Module 5: Disambiguation and K-Normal Form

Introduction

This week you'll start your journey from a machine-level language to a high-level language. You'll implement a language called *K-normal form*, which is nearly a subset of Scheme. Then next week, you'll generate assembly code from K-normal form, and you'll be able to run your first high-level-language code.

- *What am I doing?*
  - Understand the three different roles names can play in Scheme code, and write a compiler pass that uses special-purpose syntax to identify what role each occurrence of each name is playing.
  - Define an ML datatype to represent expressions in K-normal form, which is a low-level subset of Scheme.
  - Define an embedding/projection pair that relates K-normal form to vScheme syntax.
  - Extend the UFT driver with support for K-normal form.
- *Why am I doing it?*
  - Translating names is one of the deepest parts of any compiler, because it can be done only within some kind of context or environment that says what each name stands for. Moving the contextual information into syntax makes all the subsequent compiler passes much, much easier.
  - K-normal form is the next step in our long-term plan of lifting machine-level code up to something nice. It is the lowest-level intermediate code that resembles a high-level language more than an assembly language—and "making machine code look like a high-level language" is a useful tactic in almost any compiler.
  - Embedding and projection make it possible to debug. To get K-normal form out of your translator, just embed it into vScheme and use my prettyprinter for vScheme. To get K-normal form into your translator, write vScheme and project it into K-normal form.

- *How?*
  - Before lab you'll study the roles of names, and you'll learn two related representations of vScheme: `VScheme`, in which a name's role has to be learned by looking up the name in the environment; and `UnambiguousVScheme`, in which the name's role is made explicit in the abstract syntax.
  - In lab you'll complete the disambiguation pass that looks up each name in the environment and selects the correct syntax for each occurrence. I've done the boring parts; you'll do the interesting parts.
  - After lab you'll study the grammar of K-normal form and its relationship to Scheme. Then you'll define its representation, an embedding into `VScheme`, and a projection from `UnambiguousVScheme`.
  - Finally you'll add support to the Universal Forward Translator so it can read and write K-normal form. At the end of the week, you'll deliver a working Universal Forward Translator that includes a `kn-kn` pass. This pass reads vScheme, disambiguates it, projects it to K-normal form, embeds the result back into vScheme, and pretty-prints it.

    You'll also deliver test cases: one that exercises every syntactic form in KNF, and one each for every possible way that projection can fail.

Detailed instructions are below.

The big picture

K-normal form is about two things: language paradigm and naming.

- Target paradigm (assembly language): imperative code, commands, assignment, resembles a subset of C
- Source paradigm (K-normal form): applicative code, function calls, `let` bindings, **is** a subset of Scheme

Once we have K-normal form, we'll add features until eventually we'll be compiling full Scheme.

Ideas about naming are deeper. The key idea is this: **in syntax, all names look alike, but at run time, different names behave differently**. And those differences can be discovered by inspecting a *compile-time* environment:

- At run time, a global name refers to a location in the VM state's `globals` table.
- At run time, a local name (one that is bound by `let` and `lambda`) refers to a VM register.

- The name of an ordinary function (like `map` or `foldr`) codes for an ordinary function call.
- The name of a primitive function (like `cons` or `+`) codes for a VM instruction.

In Scheme source code, all these names are written using the same syntax. In K-normal form, each species of name and each species of call is written using explicitly different syntax—the syntax instantly identifies which variables are global vs local and which calls are ordinary vs VM instructions. These syntactic distinctions are independent of K-normal form; are also made in a high-level language called "unambiguous vScheme". Transforming original vScheme into unambiguous vScheme, a translation I call *disambiguation*, is the subject of this week's lab.

Aside from the explicit distinctions of names and calls, K-normal form adds two key invariants to unambiguous Scheme:

- The value of every intermediate expression has a name.
- Function definitions do not nest. (That is, no function may refer to parameters or variables of an enclosing function.)

These two invariants are common to many low-level intermediate languages for many different kinds of compilers.

Here's a snapshot from the video, cleaned up and with this week's parts highlighted:

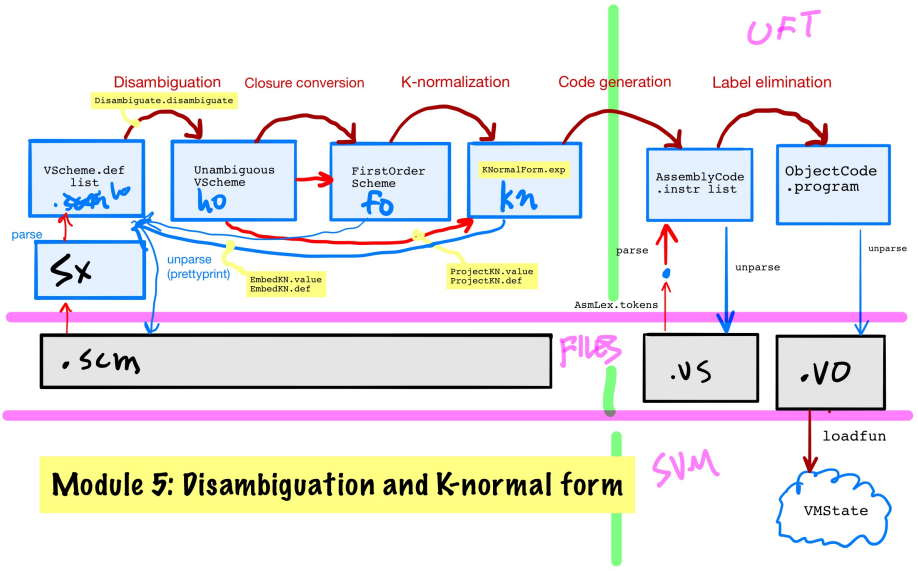

Figure 1: UFT/SVM system with highlights

The module step by step

Before lab: Understand the big picture, especially names.

(1) *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 main`). You should not have any merge conflicts.

(2) *Verify your build.* Verify that you can build the `uft` binary with `make`, and that it recognizes the translation `ho-ho`. Any expression with a name or a function application should exit with an uncaught exception, but expressions without names and applications should go through:

```
> echo '(+ 2 2)' | uft ho-ho
Uncaught exception:
LeftAsExercise: disambiguate APPLY

> echo '(if #t 1 0)' | uft ho-ho
(if #t 1 0)
```

(3) *Understand the big picture.* The overview video about the high-level languages of the UFT, including all the transformations we will implement presents the big picture of the Universal Forward Translator. Watch it before lab.

(4) *Understand the three meanings of names.* Read the first part of the handout on Unambiguous vScheme. Follow up by reading the `referent` type and `referent` function defined at the beginning of file `disambiguate.sml`.

(5) *Study vScheme.* Look over the handout on vScheme, and study the definition in module `VScheme` in file `vscheme.sml`. Be sure you understand the forms `VAR`, `SET`, and `APPLY`; these are the forms you will be disambiguating in lab.

(6) *Study Unambiguous vScheme.* Complete the handout on Unambiguous vScheme, and study the definition in module `UnambiguousVScheme` in file `vscheme.sml`. Be sure you understand the forms `LOCAL`, `GLOBAL`, `SETLOCAL`, `SETGLOBAL`, `FUNCALL`, and `PRIMCALL`; these are the forms that replace the ambiguous forms in step (6).

(7) *Understand eta expansion.* In file `disambiguate.sml`, be sure you understand what function `etaExpand` is doing. Eta expansion converts a *primitive* into an ordinary *function*. If you have not encountered the eta expansion or reduction in CS 105, you can consult any of the following resources:

- Short explanation from MLton, an ML compiler
- Short documentation, from Dotty, a Scala compiler

- Long post on Medium about Scala
- YouTube video also about Scala

To paraphrase one of the longer posts,

> Eta-expansion is what compiler does "behind the scenes" when it notices that you need a *value*, but are provided a *primitive*.

(8) *Review.* During lab, the ambiguous forms will be replaced:

- The original `VAR` form must be disambiguated into either `LOCAL` or `GLOBAL`.
- The original `SET` form must be disambiguated into either `SETLOCAL` or `SETGLOBAL`.
- The original `APPLY` form must be disambiguated into either `FUNCALL` or `PRIMCALL`.

Before lab begins, be sure you can answer these questions:

A. For each of the three possible referents of a name (local, primitive, and other global), what form replaces the `VAR` form?

B. For each of the three possible referents of a name (local, primitive, and other global), what form replaces the `SET` form?

C. If a name appears in function position in `APPLY`, for each of the three possible referents of the (local, primitive, and other global), what form replaces the `APPLY` form?

D. If a non-name appears in function position in `APPLY`, what form replaces the `APPLY` form?

(9) *Consider reading ahead.* If you want the opportunity to ask informed questions during lab, you could consider reading the handout on K-normal form.

Lab: Disambiguate names

(10) *Disambiguate names.* In file `disambiguate.sml`, complete function `exp`: Using your answers to the questions in step (8), replace all uses of `LeftAsExercise` with new code, then remove the definition of exception `LeftAsExercise`.

Function `exp` is nested inside function `exp'`, which tracks context by keeping list `locals`. The list contains all formal parameters and let-bound variables. That list is used by function `referent` to determine the referent of any vScheme name.

It's worth revisiting the disambiguation section of the handout on Unambiguous vScheme.

You can test your disambiguator by running

`uft ho-ho`

After lab: K-normal form

Defining K-normal form

(11) *Define K-normal form.* In this step you define an internal representation (abstract-syntax tree) for K-normal form. Start by reading all about K-normal form.

Now edit module `KNormalForm` in file `knf.sml`. Redefine type `'a exp`, which takes the representation of a *name* as a type parameter.[1] In your definition, each of the metavariables in the K-normal form handout should be represented as follows:

Table 1: The atomic forms

Metavariable	Representation
`e`	`'a exp`
`x`	`'a`
`v`	`literal`
`@`	`vmop`

Types `literal` and `vmop` are already defined in the file; they are (respectively) `ObjectCode.literal` and `Primitive.primitive`.

If a type called `string` or `name` appears anywhere in your representation, you are doing it wrong. The type of a name has to be unknown; it's type parameter `'a`.

Before you move on to the next step, get a sanity check from a classmate or from a member of the course staff.

(12) *Implement the K-normal form utility functions.* File `knf.sml` has placeholders for two functions `setglobal` and `getglobal`. Each is an abbreviation that creates an expression of the form `@(`x_1`, ..., `x_n`, v)`, where `@` is either `Primitive.getglobal` or `Primitive.setglobal`. For `getglobal`, `n` is zero, and for `setglobal`, `n` is one.

The abbreviations aren't strictly necessary, but they will save you some hassle down the road.

Embedding K-normal form into Scheme

(13) *Embed K-normal form into (ambiguous) vScheme.* In file `knembed.sml`, you will find a template for two functions, `value` and `def`, with these types:

[1] In μScheme source code, a name is a string, but in assembly code, a name will be a register. Making it a type parameter enables us to use K-normal form on both sides of the house.

```
val value : KNormalForm.literal -> VScheme.value
val def   : VScheme.name KNormalForm.exp -> VScheme.def
```

We embed directly into (ambiguous) `VScheme` because our goal is to reuse the prettyprinter for `VScheme`, and it's actually easier than embedding unto `UnambiguousVScheme`.

Implement these functions.

- The `value` function is actually a projection, not an embedding: the VM code supports ~~floating-point values,~~ string values and `nil`, neither of which can be written as vScheme literals. Nonetheless, we're going to treat it as if it were an embedding—we're going to cheat.
  - ~~Embed a real value as its nearest integer, using function `Real.round`.~~
  - Embed a string value as a symbol (lame, but the best we can do).
  - Embed `nil` as the Boolean `false`.
- The definition embedding uses only one definition form, `VScheme.EXP`. Internally, the main embedding should be

  ```
  val exp : VScheme.name KNormalForm.exp -> VScheme.exp
  ```

  The embedding is described by equations for in the KNF handout. To implement those equations in ML code, you will use these value constructors:

K-normal form object language	embeds in μScheme using metalanguage
`v`	`LITERAL`
`x`	`VAR`
`getglobal(STRING s)`	`VAR`
`setglobal(x, STRING s)`	`SET`
`@(`x_1`, …, `x_n`)`	`APPLY`
`@(`x_1`, …, `x_n`, v)`	`APPLY`
`x(`x_1`, …, `x_n`)`	`APPLY`
`if x then `e_1` else `e_2	`IF`
`let x = e in e'`	`LET`
`(`e_1`; `e_2`)`	`BEGIN`
`x := e`	`SET`
`while x := e do e'`	`WHILE` and `LET`
`funcode (`x_1`, …, `x_n`) => e`	`LAMBDA`

Constructing vScheme `LET` forms is a bit of a nuisance, so take advantage of the `let'` helper function that I have provided for you.

The embedding has a few tricky cases:

- The `@(`x_1`, ..., `x_n`, v)` form has two special cases, where `@` is either `getglobal` and `setglobal`. Unfortunately, because `@` has an abstract type (for good reasons), you cannot pattern match on it. Instead you have to pattern match on the result of calling function `P.name`.
- The `funcode` form should embed as `lambda` even though they don't have identical semantics.

One note: my prettyprinter condenses nested `let` expressions into Scheme's `let*` form, so if you see a `let*` in your output, it is expected.

Projecting Scheme into K-normal form

(14) *Project unambiguous vScheme expressions into K-normal form.* In file `knproject.sml`, you will find a template for two functions, `value` and `def`, with these types:

```
val value : UnambiguousVScheme.value -> KNormalForm.literal
val def   : UnambiguousVScheme.def -> string KNormalForm.exp Error.error
```

Internally you will also define

```
val exp   : UnambiguousVScheme.exp -> string KNormalForm.exp Error.error
```

The projection will enable us to read K-normal form code from disk, using the Scheme lexer and parser, plus the disambiguator you wrote in step (10). We project from `UnambiguousVScheme` because it is easier than going direct from `VScheme`.

The projection of expressions is described by equations for in the KNF handout. The left-hand sides that look ambiguous in those equations are exactly the ones that you disambiguated in step (10).

The equations don't specify a projection of values. But every vScheme value can be represented as a K-normal form literal. And in fact, the `value` direction is actually an embedding, as suggested by its type. A vScheme symbol embeds as a K-normal form string, and the other forms of value embed one for one.

Important: The projection functions simply change the representation of vScheme code that is *already* in K-normal form. You won't translate general Scheme to K-normal form until module 9. For this module, if Scheme code is *not* already in K-normal form, the projection should fail for one of two reasons: either a form is outright unacceptable, or the form has a non-variable in a position where a variable is expected. The forms that are outright unacceptable are as follows:

- Any `while` loop whose condition does not have the form

  ```
  (let ([x e]) y)
  ```

- Any `while` loop whose condition has the form

  `(let ([x e]) y)`

  but in which `x` is different from `y`
- Any `begin` not of the form `(begin` e_1 e_2`)`
- Any `let` with more than one binding
- Any `letrec` form
- ~~Any `lambda` form except for the special case of a global function definition, which should be handled by the `def` function~~

In step (18) you'll write a test for each of these forms, so if you like, you can start writing those tests now.

Even a good form like `if` can be rejected if it doesn't satisfy the invariants of K-normal form. The key invariant is that many forms are required to be names. One example is the condition in an `if` expression: if the condition isn't a name, then the `if` expression isn't in K-normal form. In this case the projection should fail. It is useful to define a helper function that insists on getting a name:

```
val asName : X.exp -> X.name Error.error
       (* project into a name; used where KNF expects a name *)
  = fn X.LOCAL x => succeed x
    | e => error ("expected a local variable but instead got " ^ (X.whatIs e))
```

This function will be used to project `FUNCALL` and `PRIMCALL`, which should be rejected unless every argument is a variable; `IFX`, which should be rejected unless the condition is a variable; and `SETGLOBAL`, which should be rejected unless the right-hand side is a variable. (A `FUNCALL` will also be rejected unless the function being called is a variable.)

Managing all the potential sources of error requires a lot of plumbing, but we can hide the details by using the same abstraction we used in the lexer and parser: an "applicative functor." The code I give you includes suitable abbreviations:

```
infix  3 <*>  val op <*> = Error.<*>
infixr 4 <$>  val op <$> = Error.<$>
val succeed = Error.succeed
val error = Error.ERROR
val errorList = Error.list
```

Using these functions, the bureaucracy of error handling becomes manageable. For example, here's my code for projecting `if` expressions:

```
fun exp (X.IFX (e1, e2, e3)) =
      curry3 K.IF <$> asName e1 <*> exp e2 <*> exp e3
```

You are now ready to define the projection function for expressions.

(15) *Project unambiguous vScheme definitions into K-normal form.* The last step is to project definitions. This step is easier because the `val`, `check-expect`, and `check-assert` forms simply aren't in K-normal form; the projection fails. There are really only a couple of special cases, to do with functions.

- The main case is `X.EXP`; you just pass the payload to your internal `exp` function. This case will cover 90% of your needs, and you can write it first.
- ~~Although the general case is the one to write first, it has to follow this special case: A top-level expression of the form~~

  ```
  (let ([t (lambda (xs) e)]) (set f t))
  ```

  ~~should be projected into the K-normal form~~

  ```
  let t = funcode xs => ⌊e⌋ in setglobal("f", t)
  ```

  ~~where ⌊e⌋ stands for the projection of e (that is, ⌊e⌋ = exp e).~~

  ~~One fine point: you have to match on~~

  ```
  (let ([t (lambda (xs) e)]) (set f t'))
  ```

  ~~and then insist that t = t'.~~

  ~~To generate the K-normal form `let` expression, I define an internal helper function `fundef` of type `string` KNormalForm.exp -> string KNormalForm.exp. It takes ⌊e⌋ as a parameter and returns a representation of the K-normal form expression~~

  ```
  let t = funcode xs => ⌊e⌋ in setglobal("f", t)
  ```

  ~~I then handle the `lambda` case with~~

  ```
  fundef <$> exp e
  ```

- Strictly speaking, `define` isn't in K-normal form, but it can be handy to pretend. Try inventing a `t`, like say

  ```
  val t = "$t1"
  ```

  and then project

  ```
  (define f (xs) e)
  ```

  as

  ```
  let t = funcode xs => ⌊e⌋ in setglobal("f", t)
  ```

You're now ready to extend the UFT to handle your new functions.

Adding a pass to the Universal Forward Translator

(16) *Add a new pass to the UFT.* Once you have defined your embedding, disambiguation, and projection functions, you can extend the UFT driver by adding support for K-normal form.

A. Begin with the handout on the UFT driver. It explains how the UFT driver works and what code has to be written to incorporate a new language.

B. Define reader function `KN_of_file`, which should work by composing the reader `schemexOfFile` with the projection code you wrote in step (15). Its type should be

```
val KN_of_file : instream -> string KNormalForm.exp list error
```

C. Define materializer function `KN_text_of`, which should materialize K-normal form. It should look almost exactly like `VS_of`: read the code from a file or bleat that there is no translation.

D. Define emitter function `emitKN` by composing `emitScheme` with your embedding function.

E. Add a case to `translate` to handle the case when `outLang` is `KN`.

Testing

(17) *Preliminary testing.* Your UFT should now be capable of running your embedding and projection. Test by running

```
uft kn-kn
```

Here are some cases for you to test:

```
(+ 2 2)                                ;; should be rejected

(let* ([$r0 2] [$r1 2]) (+ $r0 $r1))  ;; should be OK

(if (< n 0) #t #f)  ;; reject

(let* ([$r0 n] [$r1 0] [r2 (< $r0 $r1)]) (if $r0 #f #t)) ;; accept
```

(18) *Systematic testing.*
Create a test file for each of the forms that should be rejected by the projection function in step (14), plus one more file that contains all the good forms.

`bad.while.scm`	Condition in `while` isn't the right `let`
`bad.whilevars.scm`	In `let` in `while` condition, names don't match
`bad.begin.scm`	`begin` with wrong number of subexpressions
`bad.let.scm`	`let` with wrong number of bindings

`bad.letrec.scm`	Any `letrec`
`bad.lambda.scm`	Any `lambda` that is not a global definition
.	.
`good.scm`	One each of every good KNF form (as embedded into Scheme)

The `good.scm` should contain Scheme versions of `x`, `v`, `@(x1, ..., xn)`, `x(x1, ..., xn)`, `if x then e1 else e2` `let x = e in e'`, `(e1; e2)`, `x := e`, and `while x := e do e'` forms. It is ok if the "VM operation with literal" and `funcode`/`lambda` forms are omitted.

18b. *Round-trip testing*

Testing success and failure in step (17) is likely sufficient. But if you want to test *semantics*, you can do it by comparing round-trip results from your UFT/SVM combo with result from the `vscheme` interpreter.

For example, if `test.scm` evaluates a `begin`, a couple of `let` bindings, and a few primitives, it might look like this:

```
(begin
   (let* ([tmp 2]
          [tmp (+ tmp tmp)])
     (check tmp 'two-plus-two))
   (let ([tmp 4])
     (expect tmp 'four)))
```

This code can be run through `vscheme` *without* using embedding and projection, and then again *with* embedding and projection:

```
$ cat test.scm | vscheme
The only test passed.
$ cat test.scm | uft kn-kn | vscheme
The only test passed.
```

The call to `uft kn-kn` puts my code through embedding and projection, which doesn't change the test results.

What and how to submit

(19) On Monday, *submit the homework.* In the `src/uft` directory you'll find a file `SUBMIT.05`. 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 hw05 src
```

(20) 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 reflection05 REFLECTION
```

Reading in depth

Occasionally I'll suggest reading that may enrich your view of programming-language implementation.

- *Normal forms.* K-normal form is defined by Birkedal, Tofte, and Veilstrup (1996), who use it to infer properties about memory use. It is based on the more famous *A-normal form* of Flanagan *et al.* (1993), which is definitely worth reading.
- *Prettyprinting.* My prettyprinter uses an interface designed by Wadler (1998), who pitches his design as an improved version of the original by Hughes (1995). Both papers are well worth reading.

  My code uses only Wadler's interface; my back end is a clean-room implementation of an algorithm by Pugh and Sinofsky (1987). If you like that sort of thing, it's a nice application of dynamic programming.

Learning outcomes

Outcomes available for points

You can claim a project point for each of the learning outcomes listed here. Instructions about how to claim each point are found below.

1. *Global and local names.* You can create a term in which a single name appears as both a global variable and a local variable.
2. *K-normal form.* You can write simple K-normal form by hand.
3. *K-normal form embedding.* You can, by hand, embed simple K-normal form into Scheme.
4. *Names in K-normal form.* You can say which names in the source code show up as what types in K-normal form.
5. *Embedding, projection, and language design.* You can justify the fact that K-normal form has fewer expressions but more literals than Scheme.
6. *Eta-expansion.* You can say which of the K-normal form invariants is satisfied by the body of an eta-expanded primitive.
7. *UFT driver.* Your `uft` builds and understands what `kn-kn` is asking for.
8. *Testing.* You can explain the results of your tests.
9. *Notation.* You can use Oxford brackets to write translation equations.

You can claim a depth point for floating `let` out of `let`, and two more for improving the Scheme prettyprinter.

10. *Let-floating transformation [1 point].* When variable `y` is chosen so it is ~~distinct from x and~~ the same as `x` or not free in e_3, the following expressions are equivalent:

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

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

    But the second expression results in code that is easier to read and that runs faster on some platforms (Peyton Jones, Partain, and Santos 1996). Implement let-floating on K-normal form. See what sort of difference it makes to the generated VM code.

11. *Prettyprinting [2 points].* The indentation and line breaks for the vScheme prettyprinter are just barely tolerable. Improve them.

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, write an expression in vScheme in which `x` appears both as a `GLOBAL` variable and as a `LOCAL` variable.

2. To claim this point, write an expression, using the ML-like syntax of K-normal form from the handout on K-normal form, that corresponds to the ML expression `n + 1`, where `n` is a local variable.

3. To claim this point, embed the previous expression into valid vScheme. That is, write an expression in vScheme that is the embedding of an expression in K-normal form; the expression that is embedded must correspond to the Scheme expression `(+ n 1)`, where `n` is a local variable.

4. You must understand all the relevant categories of the ways names can be used in Scheme: formal parameter, local variable, global variable, user-defined function, and primitive function. And in your ML representation of K-normal form, you must understand the use of each of these types:

   - Type `'a`
   - Type `vmop`
   - Type `literal`

   To claim the point, for each of the three types listed, say what categories of Scheme names are represented by values of that type.

5. Observe that expressions in K-normal form are a *subset* of vScheme expressions, but literals in K-normal form are a *superset* of (Unambiguous) Scheme values. It seems strange to have the relation point in opposite directions. To claim this point, answer these two questions:

   - In a system that is targeting the SVM but does not necessarily want to be locked into translating Scheme, why is it a good idea to have K-normal form expressions be a *subset* of Scheme expressions?

   - In a system that is targeting the SVM but does not necessarily want to be locked into translating Scheme, why is it a good idea to have K-normal form literals be a *superset* of Scheme values?

6. To claim this point, analyze the eta-expansions produced by function `eta-Expand` in file `disambiguate.sml`. This function returns a `lambda` written in Disambiguated vScheme. Analyze the **body** of the `lambda` and say which of the K-normal-form invariants it satisfies and why.

7. To claim this point, submit code that compiles and runs so that `uft kn-kn` produces a sensible result, better than "I don't know how to translate." This point is awarded for *running* the translation; you get the point even if one or more of the functions have bugs.

8. To claim this point, say in a few short sentences what your tests tell you about what parts of your code do and don't work. This point is awarded for understanding the results of your tests; you get the point even if your UFT does not yet behave the way you hope.

9. To demonstrate the Oxford brackets, you should be able to specify a key element of a translation you already know well: the translation from assembly language to object code that you implemented you implemented in the previous module. This translation is specified by a function

   $\mathcal{A}$ `: AssemblyCode.instr -> int -> (name -> int) -> Object-Code.instr`

   The parameter of type `int` is the position that the instruction occupies in the instruction stream. The parameter of type `name -> int` is the environment ρ; it is the mathematician's way of writing an environment of type `int Env.env`.

   To demonstrate ability with Oxford brackets, it is sufficient to write an equation that describes the translation of just one instruction: the `GOTO` instruction. When function $\mathcal{A}$ is given an assembly-language `GOTO` with a label, it turns an object-language `GOTO` with a PC-relative offset. To claim the point, use Oxford brackets to write an equation that describes just the translation of the `GOTO` instruction. Notate the position parameter as k and the environment parameter as ρ.