Module 6: Code Generation

Introduction

This week you'll translate K-normal form into assembly language. Then you'll be able to run vScheme code—as long as it's in K-normal form and it calls only primitive functions.

- *What am I doing?*
  - Define a utility function that will enable you to change the representation of register names in K-normal form.
  - Update the UFT driver so it can translate from K-normal form to assembly code, enabling new passes `kn-vs` and `kn-vo`.
  - Define two of the three code-generation functions that translate K-normal form into assembly code.
- *Why am I doing it?*
  - This translation is the most dramatic step in the UFT. You'll learn how to translate primitive operators as well as control-flow constructs like `if` and `while`. And you'll start to identify *contexts* of translation, which is a key concept in compilers and which will eventually enable you to optimize tail calls.
  - The utility function and the UFT-driver update will enable you to take vScheme code (in K-normal form) all the way through your compiler to the SVM and run it. You'll also be able to run the exact same code using the `vscheme` interpreter, so you can compare results.
- *How?*
  - Before lab you'll implement name-changing code for K-normal form, which will enable you to pull K-normal form *with register names* off of disk. You'll also study how K-normal form is translated to assembly code.
  - In lab you'll add the code-generation translation to the UFT driver, which will enable you to run `uft kn-vo`, taking K-normal form Scheme all the way to virtual object code. You'll also implement three cases

for the code-generation functions, so you can compile and run a simple Scheme expression.

– After lab, you'll complete your code generator, you'll test every syntactic form, and you'll make sure both lists of primitives (UFT and SVM) are up to date. At the end of the week, you'll deliver a working Universal Forward Translator that includes a `kn-vo` pass.

The module step by step

Before lab (essential steps)

In lab you'll add your code generator to the UFT driver, and you'll take the first steps in implementing it. To get any of this done productively, you'll need to have the stubs for the code, and you'll need to understand how code generation works.

(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`). Merge conflicts are unlikely.

(2) *Study code generation.* Everything you need to generate assembly language from K-normal form is described in a handout. To be productive during lab, the most important sections are the opening example, the translations of `let` bindings, and the explanation of destinies. The equations of translation are also super useful, but you won't need them until the lab starts.

(3) *Review the secrets of the UFT driver.* In lab you'll update a materializer function and use a reader function. Review the materializer function `KN_text_of` that you wrote last time, and remind yourself of the type of the reader function `KN_of_file`. If you don't remember what materializer and reader functions are or how they fit together, review the handout on the UFT driver and the note on function composition with error types.

Before lab (recommended steps)

To maximize the value of your lab, I'm recommending that you complete two more steps before the lab begins. If you complete these steps, then by the end of the lab, you should be able to *compile the following expression to virtual object code and run it in your SVM:*

```
(let ([r100 'victory]) (println r100))
```

Here's the demo:

```
> echo "(let ([r100 'victory]) (println r100))" | uft kn-vo | svm
victory
```

To make the demo possible, and to test whatever other code you may write in lab, your UFT needs to be able to translate that `r100` into register number 100. That's this step:

(4) *Change the representation of names in K-normal form.* In order to generate code, you need K-normal form in which each name is identified with a particular machine register: type `ObjectCode.reg KNormal.exp`. But your projection function produces a K-normal form in which a name is a string: type `string KNormal.exp`. To get K-normal form of the right type, you'll implement a `mapx` function. Function `mapx` copies the *structure* of a K-normal form expression while changing the representation of each *name*. (If you think of K-normal form as a tree, `mapx` changes some of the values at the leaves, but it does not touch internal nodes.)

    Not every string represents the name of a register, so the change of names is a projection, which can fail. The change of names is already defined for you as function `AsmLex.registerNum`, and it has type `string -> Object-Code.reg Error.error`. Your job is to lift the function on names up to a function on expressions. The lifting function is higher-order and polymorphic; you will define

    ```
    val mapx : ('a -> 'b Error.error) ->
               ('a KNormalForm.exp -> 'b KNormalForm.exp Error.error)
    ```

    This function combines two skills you already have:

    - The function requires the tedious copying of program structure that you find in function `Disambiguate.disambiguate`.
    - The function requires the management of projection failure that you find in function `ProjectKN.def`.

    Once you have mastered the `<$>`, `<*>`, and `succeed` functions from the `Error` interface, the coding will go quickly.

Your lab experience will also benefit if you know about two convenience functions:

(5) *Learn two functions in* `asmutil.sml`*.* File `asmutil.sml` contains the `AsmGen` interface, which exports a bunch of functions that make it more convenient to write your code generator. Everything here can also be done using functions in the `Primitive`, `AssemblyCode`, and `ObjectCode` interfaces, but with more effort and with more opportunities for error.

    Before lab, all you need to know is these two functions:

    - Use `effect` to generate a `println` instruction.
    - Use `loadlit` to generate a `loadliteral` instruction.

Lab

(6) *Add a translation to the UFT driver.* In this step you'll make the UFT driver aware of the translation you're about to write (and for which stub code already exists).

A. To complement the materializer `KN_text_of` that you wrote last time, write a function `KN_reg_of` that materializes a program of type `ObjectCode.reg KNormalForm.exp list`.

   When asked to materialize K-normal form from input language KN, your `KN_reg_of` function should use the reader `KN_of_file`, plus functions `KNRename.mapx` and `KNRename.regOfName` from step (4). If you didn't complete step (4) before the lab, you can still write this code and get it to typecheck, but when you run it the UFT will halt with an assertion failure.

   Because function `KN_of_file` produces a *list* of expressions, but `KNRename.mapx KNRename.regOfName` renames *one* expression, you will want to use `Error.mapList`. The easiest way to get this code right is to define a helper function that does the renaming. The benefit of defining a helper function is that you can give it an explicit type signature, which will help the compiler tell you what you need to do:

```
val KN_reg_of_KN_string :
      string KNormalForm.exp list ->
      ObjectCode.reg KNormalForm.exp list error
  =  ... fill in with a composition of functions ...
```

   You can then compose this function with `KN_of_file`. If you want guidance, consult the composition handout. Once everything works, you can inline the helper function, or if you prefer, keep it as a named helper function.

   When asked to materialize K-normal form from an input language *other* than KN, your `KN_reg_of` function should raise `NoTranslationTo KN`—the translation to K-normal form is the topic of module 9.

B. Update function `VS_of`, which you first encountered in module 5; the promised "main event" has arrived. Replace the `NoTranslationTo` exception with a short materialization pipeline: call `KN_reg_of` on `inLang`, and pipe the result through a helper function with this type constraint:

```
val VS_of_KN : ObjectCode.reg KNormalForm.exp list ->
               AssemblyCode.instr list
  = ... fill in with a composition of functions ...
```

   To implement `VS_of_KN`, you will find it useful to employ `List.concat`.

Because code generation cannot fail, function `VS_of_KN` does not have an `error` in its type. To compose it with `KN_of_file`, which *does* have `error` in its result type, you'll need to use `Error.map` in the pipeline. I recommend using the abbreviation `!`, which is already defined in file `uft.sml`. For details, consult the composition handout.

(7) *Implement three cases for your code generator.* To compile the expression `(let ([r100 'victory]) (println r100))`, you need to be able to compile the literal `'victory` into a register (function `toReg'`), and you need to be able to compile the `let` expression and `println` call for side effect (function `forEffect'`).[1] Implement these three cases, then test your system at the Unix command line:

```
echo "(let ([r100 'victory]) (println r100))" | uft kn-vs

echo "(let ([r100 'victory]) (println r100))" | uft kn-vo | svm
```

After lab

After lab, you will complete your code generator and test it. You will also verify that your UFT and SVM agree on primitives.

One more instruction

(8) *Generate code for computation.* Using function `setreg` from file `asmutil.sml`, extend your code generator so it can compile and run

```
(let* ([r100 2] [r101 (+ r100 r100)]) (println r101))
```

A pause for infrastructure

(9) *Learn the rest of the* `AsmGen` *interface.* Return to the `AsmGen` interface in file `asmutil.sml` and identify the other functions that look useful.

(10) *Implement register-register move.* In file `asmutil.sml`, correct the implementation of `copyreg`. Your SVM will need an opcode for an instruction that copies a value from one register to another. Just use that opcode with the internal `regs` function.

Code generation and testing

(11) *Test a simple check-expect.* At this point your UFT ought to be able to read in this K-normal form and translate it to virtual object code:

```
(let* ([$r0 2] [$r1 2] [$r0 (+ $r0 $r1)])
  (check $r0 'two-plus-two))
(let* ([$r0 5])
```

[1] The names of these functions are primed because they are internal functions that use an efficient representation of lists: lists as functions.

```
  (expect $r0 'five))
```

If I put this code into file `plustest.scm`, I get the exact same results from my UFT/SVM system as I do from the `vscheme` interpreter:

```
> uft kn-vo plustest.scm | svm
Check-expect failed: expected two-plus-two to evaluate to 5, but it's 4.
The only test failed.
> vscheme < plustest.scm
Check-expect failed: expected two-plus-two to evaluate to 5, but it's 4.
The only test failed.
```

(12) *Complete your code generator.* Finish the cases for your two code-generation functions `toReg` and `forEffect`. You should be able to implement every syntactic form except function call. (You won't implement `toReturn` until module 8; I've provided a placeholder that just runs the function body for side effect.)

(13) *Test every syntactic form.* Create file `kntest.scm` with test cases for your code generator. The file should contain a sequence of Scheme expressions, in K-normal form, with hardware register names. These expressions should eventually call primitives `check` and `expect`. And there should be a distinct `check` and `expect` for every syntactic form in your definition of type `KNormalForm.exp`. As an example, the pair in step (11) tests application of a VM primitive (the form `@(`x_1`,…,`x_n`)`).

Ideally, every test successfully compiles to virtual object code and all the tests pass. But as long as the input is valid K-normal form, as verifed by `uft kn-kn`, and every syntactic form is tested, you will get full credit for the work.

Primitives

(14) *Make the UFT and SVM agree on primitives.* I won't ask you to test primitives just yet, because the testing will be so much easier in module 9, when you can test them by writing and compiling `check-expect` with function calls. But between now and module 9 is two weeks worth of distraction, so this is the time to ensure that the UFT and SVM agree on a set of primitive operations.

As needed, modify both the SVM and the UFT:

- Confirm that every UFT primitive in file `primitives.sml` has an entry in the SVM's instruction table (file `instructions.c`) and that the entry has the opcode that the UFT is going to emit. If there is a UFT primitive that is not in the SVM's instruction table, add it to the instruction table.

  Pro tip: If your SVM uses different names for any primitives, you don't have to change *any* existing code. Just add a new entry to the

SVM's instruction table, using your old binary opcode and the new name in the virtual object code. Multiple names for a single binary opcode can coexist.

Take notes on anything you have to add or change.

- Confirm that every instruction in SVM's instruction table (file `instructions.c`) meets one of these three criteria:

  - The instruction implements a UFT primitive of the same name, and the primitive is found in file `primitives.sml`.
  - The instruction implements a UFT primitive, but under a different name than the one used in file `primitives.sml`.
  - The instruction is used only for control flow (e.g., `if` and `goto`).

  If there is an entry in the SVM's instruction table that does not meet any of these criteria, *add it as a UFT primitive:*

  - Give it the proper arity.
  - Mark whether the instruction sets a register or is executed for a side effect.[2]

  Take notes on anything you have to add or change.

What and how to submit

(15) On Monday, *submit the homework.* In the `src/uft` directory you'll find a file `SUBMIT.06`. 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 hw06 src
```

(16) 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`:

[2] If you have created an instruction that is used *both* to set a register and *also* for an important side effect, the UFT does not know how to generate code for it. The instruction should not be added to the UFT; it can be used only in assembly language.

```
provide cs106 reflection06 REFLECTION
```

Reading in depth

Occasionally I'll suggest reading that may enrich your view of programming-language implementation or functional programming. This week I suggest John Hughes's classic 1986 paper on the implementation of lists as functions, which we are using in `codegen.sml`. This paper builds on Tony Hoare's foundational work on abstract data types (with abstraction functions), shows how the lists-as-functions representation can be used to calculate `revapp` from a naïve reversal function, and even reports on the result of a couple of experiments. All in three pages! (John is a super genius and is also a coauthor of QuickCheck.)

Learning outcomes

Outcomes available for points

Learning outcomes available for project points:

1. *Error monad.* You can use the error monad effectively as an applicative functor.
2. *Function composition.* You understand how to construct lists by function composition.
3. *Meaning of primitives.* You understand why the UFT defines two classes of primitives.
4. *Common invariants.* You can identify invariants that are common to K-normal form and assembly language.
5. *Unique invariant.* You can justify the translation of `let` into assignment by appealing to a precondition that incoming K-normal form must satisfy.
6. *Unchecked run-time errors.* You can use unchecked run-time errors to simplify your compiler.
7. *Systematic testing.* You know how to test every syntactic form of K-normal form.
8. *Consistent primitives.* Your SVM and UFT have a consistent view of primitive operations.
9. *Language extensions.* You can say how the system you have built so far would be extended to support new language features.

Learning outcomes available for depth points:

10. *Invariants and preconditions [1 point].* You have written an ML function that checks the precondition in outcome 5, thereby confirming that it is

OK to translate `let` into assignment. Your `forEffect` function checks the condition before calling `forEffect'`.

11. *Intraprocedural control operators [2 points].* You add `break` and `continue` to K-normal form and to everything upstream of it, and you translate them into `goto` instructions. (The vScheme parser already has hooks for `break` and `continue`; you just need to add the abstract syntax.)

12. *A-normal form [3 points].* A-normal form (Flanagan *et al* 13) is like K-normal form, but with more restrictions: in an expression of the form `let x = e in e'`, expression `e` may not be a `let`, `if`, or `while` form. Define a variant of A-normal form suitable for the UFT, generate code from it, and in a future module, convert first-order Scheme to it (either directly from first-order Scheme or indirectly from K-normal form).

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, submit a `mapx` that handles all cases and typechecks without ever mentioning either of the two value constructors `Error.OK` and `Error.ERROR`. (Using function `Error.succeed` is good and necessary.) Moreover, achieve this outcome without using the monadic bind operation `>>=`. (Using the `<$>` and `<*>` functions is encouraged.)

2. To claim this point, identify a line number in file `codegen.sml` where you wrote code that uses function composition to build a list of instructions, and explain how the code would look different if it instead used standard list operations like `::`, `@`, and `List.concat`.

3. To claim this point, identify the two classes of primitives, show how they are treated differently in your code generator (with line numbers), and give an example of what could go wrong if the two classes of primitives were indistinguishable in the UFT.

4. To claim this point, identify at least one invariant that is common to both K-normal form and assembly language, but *not* to Scheme source code.

5. The translation equations in the translation handout preserve semantics only if the input code satisfies a precondition. To claim this point, specify the precondition. Your specification must appeal solely to properties of K-normal form; it must not mention assembly code or translation.

6. The terms “unchecked run-time error” (Modula-3 specification and others), “undefined behavior” (C specification and others), and “getting stuck” (operational semantics) all mean the same thing. This concept recurs in too many contexts to be an accident. To claim this point, identify one specific place in your code (file name and line number) where your compiler exploits this concept, and explain how things would change if you were

required to *check* for this error (at either compile time or run time, your option).

7. To claim this point, ensure that your Monday submission includes test file `kntest.scm` with these properties:

   - It is accepted as valid K-normal form by the UFT checker `uft kn-kn`.
   - Every syntactic form of K-normal form occurs in a position that affects the result of primitives `check` and `expect`.

   The point is for knowing how to test—you can earn it even if the tests don't pass.

8. Every primitive in file `src/uft/primitives.sml` should have a corresponding entry in the instruction table in file `src/svm/instructions.c`. And except for control-flow instructions, every entry in the instruction table should correspond to a primitive in file `primitives.sml`.

   To claim this point, submit the files with these properties, plus one of the following two forms of supporting evidence:

   - If you had to add or change anything to make the primitives consistent, submit your notes from step (14).
   - If you didn't have to add or change anything in step (14), explain whether you owe this happy outcome to good luck or good management. Justify your answer.

9. To claim this point, answer three questions about *each* of three language extensions that might be added to vScheme. The questions are:

   A. To implement the feature, which languages (in addition to the source language) will have to change?

   B. What translation is best qualified to implement the new feature?

   C. Roughly how will it work?

   The extensions are:

   - vScheme is extended with Lisp's `defconstant` form, which defines a name that afterward acts like a literal value (a.k.a. a "compile-time constant"). Such a constant behaves a lot like an enumeration literal in C, or like a constant named with `#define`. (Attempts to assign to or call a defined constant can be treated as calls to the `error` primitive.)
   - vScheme is extended with Perl's `(`*`command`*` unless `*`condition`*`)`, which evaluates expression *`command`*, *unless* the *`condition`* is satisfied, in which case it does nothing and returns `nil`.

- vScheme is extended with two new forms of expression, `(break)` and `(continue)`, which respectively exit or restart the innermost loop in which they appear—just like the corresponding statements in C.