Module 9: K-Normalization

Introduction

This week you'll finally translate a real high-level language—and you'll be able to run a vast quantity of Scheme code.

- *What am I doing?*
    - Translate first-order Scheme into K-normal form.
- *Why am I doing it?*
    - You'll be running Scheme code! The UFT is almost done!
    - You'll get an idea how a compiler manages machine registers.
    - You'll capture a key compilation idea ("put this result in a register") using continuation-passing style.
- *How?*
    - Before lab you'll extend the UFT driver with support for K-normalization, and you'll read about continuation-passing style for register binding.
    - In lab you'll build a simple representation of register sets, and you'll get K-normalization working for two forms of expression: a literal, and a primitive applied to one argument. These forms will get you Scheme code that you can compile and run.
    - After lab, you'll complete your K-normalizer, starting with `check-expect` and working your way through all the syntactic forms of First-Order Scheme. When allocating multiple registers, as for both call forms and the `let` form, you'll use a higher-order function to separate policy from mechanism and to reuse code.

      You'll test your K-normalizer with at least one new test per syntactic form, plus a test suite derived from a CS 105 homework assignment.
    - At the end of the week you'll deliver a translator that translates first-order vScheme, including passes `fo-fo`, `fo-kn`, `fo-vs`, and `fo-vo`. Your translator will pass an impressive set of tests.

The module step by step

Before lab

Getting your UFT code ready for lab

(1) *Get new code.* Using `git pull`, get the new code for this module. I'm not expecting merge conflicts.

(2) *Add a new pass to the UFT.* You'll integrate K-normalization into the UFT by adding support for First-Order Scheme. This step is just like what you did in modules 5 and 6 to add support for K-normal form and code generation.

   This is a key step. Some other pre-lab steps can perhaps be fudged, but if you skip this one, you're unlikely to be able to use your lab time productively. The step looks like it has a lot of moving parts, but most of the parts are copy/paste/edit, so the step should go quickly.

   A. If you need to, review the handout on the UFT driver.

   B. Define reader function `FO_of_file`, which should work by composing the reader `schemexOfFile` with the projection function `FOUtil.project`. Its type should be

   ```
   val FO_of_file : instream -> FirstOrderScheme.def list error
   ```

   Function `FO_of_file` should look a lot like function `KN_of_file`.

   C. Define materializer function `FO_of`, which should materialize First-Order Scheme. If asked to materialize `FO` from `FO`, it should return `FO_of_file`; otherwise it should raise `Backward` (because as we'll see next week, nothing else translates into `FO`).

   D. Define emitter function `emitFO` by composing `emitScheme` with embedding function `FOUtil.embed`. Function `emitFO` should look a lot like function `emitKN`.

   E. Add a case to `translate` to handle the case when `outLang` is `FO`. It should look a lot like the other cases.

   F. Update the function `KN_reg_of` that you wrote in module 6. Your current version should have a catchall case that raises `NoTranslationTo KN`. Replace that case with a function that materializes `FO` and composes the result with `List.map KNormalize.def`. The materializer can fail, but K-normalization cannot fail, so you will have to manage the error type. If you need help, consult the handout on composing functions with error types.

   G. Update the `KN_text_of` function you wrote in module 5. The updated function should *either* read K-normal form from a file or should return a result from KN_reg_of, with registers renamed to strings. A suitable renaming will rename register 1 to string "r1", and so on.

Getting your brain ready for lab

(3) *Read about K-normalization and register allocation.* Read and understand the background handout *K-Normalization and Register Allocation*. Only two sections are important for lab: the introduction and the section "Our approach to register allocation." You can fudge the rest, but if you don't grok those sections, you may have a hard time being productive in lab.

(4) *Review the source and target languages.* Review the abstract syntax of first-order vScheme in file `foscheme.sml`. This language is a subset of the Unambiguous vScheme you worked with in module 5.

Review the abstract syntax of your K-normal form in file `knf.sml`, also from module 5.

You can fudge this step.

(5) *Look over the environment interface.* In file `env.sml` you'll find an environment interface. It's the same one used for disambiguation in module 5, but in module 5 you didn't have to pay much attention. In your K-normalizer you'll need not only to look up names in an environment but also to build new environments.

The only environment you'll use in lab is the empty environment `Env.empty`, so in practice you can postpone this step until after lab.

Lab

In lab you will start building the following ML module, which is templated for you in file `knormalize.sml`:

```
structure KNormalize :> sig
  type reg = int  (* register *)
  type regset     (* set of registers *)
  val regname : reg -> string
    val  exp  :  reg  Env.env  ->  regset  ->  FirstOrderScheme.exp  -
> reg KNormalForm.exp
 val def :                    FirstOrderScheme.def -> reg KNormalForm.exp
end
```

The `FirstOrderScheme` structure is defined in source file `foscheme.sml`; it is just like Disambiguated vScheme, except it doesn't have `LETREC` or `LAMBDA`. The `KNormalForm` structure is the one you built in module 5.

Register sets

(6) *Limited implementation of register sets.* You can do a full-blown implementation of sets, but because the K-normalizer always picks the *smallest* available register, I recommend a trick: represent only sets of *contiguous* available registers. For example,

```
datatype regset = RS of int  (* RS n = represents { r | r >= n } *)
```

A set of registers needs to support two main operations:

```
val smallest : regset -> reg
val -- : regset * reg -> regset   (* remove a register *)
```

- If somebody asks to remove a register `r` that is not the smallest in set `rs`, it is OK to remove additional registers. Just return what's left of `rs` after removing `r` *and* everything smaller than `r`.

Implement `smallest` and `--`. You might like to declare

```
infix 6 --
```

Allocating registers in continuation-passing style

(7) *Allocating registers with smart let bindings.* The primary operation performed by the K-normalizer is "put the value of this expression in a register." This operation will be implemented by function `bindAnyReg`, which ensures that a *target-language* expression (type `exp`) is in a register—allocating that register if need by. Function `bindAnyReg` uses continuation-passing style; the continuation has type `reg -> exp`.

```
type exp = reg KNormal.exp
val bindAnyReg : regset -> exp -> (reg -> exp) -> exp
```

As described in the background handout, `bindAnyReg` has this contract: `bindAnyReg A e k` returns an expression `e'` that is roughly equivalent to

⟦`let t = e in $(k t)`⟧, where `t ∈ A`

In detail,

- Evaluating `e'` may kill registers in `A` (and only in `A`)
- The result of evaluating `e'` is equivalent to the result of evaluating `let t = e in e''` where `t` is an element of set `A` and expression `e''` is produced by applying continuation `k` to `t`.
- If `e` has the form of a name `y`, then `e'` equals `k y`. That is, if `e` is already in a register, `bindAnyReg` reuses that register; it does not allocate a new one.

Less precisely, `bindAnyReg` binds the value of `e` into a newly allocated register, *unless* `e` is in a register already. It then continues with the identity of the register that holds `e`.

Function `bindAnyReg` need not manage the set of available registers. That will be the job of the calling function, `exp`.

Implement `bindAnyReg`.

K-normalizing a primitive call

(8) *Special-case primitives.* Implement enough cases so that you can K-normalize a call to `print`, like this:

```
$ echo "(println 'first-steps)" | uft fo-kn
(let ([$r0 'first-steps]) (println $r0))
```

You'll need these cases:

- A *special* case in `exp` that handles `F.PRIMCALL (p, [e])`
- A case in `exp` that handles `F.LITERAL v`
- A case in `def` that handles `F.EXP e`

The source-language expression `F.PRIMCALL (p, [e])` should be K-normalized by recursively K-normalizing `e`, binding the result to a register, and generating a primitive call in K-normal form. (A very similar example can be found in a handout.) In step (18) you will generalize this case to handle multiple arguments to a primitive.

The `F.LITERAL v` case you can figure out.

The `F.EXP e` case means an expression at top level, so that means K-normalizing `e` in a context where all registers are available and the environment is the empty environment `Env.empty`.

After lab

Be alert to `let*`

You might be surprised by seeing `let*` in your debugging output, when you haven't done anything to support it. As always, `let*` is syntactic sugar for a nested series of let expressions. My parser desugars `let*` using function `letstar` in file `vscheme-parse.sml`. And my prettprinter resugars nested `let` forms into `let*` , using function `nestedBindings` in file `wppscheme.sml`. So internally there are only `let` and `letrec`; `let*` is *only* syntactic sugar.

The desugaring and resugaring enables us to work with more readable code.

Unit tests

Next you'll build unit tests. They make testing everything else easier.

(9) *Unit tests.* Your source language is a subset of Unambiguous vScheme, and the `F.CHECK_EXPECT` form includes strings representing both expressions. To K-normalize `F.CHECK_EXPECT`, all you need to do is generate a sequence of the form `(e₁; e₂)` where `e₁` calls primitive `check` and `e₂` calls primitive `expect`. Each primitive call will have the form `@(t, v)`, where `@` is `P.check` or `P.expect`, `t` is a register holding the value of an expression and `v` is the literal string describing the source code of the expression. Because such a call is a primitive call taking exactly one register, the code will closely resemble your K-normalization of `print` in step (8). You may want to write a helper function.

The `F.CHECK_ASSERT` form is similar.

In your `def` function, write cases that K-normalize the `F.CHECK_EXPECT` and `F.CHECK_ASSERT` forms. You will use primitives `P.check`, `P.expect`, and `P.check_assert`. And since these forms are executed at top level, all registers are available. You'll test this code in the next step.

(10) *Start a test file.* Test your work so far by creating file `kntest.scm`. This file should contain first-order Scheme that your system can successfully K-normalize, generate code for, assemble, and run. Start with these three tests:

```
;; F.LITERAL, F.CHECK_EXPECT, F.CHECK_ASSERT
;;
(check-expect (number? 3) #t)
(check-expect (number? 'really?) #f)
(check-assert (symbol? 'really?))
```

Confirm that everything works:

```
> uft fo-vo kntest.scm | svm
All 3 tests passed.
```

Easy cases

First-Order Scheme has a bunch of forms that neither allocate registers nor manipulate environments. These forms are relatively easy to K-normalize—every subexpression uses the same set of available registers as its parent expression

(11) *Local variables.* Using the environment passed to `exp` as a parameter, K-normalize the `F.LOCAL` and `F.SETLOCAL` forms. Neither of these forms has to allocate a register.[1]

You won't be able to test local variables until you can create some local variables—for example, using `let` or `lambda`. But once you've

[1]The disambiguator guarantees that every local variable is defined. It is up to the K-normalizer to build the environment correctly so that the lookup always succeeds. If your environment lookup fails, either there is a bug in your disambiguator or there is a bug in your K-normalizer.

done step (20), come back and add test cases to `kntest.scm`, and document them with comments `;; F.LOCAL` and `;; F.SETLOCAL`.

(12) *Global variables.* Using the environment passed to `exp` as a parameter, K-normalize the `F.GLOBAL` and `F.SETGLOBAL` forms.

- The `F.GLOBAL` form translates into a call to primitive function `P.getglobal`.
- The `F.SETGLOBAL` form, like `print`, takes a parameter that has to be allocated into a register. And its translation is nuanced: in Scheme source code, `F.SETGLOBAL` is executed *both* for value *and* for side effect. But in K-normal form, primitive `P.setglobal` is executed *only* for side effect. So the `F.SETGLOBAL` form has to be translated into a `let` expression whose body has the form `(setglobal(localname, globalname); localname)`.

K-normalize these two forms. Add test cases to `kntest.scm` and document them with comments like those in step (11).

Now K-normalize the `F.VAL` definition form. It desugars into a `F.SETGLOBAL`.

(13) *Sequence.* The Scheme `begin` form sequences the evaluation of a list of expressions: none, one, or many. No registers have to be allocated or preserved. An empty `begin` must evaluate to `#f`; a nonempty `begin` must evaluate to the result of its last expression. To implement this semantics, it will be useful to define a recursive helper function.

K-normalize form `F.BEGIN`, add test cases to `kntest.scm`, and document the test cases.

(14) *Control flow.* K-normalize the `F.IFX` and `F.WHILEX` forms. Both forms include a condition that must be stored in a register.

- The `F.IFX` form is K-normalized by evaluating the condition, binding the result to a temporary register `t`, and using `t` in the K-normal form `if`. This transformation works because the condition is evaluated only once.

  Also, once the condition has been evaluated, the decision is made, so register `t` is dead. That means register `t` is available to be reused in both branches.

- The `F.WHILEX` form is K-normalized by allocating an available register `t` that does *not* bind the result of the expression. Instead that register is used in the K-normal form as in `while t := e do e'`, where the condition is evaluated multiple times. Register `t` is assigned to on every trip through the `while` loop.

  As in the `if`, register `t` is available to be used in both `e` and `e'`.

K-normalize `F.IFX` and `F.WHILEX`, add test cases to `kntest.scm`, and document the test cases.

Environment manipulation: Function definitions

The `define` form K-normalizes into an assignment to a global variable. The right-hand side of the assignment is `FUNCODE`, and the key is to get the right names and environments:

- Formal parameters arrive in registers 1 through n, where n is the number of formal parameters.
- The body should be K-normalized in an environment where the function's name stands for register 0 and the name of each formal parameter stands for the corresponding register. The environment can be built using `foldl` or a recursive function.

  When the body is K-normalized, registers 0 to n are unavailable; the first available register is register $n + 1$.

To set the global variable, you need to put the `FUNCODE` in a register. At top level, register 0 is always available.

(15) *Function definition.* In your `def` function, K-normalize the `F.DEFINE` form.

Add a simple test case to your `kntest.scm` and document it. (Until you can call the function, about all you can do is apply a type predicate like `function?`.)

Expressions that allocate multiple registers: Calls

Calls are the first form for which your K-normalizer has to manage the set of available registers. For example, if you are K-normalizing `(append e₁ e₂)`, the register that holds the normal form of `e₁` is *not* available while `e₂` is being normalized. And the computation of the available-register set depends on the form of the call: A *primitive* call may use any available registers; a *function* call must use consecutive available registers. Each form therefore requires a different register-allocation policy. But they are otherwise K-normalized using the same algorithm. You will implement this algorithm using a higher-order function that takes the register-allocation policy as a parameter.

The core of the algorithm, which implements both calls, is to bind a *list* of actual parameters to a *nest* of `let`-bound names. Compared with `bindAnyReg`, there are two complications:

- The actual parameters can't be K-normalized independently: register allocation for later parameters must be affected by the registers chosen for earlier parameters. For example, if I am K-normalizing a call like `(cons e₁ e₂)`, and if the value of `e₁` is bound to register `t₁`, then `t₁` is not available to be used in the computation of `e₂`. This dynamic can be seen in the example K-normalization of `(+ 2 3)`.

- Primitive calls and function calls are governed by different register-allocation policies: a primitive can take its arguments in any registers, but a function must have its arguments in *consecutive* registers. For example, a call like `(append xs ys)` might be K-normalized into code like this:

  ```
  let r7 = append in let r8 = xs in let r9 = ys in call r7 (r8, r9)
  ```

  This call would kill (that is, potentially overwrite) registers numbered `r7` and up.

(16) *Warmup: Normalize a primitive with two arguments.* Add another special case in `exp` that handles `F.PRIMCALL (p, [e1,e2])`. This case should create a continuation that removes a temporary register from the available set.

Test your code as follows:

```
$ echo "(println (car (cons 'second-step '())))" | uft fo-vo | svm
second-step
```

There are too many cases to deal with them all by hand. Instead, you will define a higher-order function that handles *every* first-order form that takes multiple arguments. This function will be parameterized by a register-allocation policy. The type of a policy is the same as the type of `bindAnyReg`:

```
type policy = regset -> exp -> (reg -> exp) -> exp
```

A policy looks at the `exp` and `regset` and chooses a `reg` to be passed to the continuation (which has type `reg -> exp`).

You will define a function `nbRegsWith`, which takes a *normalizer* and a policy and returns a register-allocation function for *lists* of items.

```
type 'a normalizer = regset -> 'a -> exp
val nbRegsWith : 'a normalizer -> policy ->
                 regset -> 'a list -> (reg list -> exp) -> exp
```

The name is short for "K-**n**ormalize and **b**ind to **reg**ister**s**." With the help of a suitable parameter of type `'a normalizer`, this function can K-normalize any list, but in this module, you will use it only to normalize lists of first-order Scheme expressions.

By contract, `nbRegsWith normalize p A es k` returns a nested `let` expression that

- Sequentially K-normalizes each expression in `es` (using `normalize`)
- Sequentially binds each K-normalized expression to a register in `A` according to policy `p`

- Marks registers bound to earlier expressions as “not available” for the K-normalization of later expressions.

- Finishes with expression `k ts`, where `ts` is the list of registers to which expressions in `es` are bound

What is the continuation `k` of type `reg list -> exp`? It’s a function that takes the list of registers that hold the values of expressions `es`. When you get a list of registers, most likely you are going to use it to make a primitive form like `@(`x_1`,…,`x_n`)` or a function-call form like `x(`x_1`,…,`x_n`)`. If the continuation has a form like `fn ts => ⟦e'⟧`, then the result returns by `nbRegsWith` will be morally equivalent to this:

`let` t_1 `=` e_1 `in … let` t_n `=` e_n `in e'`

The moving parts all cooperate:

- Policy `p` finds registers t_1 to t_n in `A`
- Function `normalize` puts each expression e_1 through e_n into K-normal form.
- Continuation `k` builds `e'`
- Function `nbRegsWith` orchestrates it all.

If your continuation-passing skills are rusty, have a look at the example of `map` in continuation-passing style.

You will implement `nbRegsWith` and use it to K-normalize both forms of call.

(17) *Implement function* `nbRegsWith`. Define function `nbRegsWith`. The key ideas are as follows:

   - Function `nbRegsWith` is an ordinary recursive function that consumes a list, and it deals with just two cases: the input list is either `[]` or `e::es`.

   - When the input list is `[]`, the result is just `k []`.

   - When the input list is `e::es`, `nbRegsWith` must K-normalize `e`, bind the result to a register according to policy `p`, then continue by recursively binding the remaining `es`. The binding to `e` may use any of the available registers in `A`. The recursive bindings to `es` may also use any of those available registers *except* the register to which `e` is bound.

     Each of these bindings requires a continuation: the call to `p` needs a continuation that expects the single temporary to which `e` is bound. And the recursive call needs a continuation that expects the *list* or temporaries to which the remaining `es` are bound.

   The key to getting the code right is constructing the right continuations to pass to both the policy and the recursive call. These are new continuations that you will have to synthesize using `fn`. (You might wish to review the synthesis of new continuations in the Boolean-formula solver from CS 105.)

If your continuations have the right types and they remove the register that binds `e`, you are probably in good shape.

List processing in continuation-passing style: An example

If your continuation skills are rusty, check out this example where I transform an ordinary recursive `map` into a CPS `map'`. The standard ("direct style") `map` looks like this:

```
val map : ('a -> 'b) -> 'a list -> 'b list
fun map f = []
  | map f (x :: xs) =
      let val y  = f x
          val ys = map f xs
      in  y :: ys
      end
```

The intermediate results are `let`-bound to names `y` and `ys` for this reason: in the continuation-passing version, those names become lambda-bound names (parameters to a continuation written with `fn`). And the function parameter is also in continuation-passing style:

```
val map' :  ('a -> ('b -> 'answer) -> 'answer)
         -> 'a list
         -> ('b list -> 'answer)
         -> 'answer
fun map' f' [] k = k []
  | map' f' (x :: xs) k =
      f' x (fn y => map' f' xs (fn ys => k (y :: ys)))
```

Function `map'` synthesizes *two* continuations: one passed to `f'` which receives `y`, and one passed to the recursive call, which receives `ys`. These two continuations cooperate to build the list `y :: ys` which is then passed to the original continuation `k`.

Your function `nbRegsWith` will be structured along similar lines. In the role of function `f'`, you will provide a `policy`, and the `'answer` type will be `exp`. And you will synthesize two continuations: one that takes a single temporary register, and one that takes the rest of the temporary registers.

Confirm that `nbRegsWith` has the right type by placing the following lines *after* the definition of `nbRegsWith`:

```
val nbRegsWith : 'a normalizer -> policy -> regset -> 'a list -
> (reg list -> exp) -> exp
  = nbRegsWith
```

(18) *Implement primitives.* Retire your special-case primitive code from step (8). Replace it with general-case code that K-normalizes

`F.PRIMCALL (p, es)`, where list `es` may contain any number of expressions.

The template I provide you for the `exp` function includes an internal definition of `nbRegs`, which normalizes and binds expressions in the *current* environment `rho`. Your case for `F.PRIMCALL` should use `nbRegsWith` with `bindAnyReg` as the policy.

Add suitable test cases to `kntest.scm` and document them.

(19) *Define a consecutive-register policy.* Define a function

```
val bindSmallest : regset -> exp -> (reg -> exp) -> exp
```

that behaves just like `bindAnyReg`, except it doesn't optimize for the case of an expression already in a register. That is, `bindSmallest e A k` returns `let t = e in e''` where `t` is the smallest register in set `A` and expression `e''` is `k t`. When used sequentially, `bindSmallest` will produce consecutive registers.

If you like, you can refactor `bindAnyReg` so that it and `bindSmallest` share some code.

(20) *Implement function calls.* K-normalize the `F.FUNCALL` form. You will need to use `bindSmallest` to put the function in the smallest available register, then K-normalize the arguments using `nbRegs` with `bindSmallest` as the policy. When K-normalizing the arguments, do not overwrite the register that holds the function.

Like the helper function `nbRegsWith` itself, this one is all about finding the right continuations.

(21) *Test function calls.* Add simple test cases to `kntest.scm` and document them.

- Tests for function calls and function definitions
- Tests for local variables that you put off in step (11)

For a more ambitious test, try un-nesting calls, as in this example from a functional Quicksort:

```
(append (qsort (filter left? rest))
        (cons pivot (qsort (filter right? rest))))
```

K-normalizes to

```
(let* ([$r0 append]


       [$r1 qsort]
         [$r2 filter]
         [$r3 left?]
         [$r4 rest]
         [$r2 ($r2 $r3 $r4)]
```

```
        [$r1 ($r1 $r2)]

        [$r2 pivot]
          [$r3 qsort]
            [$r4 filter]
            [$r5 right?]
            [$r6 rest]
            [$r4 ($r4 $r5 $r6)]
          [$r3 ($r3 $r4)]
        [$r2 (cons $r2 $r3)])

  ($r0 $r1 $r2))
```

Expressions that allocate multiple registers: Scheme's `let`

Like a call, a `let` binds a sequence of expressions to a list of registers. What's interesting is the K-normalization of the body: The body needs to be normalized in an environment that knows in what register each `let`-bound name is placed. And in the body, none of those registers are available.

(22) *Review the difference between* `let` *and* `let*`*.* The difference is entirely in the environments. In a `let` form, no bound names are visible on any right-hand side. All right-hand sides are evaluated in the same environment. In a `let*` form, names bound in earlier bindings are visible on the right-hand sides of later bindings. Every right-hand side is evaluated in a different environment.

   Optional: Grab the Scheme interpreter from Chapter 5 of *Programming Languages: Build, Prove, and Compare* (page 310), and compare how environments are manipulated in the interpretation of `let` and `let*`.

   Create test cases in first-order Scheme that shows the difference between `let` and `let*` with the same bindings. Be sure that at least one test case includes something of the form `(let ([x y] [y x]) …)`, where both `x` and `y` are *local* names. (Function parameters will do.) Confirm your understanding by making sure your tests pass the `vscheme` interpreter.

(23) *Implement* `let` *bindings.* K-normalize the `F.LET` form. I encourage you to use `nbRegs`; you can figure out an appropriate policy. The continuation will have to remove *all* the bound registers from the available set. You can implement this operation in a special-purpose recursive function, or you can use a fold with a "flipped" version of function `--`.[2]

   Some functions in the [ListPair].meta interface are useful here.

   Although `F.LET` bindings "feel" parallel, in K-normal form all the names are bound sequentially. The parallel feel comes entirely from the fact that

[2]You'll have to define `flip`.

all the right-hand sides are K-normalized in the same environment.

Run your code on the `let` binding from the test case you wrote in step (22). If you choose the `bindAnyReg` policy and if both `x` and `y` are already in registers, **the swap will generate no code at all**. Your UFT will simply K-normalize the body of the `let` using a different environment in which the names are swapped. With the `bindAnyReg` policy, that's expected behavior, not a bug. If you have concerns about that behavior, choose a different policy.

Add your tests to `kntest.scm`. Document them.

Integration test

(24) *Predefined functions.* Use the `vscheme` interpreter to extract the *first-order* predefined functions:

```
vscheme -predef | grep -vw lambda > fopredef.scm
```

Confirm that your system can compile and load all these functions:

```
uft fo-vo fopredef.scm | svm
```

There should be no output: no test results, no complaints.

If so, **place your compiled functions in the `build` directory** by running

```
make predef
```

from the `vscheme` directory.

(25) *End-to-end testing.* Demonstrate the complete viability of your system via end-to-end testing:

- If you are now taking or have taken CS 105 at Tufts, create a file `scheme105.scm` that contains your complete solution to the CS 105 `scheme` homework (the third homework) of whatever version you took. Update the source code as follows:
  - If any `check-error` form takes more than one line, remove it.
  - If any definitions or tests depend on `lambda` (likely if you did `arg-max`), remove those also.

  Confirm that all the `check-expect` and `check-assert` tests pass with both the `uscheme` and `vscheme` interpreters.
- If you have not taken CS 105 at Tufts, implement a recursive merge sort on a list of integers, *not* higher-order. Write test cases. Put your results in a file `msort.scm`. Confirm that the tests pass with `vscheme`.

Confirm that your stuff compiles with your UFT and that all tests pass with the SVM. In your `bin` directory I have provided a shell script that compiles and runs first-order code with the first-order predefined functions:[3]

```
$ run-fo-with-predef scheme105.scm
```

If anything goes wrong, **take notes**, which you will use in a learning outcome.

(26) *Congratulate yourself.* Congratulations! You have implemented a complete (if small) first-order language that can run efficiently on commodity hardware. We won't stop now—we'll be doing higher-order functions next week—but this is a good time to recognize just how far you've come.

What and how to submit

(27) On Monday, *submit the homework.* In the `src/uft` directory you'll find a file `SUBMIT.09`. 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 cs106 hw09 src
```

or if you keep an additional `tests` directory,

```
provide cs106 hw09 src tests
```

(28) On Tuesday, *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 cs106 reflection09 REFLECTION
```

Overview of the code

Code you will write or edit

`uft.sml` The UFT driver, which you have edited before. You'll be adding an new language and translation, just as you did in modules 5 and 6.

[3]This shell script has some sanity checks. For example, if your compiled predefined functions are older than your UFT, it complains.

`knormalize.sml` You'll define your K-normalizer here.

Code you will need to understand

`env.sml` A definition of environments. Your `find` calls should always succeed, so you should **not** catch the `NotFound` exception—if that exception is raised, it indicates a bug in your UFT.

`foscheme.sml` Defines First-Order Scheme, which is a subset of Unambiguous vScheme. The ASTs are identical, except that First-Order Scheme lacks `lambda` and `letrec`.

New code that you don't need to care about

`foutil.sml` Embedding and projection for First-Order Scheme. This is the easiest embedding/projection pair ever. Because the code is boring and repetitive, I've written it for you.

Learning outcomes

Outcomes available for points

Learning outcomes available for project points:

1. *Craft.* You can add a new pass to the UFT driver.
2. *Continuations and register allocation.* You can define a continuation that reserves an allocated register, preventing its reuse.
3. *Continuations and register reuse.* You can define a continuation that reuses an allocated register.
4. *Functional programming.* You can use higher-order functions to avoid near-duplicate code.
5. *Calling conventions.* You can implement a procedure calling convention.
6. *New calling conventions.* You can identify implications of changing calling conventions.
7. *Syntactic-form testing.* Your K-normalizer is comprehensively tested.
8. *Integration testing.* Your K-normalizer passes an integration test.
9. *Functional languages and mutation.* You can explain how the possibility of mutation affects choices available to the UFT.

Learning outcomes available for depth points:

10. *Let-floating [3 points].* Use the equations of translation (the ▯ function) from module 6 to show when the following two expressions have the same translation:

- `let y = (let x = e₁ in e₂) in e₃`
- `let x = e₁ in (let y = e₂ in e₃)`

Hint: it works when `x` is not free in `e₃` and in one other special case.

Then deploy your insight in your K-normalizer to rewrite the first form to the second form. You'll know you've got it right when your `uft fo-kn` produces no nested `let` expressions—my prettyprinter will sugar everything into `let*`.

Hint: define a so-called "smart constructor" to use in place of `K.LET`.

11. *More powerful* `if` *instructions [3 points].* Extend your SVM to include two-register `if` instructions that can compute a Boolean expression and "skip next if false", all in a single VM instruction. For example, `if r1 < r2`. Similarly, extend your SVM to include a one-register `if` instruction that uses the type predicate `null?` in the condition.[4] Extend your K-normal form and your K-normalizer to exploit these instructions. Demonstrate your extensions and measure how much the extension can shrink your generated VM code.

12. *Small-integer literals [3 points].* Extend your SVM to include instructions in `R2U8` or `R2I8` format, so that expressions like `x > 0` or `n + 1` can be computed using a single VM instruction. Extend your K-normalizer to exploit these instructions. Demonstrate your extensions and measure how much the extension can shrink your generated VM code.

13. *Optimized* `let` *expressions [2 points].* K-normalize `let` by defining a hybrid policy that results in code that is superior to whatever you would get by using a one-size-fits-all policy in step (23).

14. *Efficient compilation of long list literals [1 point].* Using strict left-to-right evaluation for long list literals uses a number of registers proportional to the length of a list. That means, for example, that we can't compile a literal list with 15 numbers, because we would run out of registers. But if the first argument to `cons` is a literal value, we can change the evaluation order to compute the second element first. Do so, and confirm that your UFT can compile a literal list of numbers, no matter how long, using only two registers.

15. *Polymorphic code generation [1 point].* Assuming you've already implemented A-normal form, define your K-normalizer as a functor so that your K-normalizer can generate *both* K-normal form *and* A-normal form, just by applying the functor to two different actual parameters.

16. *Bignums in vScheme [2 points].* (Not related to K-normalization.) Repurpose your SML bignum implementation from CS 105 to work inside the vScheme interpreter. Add a suitable type of value, and update the arithmetic primitives so they do mixed arithmetic. (Alter higher-order

[4] Other type predicates optional.

function `arithOp` to handle mixed inputs and to provide a single point of truth about promotion rules.)

17. *Bignums in the SVM [4 points].* (Not related to K-normalization.) Implement bignum arithmetic in the SVM. You can port one of your implementations to C, or you can port my array-based implementation from Smalltalk. Or you can try using the GNU multiprecision library (gmp).[5]

How to claim the project points

1. To claim this point, submit source code that compiles and builds a `uft` binary that understands what `fo-kn` is asking for.
2. To claim this point, identify a line of your code that contains a continuation passed to `bindAnyReg`, `bindSmallest`, or a similar function, and explain how the continuation reserves the allocated register to prevent its reuse.
3. To claim this point, identify a line of your code that contains a continuation passed to `bindAnyReg`, `bindSmallest`, or a similar function, and explain how the continuation does *not* reserve the allocated register but rather allows its immediate reuse.
4. To claim this point, identify (by line number) *every* case in your K-normalizer that binds a list of expressions to a list of registers, and confirm that each case uses the same higher-order function.
5. To claim this point, identify the lines of your code where it is determined that a function's *incoming* actual parameters are in consecutive registers starting at register 1.
6. Suppose we change the calling convention so that the function register and argument registers are *not* killed by a call. Instead they are required to have the same values after the return that they had at the call. With this change, it becomes very difficult to use the `tailcall` instruction except for direct recursion. And other changes, unrelated to tail calls, might also be required in the UFT. To claim this point, identify `one` such required change, either in the K-normalizer or in the code generator.
7. To claim this point, submit a file `kntest.scm` in which every value constructor in `foscheme.sml`, for both `exp` and `def` types, is exercised by some test. Each test must be documented by a comment that names the value constructor or value constructors that it tests. And `uft fo-kn kntest.scm` must actually generate code. (It is not necessary for the generated code to run or for all the tests to pass.)
8. To earn this point, your system must run and pass all the tests in `scheme105.scm`. To claim the point, let us know that you accomplished this goal, and in the reflection, tell us how many tests are included in the

[5]Field reports suggest that gmp is a mixed blessing. At best.

file. If anything went wrong in your first run of step (25), let us know one thing that went wrong and how you fixed it.

9. To claim this point, justify your choice of policy for K-normalizing a `let` expression in step (23). Justification should include an explanation of why another policy is inferior and should be demonstrated with a code example.

   Keep in mind that vScheme local variables are mutable; that's what makes this issue difficult.