the `vscheme` interpreter:

```
$ uft ho-kn qsort-orig.scm | vscheme
The only test passed.
```

Ripples downstream: Code generation and VM instructions

The SVM already has the data structures you need to finish the module. In particular, it has the `struct VMclosure` described in the closures handout. It remains only for you to define, implement, and generate the relevant instructions.

- (15) *Implement new VM instructions.* To create and use closures, you'll need three new VM instructions:

- You'll need an instruction that allocates a new closure. This instruction should take three operands: the register in which to place the new closure, the function that should go in the `f` field, and a literal saying how many slots to allocate for captured variables. This instruction will resemble the instruction for `cons`.

I've called mine `mkclosure`, and I unparse it using the template `"rX := closure[rY,Z]"`.

- You'll need an instruction that loads the value of a captured variable from a slot. This instruction should take three operands: the register into which the value should be loaded, the register holding the closure, and the literal index of the slot from which the value should be loaded. This instruction will resemble the instruction for `car` or `cdr`, except the slot index will be a field of the instruction, not hard-coded into `vmrun`.²

I've called mine `getclslot`, and I unparse it using the template `"rX := rY.Z"`.

- You'll need an instruction that stores the value of a captured variable in a slot. This instruction should take three operands: the register holding the closure, the register holding the value, and the literal index of the slot into which the value should be stored.

I've called mine `setclslot`, and I unparse it using the template `"rX.Z := rY"`.

Implement these instructions by updating `opcodes.h`, `instructions.c`, and `vmrun.c`.

- (16) *Modify call instructions so they understand closures.* In `vmrun`, update your implementations of `call` and `tailcall` so that the thing called can be a function *or* a closure. The only difference is that instead of getting a `struct VMFunction` straight out of a register, you put the value of `funreg` in a `Value` maybe named `callee`, and then check the tag. If it's a straight function, use `callee.f`, and if it's a closure use `callee.hof->f`. Anything else is a checked run-time error.
- (17) *Write utilities to generate the new instructions.* In file `asmutil.sml`, extend the *signature* of the `AsmGen` structure with these declarations:

```
val mkclosure : reg -> reg -> int -> instruction
  (* x := new closure with k slots; x->f := y; *)
val setclslot : reg -> int -> reg -> instruction
  (* x.k := y *)
val getclslot : reg -> reg -> int -> instruction
  (* x := y.k *)

val captured : reg -> int -> instruction
```

Implement the first three by using `A.OBJECT_CODE'` with the `O.REGINT` form of object code. Implement `captured` by using `getclslot` with the closure in register 0.

²And you could consider bounds-checking the index against the `nslots` field of the closure.

- (18) *Unparse the new instructions.* In file `asmparse.sml`, add patterns that match the new instructions and that unparse them according to the templates you chose in file `instructions.c` in step (15).
- (19) *Generate code for closures.* By now you know the data structure used to represent a closure, you have the instructions needed to manipulate it, and you have the utility functions that emit the instructions. You're ready to go back and finish the code-generator cases that you put off in step (12).
- To fetch the value of a captured variable, fetch it out of its slot in its closure, which is in register 0. Use `A.captured` or `A.getcslot`.
 - To allocate and initialize a new closure in a `LET_CLOSURE` form `let t = C in e`,
 - a. Load the closure's `funcode` into the temporary register `t`.
 - b. Using `A.mkclosure`, allocate the closure into that register.
 - c. Initialize the slots by emitting a sequence of instructions created using `A.setcslot`.

Then follow with the translation of the body `e`. Since a closure has to be translated in both `toReg'` and `return`, it may be worth creating a helper function..

During this step, the SML Basis Library function `List.mapi` could be useful, but it is missing from Moscow ML. Here you go:

```
(* mapi : (int * 'a -> 'b) -> 'a list -> 'b list *)
fun mapi f xs = (* missing from mosml *)
  let fun go k [] = []
      | go k (x::xs) = f (k, x) :: go (k + 1) xs
  in go 0 xs
  end
```

- (20) *Test code generation.* Eyeball your code generator by checking out results for `o` and `curry`:

```
echo '(define o (f g) (lambda (x) (f (g x))))' | uft ho-vs
echo '(define curry (f) (lambda (x) (lambda (y) (f x y))))' | uft ho-vs
```

You can compare your results with mine:

My assembly code for `o`

```
r0 := function (2 arguments) {
  r3 := function (1 arguments) {
    r2 := r0.1
    r3 := r0.0
    r4 := r1
    r3 := call r3 (r4)
    tailcall r2 (r3)
  }
}
```

```

    r3 := closure[r3,2]
    r3.0 := r2
    r3.1 := r1
    return r3
}
_G["o"] := r0

```

My assembly code for curry

```

r0 := function (1 arguments) {
  r2 := function (1 arguments) {
    r3 := r0.0
    r2 := function (1 arguments) {
      r2 := r0.1
      r3 := r0.0
      r4 := r1
      tailcall r2 (r3, r4)
    }
    r2 := closure[r2,2]
    r2.0 := r1
    r2.1 := r3
    return r2
  }
  r2 := closure[r2,1]
  r2.0 := r1
  return r2
}
_G["curry"] := r0

```

Once these look good, you can try Quicksort:

```

$ uft ho-vo qsort-orig.scm | svm
The only test passed.

```

Recursive and mutually recursive **lambdas**

Every functional language provides a mechanism that enables internal recursion and mutual recursion. In Scheme, it's `letrec`. The ideas are the same ideas you've already implemented, but the details can be fiddly. The key notion is that both the right-hand side `lambdas` and the body of the `letrec` are translated in the same environment with the same available registers: one in which each `lambda` corresponds to a closure that is stored in a known register. The rest is detail.

We'll implement the `letrec` from vScheme or Scheme, both of which require that all the right-hand sides be `lambdas`. The general form of `letrec` that is specified by the full Scheme standard can be translated into our simplified form (Waddell, Sarkar, and Dybvig 2005).

You'll build `letrec` from the bottom up. Everything in the SVM is already in place, so you'll start with K-normal form and code generation.

- (21) *Add LETREC to K-normal form, and translate it.* In both Closed Scheme and K-normal form, a LETREC recursively names closures, then evaluates an expression in the body. Only the representation of closures has changed.

The LETREC should leverage the closures you already have. Here's what mine looks like in K-normal form; compare it with the definition in file `clscheme.sml`:

```
datatype 'a exp
  = ...
  | LETREC of ('a * 'a closure) list * 'a exp
```

The translation of a single closure allocates the closure and initializes the slots. The translation of the closures in LETREC is the same, with one proviso: *all* of the closures are allocated before *any* are initialized. If every binding has the form (f_i, c_i) , with a list of captured variables of length k_i , then the translation is roughly like this:

```
let f1 = mkclosure c1 k1 in
let f2 = mkclosure c2 k2 in
  []
let fn = mkclosure cn kn in
  ( set slots of f1
    ; set slots of f2
      []
    ; set slots of fn
    ; e
  )
```

What makes it mutually recursive is that the list of captured variables in each closure may include some of the f_i registers, *even though those registers haven't been initialized yet*. That's OK, because when the K-normalizer emits the LETREC step (22), it knows that the code generator will initialize all the f_i registers before storing them in the closure slots (this step).

Complete this step in ~~two~~ three parts:

- A. Add LETREC to your K-normal form in file `knormal.sml`.
- B. Add cases to files `embedkn.sml` and `knrename.sml`. (Your compiler should complain that the relevant cases are missing)
- C. Extend your code generator to translate LETREC using a sequence of allocations and assignments.

Because the body of a LETREC can have any of the three destinies (to be returned, to be put in a register, or to be evaluated for effect), you'll

want to implement the translation using a higher-order helper function that is mutually recursive with `return`, `toReg'`, and `forEffect'`. A useful template might look like this:

```

fun letrec gen (bindings, body) =
  let val _ = letrec : (reg K.exp -> instruction hughes_list)
              -> (reg * reg K.closure) list * reg K.exp
              -> instruction hughes_list
          (* one helper function to allocate and another to initialize *)
          fun alloc (f_i, closure as (funcode as (formals, body), captures)) = ...
          fun init (f_i, closure as (funcode as (formals, body), captures)) = ...
        in hconcat (map alloc bindings) o hconcat (map init bindings) o gen body
        end
  and toReg' ...
  and forEffect' ...
  and return ...

```

Each of the other functions `toReg'`, `forEffect'`, and `return` will need to call `letrec` while passing itself as the `gen` parameter.³

(22) *K-normalize* LETREC. A LETREC is K-normalized using ideas and techniques you’ve already mastered. But a warning: I was not very successful reusing the code. My LETREC translation is a full 20 lines of ML, which is a lot. The code breaks down into these steps:

C. Allocate a fresh register for each closure.

Because the closures must all be allocated before any can be initialized, we can’t use `bindAnyReg` or `nbRegsWith` for this step. I wound up writing a helper function that takes one fresh register for each binding in the LETREC. My list of fresh registers is called `ts`.

D. Create a new available-register set `A'`: start with the currently available set and remove every register in `ts`.

If you like accumulating parameters, you could combine this step with the previous step. Or you could do it using an inscrutable fold.

E. Create a new environment `rho'` in which to K-normalize the right-hand sides and the body.

The environment extends the current environment with a new binding for each name in the original `letrec` to the corresponding register from the list `ts`. I used `ListPair.foldrEq` with `E.bind`.

F. Using the new available-register set `A'` and the new environment `rho'`, define a helper function that K-normalizes a single closure (in continuation-passing style).

³In the case of `toReg'`, “passing itself” means passing `toReg' dest` where `dest` is the current destination register.

Mine is called `closure`, and it has type

```
val closure : F.closure -> (reg K.closure -> exp) -> exp
```

It uses `nbRegsWith (exp rho')` to put all the captured variables into registers.

- G. Produce the K-normal LETREC by using the continuation-passing `map'` from module 9 with function `closure o snd`, where `snd` is defined by

```
fun snd (_, c) = c
```

This one gets a continuation that receives a list of K-normal closures `cs` and returns a K.LETREC whose bindings zip together `ts` and `cs` and whose body is K-normalized using `rho'` and `A'`.

- (23) *Closure-convert LETREC.* You'll finish by completing the closure-conversion function you started in step (10): in file `closure-convert.sml`, complete the case for `X.LET` with the `X.LETREC` keyword. Now that you know what's going on, it's simple: you closure-convert each right-hand side and the body.

There's just one subtle point: The source syntax in structure `X` permits *any* expression on the right-hand side of a `letrec`, but the target syntax in structure `C` permits only a `lambda`. Internal function `closure` already produces target syntax of the right type, but the natural argument is too general. I resolved the incompatibility with a function that assumes a `lambda` check is done in the parser:

```
fun unLambda (X.LAMBDA lambda) = lambda
  | unLambda _ = Impossible.impossible "parser failed to insist on a lambda"
```

Confirm that your UFT builds.

- (24) *Test mutual recursion.* To test mutual recursion, try the example below.

Classic slow test of parity using mutual recursion

```
(define parity (n)
  (letrec ([odd? (lambda (m) (if (= m 0) #f (even? (- m 1))))]
    [even? (lambda (m) (if (= m 0) #t (odd? (- m 1))))])
    (if (odd? n) 'odd 'even)))

(check-expect (parity 0) 'even)
(check-expect (parity 1) 'odd)
(check-expect (parity 30) 'even)
(check-expect (parity 91) 'odd)
```

What and how to submit

- (25) On Sunday, *submit the homework*. In the `src/uft` directory you'll find a file `SUBMIT.10`. That file needs to be edited to answer the same questions

you answer every week.

To submit, you'll need to copy your working tree to the department servers. We recommend using `rsync`, but `scp` also works.

Now log into a department server, change to your working tree, and submit your entire `src` directory:

```
provide comp150vm hw10 src
```

or if you keep an additional `tests` directory,

```
provide comp150vm hw10 src tests
```

- (26) On Monday, *submit* your reflection. Create a plain text file `REFLECTION`, which will hold your claims for project points and depth points.

For each project point you claim, write the number of the point, plus whatever is called for in the section “How to claim the project points”—usually a few sentences.

Now copy your `REFLECTION` file to a department server and submit it using `provide`:

```
provide comp150vm reflection10 REFLECTION
```

Overview of the code

Your new work is going to be concentrated in four files: `closure-convert.sml` (closure conversion), `knormalize.sml` (K-normalization of closures), `codegen.sml` (code generation for closures), and `vmrun.c` (new instructions that support closures). But a new language feature is what the software engineers are pleased to call a “cross-cutting concern,” and you'll change other files as well.

New things

New files you will write or edit

`clscheme.sml` This file defines Closed Scheme, which is the target language of closure conversion. Closed Scheme is just First-Order Scheme extended with `CLOSURE`, `CAPTURED`, and `LETREC` constructs. When you receive the file, it will have everything you need except a couple of cases of embedding of Closed Scheme into vScheme. You'll write these cases in step (6).

`closure-convert.sml` This is where you'll turn `lambdas` into closures. I provide a little scaffolding, but the bulk of the file is yours.

New files that are useful

`set.sml` This file defines a set abstraction that you can use to compute sets of free variables. You won't need to touch any of the code.

New file that is interesting for understanding or for depth points

mutability.sml This file contains a “mutability detector” which rejects any program that mutates a captured variable. The detector is very conservative and may reject programs that should be OK.

More interesting, this file also contains a placeholder for another translation pass, which migrates captured, mutable variables onto the heap. This pass can be implemented for depth points. Such a pass is needed to do some of the examples from my book chapter, like the resettable counters or the random-number generator.

New, boring file that can be ignored completely

foclutil.sml This file contains an embedding of First-Order Scheme into Closed Scheme. The embedding enables the UFT driver to materialize Closed Scheme from a file by first materializing First-Order Scheme and then embedding it. You can ignore this file completely. Why are you even reading this?

Old things

Interesting UFT files you will modify

knormal.sml You’ll add the CLOSURE, CAPTURED, and LETREC forms to K-normal form.

knormalize.sml You’ll K-normalize the new forms.

embedkn.sml You’ll embed the new forms into vScheme.

codegen.sml You’ll generate code for the new forms.

Uninteresting UFT files you will modify

uft.sml You’ll add cases to the driver’s `translate` function, so you can translate higher-order input languages.

asmparse.sml You’ll unparse the new instructions to suitable assembly code.

asmutil.sml You’ll define utilities that help generate the new instructions.

Familiar SVM files you will modify

opcodes.h You’ll add opcodes for instructions that allocate a closure, assign to a slot, and read from a slot.

instructions.c You’ll add table entries for those same instructions, including an unparsing template for readable assembly-language syntax.

vmrun.c You’ll implement the new instructions, and you’ll touch up your `call` and `tailcall` instructions so they can call closures.

Learning outcomes

Outcomes available for points

Learning outcomes available for project points:

1. *Names*. You understand all the species of names and where they come from.
2. *Global and local names*. You understand the different behaviors of local and global names.
3. *Embedding and projection*. You understand how to embed closures into vScheme.
4. *Embedded code and VM code*. You understand how the embedding relates to SVM code.
5. *Environments*. You understand the relationship between two different environments: the body of a function vs the environment where the function is defined.
6. *Closure conversion and code generation*. You can predict what will happen if a captured variable is overlooked.
7. *Closure conversion and mutable data*. You can say what would happen if we allowed arbitrary mutation in source code.
8. *Mutual recursion*. You know how to build the shared environment used by a nest of mutually recursive functions.

Learning outcomes available for depth points:

9. *More accurate mutation analysis [2 points]*. Improve my mutability detector so it is OK if a single function both mutates a variable and allocates a closure—as long as no mutated variable is *captured*. Demonstrate with a test case.
10. *Mutable variables on the heap [3 points]*. You implement the `Mutability.moveToHeap` pass, which migrates every *mutated*, *captured* variable into a mutable reference cell that is allocated on the heap. The pointer to that cell is not mutated so it can safely be shared among multiple closures.

You'll need new primitives equivalent to the `ref`, `!`, and `:=` primitives built into Standard ML; their implementations will be almost identical to `mkclosure`, `getcslot`, and `setcslot`, except they will operate on a `struct VMBlock`, not a closure.

Demonstrate with the random-number generator or the resettable counter from chapter 2 of my book.
11. *Mutable variables in closures [3 points]*. Improve your UFT so that a closure slot can be mutated, *provided* a static analysis shows that the slot

is the *only* location in which the variable is referred to. You’ll need to write the static analysis as well as update other UFT passes. Demonstrate your results with the random-number generator from chapter 2 of my book.

12. *Faster recursion for global functions [1 point]*. When a recursive function calls itself, the semantics of vScheme require that it look up its value in the global-variable table. But in the absence of mutation, it’s safe for it instead to call itself using register 0. Implement this improvement in your UFT, and demonstrate it on a long-running recursion. (Try either a very long tail recursion or an ordinary recursion that makes an exponential number of calls.)
13. *Reduced dependence on global variables [2 points]*. To prevent testing code from being compromised by malicious student code, Will Mairs and I developed a source-to-source translation we call “bulletproofing.” Bulletproofing transforms each `val` and `define` by introducing a `let` form that binds every free global variable with a local name. This transformation guarantees that the resulting code depends only on the values of the global variables at the time the *definition* is evaluated—if the values of those variables are changed afterward, the code will be unaffected.

Implement the bulletproofing transformation, either as a source-to-source transformation or as a pass inside your UFT. Using a long-running recursion, measure the performance improvement.

This transformation is routinely recommend to Lua programmers as a safe and easy way for them to speed up their code.

How to claim the project points

Each of the numbered learning outcomes 1 to N is worth one point, and the points can be claimed as follows:

1. To claim this point, give an example definition that contains a `lambda` whose body includes at least one name in each of these of these categories: a primitive, a global name that does not refer to a primitive, a local name, and a captured name. In your answer, identify one name from your example in each category.
2. To claim this point, identify the lines of your code that determine what variables are captured by a closures, and explain why these variables never include a global variable—even though according to the proof system, global variables are technically “free.”⁴
3. To claim this point, identify the lines of source code in your UFT where closures and captured variables are embedded into ordinary vScheme.

⁴Which is why the COMP 105 “improved closures” problem must capture global variables.

4. To claim this point, identify the lines of code in your `vmrun.c` that correspond to the `CAPTURED-IN` function that is predefined in `vScheme`.
5. To claim this point, identify the lines of code in your K-normalizer that K-normalize a closure, and explain how the code keeps track of which variable is in what register both where the closure is allocated and inside the closure.
6. Suppose that something goes wrong in your closure conversion and that not all captured variables are properly identified. (Perhaps there is a bug in the `free` function and it overlooks a free variable.) To claim this point, identify the line of code in your K-normalizer where the compiler would fail, and explain how the failure would manifest.
7. Imagine that the detector in `Mutability.detect` doesn't work—it always returns `Error.OK` applied to its argument, regardless of whether there are bad mutations in the source code. Assuming that there *are* bad mutations in the source code, if the detector doesn't work, the compiler will fail downstream. To claim this point, identify the line in your source code where the failure should occur.
8. To claim this point, identify the lines of code in your K-normalizer that build the environment used to compile all the right-hand sides and the body of a `LETREC`.