Module 12: Pattern Matching

Introduction

This week you'll translate pattern matching into efficient decision trees.

- *What am I doing?*
  - Implement `case` expressions by compiling each `case` expression to a decision tree. Each internal node of the decision tree will execute in almost constant time.[1]
- *Why am I doing it?*
  - Pattern matching one of the most prized features of the ML family of languages. You'll learn a simple algorithm that produces very efficient code in almost all cases.
- *How?*
  - Before lab, you'll do a ton of reading. You'll also resolve merge conflicts and add two new syntactic forms to your KNF.
  - In lab, you'll implement two key computations: a function that accounts for knowledge discovered during pattern matching, and a function that generates K-normal form from a decision tree.
  - After lab, you'll write the rest of the match compiler, and you'll implement the SVM instructions needed to run pattern matching.
  - At the end of the week, you'll submit a translator that includes pass `es-vo`. Your system (translator plus VM) should be able to run ML code as described in Chapter 8 of *Programming Languages: Build, Prove, and Compare*.
- *What should I expect?*
  - This module has not been debugged. There may be rough spots.
  - Like `lambda`, the `case` feature touches almost every aspect of the code. And match compilation is a bit more involved than closure conversion, so I expect the module might take a bit more effort than module 10.

[1] A constant number of VM instructions.

But unlike closure conversion, the algorithms aren't mind-blowing. That may help.

To help you manage the many parts of the code that must be touched, I have broken things down into *small* steps. The core steps are steps (8), (9), (18), and maybe (27). The other steps are subsidiary.

The module step by step

Before lab: Constructed data and match compilation

(1) *Case expressions and pattern matching in ML.* Look through the opening of Chapter 8 of *Programming Languages: Build, Prove, and Compare*, through the middle of page 465 end of section 8.1 (9 pages). The concepts are familiar and the section is mostly examples, so you need not read in depth—instead, read to pick up the terminology and to get comfortable with the concrete syntax.

Also read section 8.2.2 (pages 469 and 470), which explains how value constructors are written. Value constructors in eScheme use the same rules.

(2) *Semantics of pattern matching.* In the same book, read section 8.2.1 (pages 468 and 469), which describe the semantics of pattern matching, informally. (A formal semantics is shown in section 8.8.1, but that semantics describes an algorithm so different from ours that the section is probably not useful.)

(3) *Constructed data.* Read the first three sections of the handout on constructed data. Stop when you finish the section on constructed data at run time; you won't need to read about VM instructions before lab.

(4) *Get new code and repair your UFT.* Use `git pull` to update your UFT. **Expect merge conflicts.** Possibilities include:

- In `opcode.h`, new opcodes `GotoVcon` and `IfVconMatch` may conflict.
- New functions in `asmutil.sml` may conflict.
- A new primitive in `primitives.sml` may conflict.
- New cases in `disambiguate.sml` may conflict, especially for `APPLY`.
- In `uft.sml` a new language `ES` may conflict.

In order to be able to compile code during lab, you'll have to patch your UFT:

- File `asm.sml` defines a new form of assembly language: `GOTO_VCON`. That form has to be handled. For now, it can be handled by code that calls `Impossible.exercise`.

  - In file `assembler.sml`, handle the `GOTO_VCON` form in your `fold` function and in your `labelElim` function. Call `Impossible.exercise`. If you used ML's `case`, this code might be useful:

    ```
    | A.GOTO_VCON _ => Impossible.exercise "GOTO_VCON"
    ```

  - In file `asmparse.sml`, handle the `GOTO_VCON` form in your unparser. Also call `Impossible.exercise`.

- Extend your K-normalizer (`knormalize.sml`) to recognize patterns `C.CONSTRUCTED _` and `C.CASE _` and to respond to them with calls to `Impossible.exercise`.

  ```
  | C.CONSTRUCTED _ => Impossible.exercise "K-normalize data construction"
  | C.CASE _ => Impossible.exercise "K-normalize case expression"
  ```

- In `closure-convert.sml`, update your free-variable analysis to handle the `CONSTRUCTED` and `CASE` forms. The analyses are provide in file `case.sml`, so you need only pass in your existing function:

  ```
  | free (X.CASE c) = Case.free free c
  | free (X.CONSTRUCTED c) = Constructed.free free c
  ```

- Also in `closure-convert.sml`, extend your closure conversion to convert `CONSTRUCTED` and `CASE` forms. The conversion is entirely structural; my extensions look like this:

  ```
  | exp (X.CASE c) = C.CASE (Case.map exp c)
  | exp (X.CONSTRUCTED c) = C.CONSTRUCTED (Constructed.map exp c)
  ```

- Add these definitions to the top of your `KNormalize` structure:

  ```
  structure MC = MatchCompiler(type register = int
                               fun regString r = "$r" ^ Int.toString r
                                  )

  structure MV = MatchViz(structure Tree = MC)
  val vizTree = MV.viz (WppScheme.expString o CSUtil.embedExp)
  ```

Ensure that your UFT compiles.

(5) *Extend K-normal form.* Dive into your `knf.sml` and extend your K-normal form with two new syntactic forms. Each form has an operational semantics that is described informally in a comment.

```
| BLOCK of 'a list
     (* allocate a block and initialize each slot with the
        corresponding register *)

| SWITCH_VCON of 'a * ((Pattern.vcon * int) * 'a exp) list * 'a exp
   (* given SWITCH_VCON (r, choices, other), if the value in register
```

```
        r matches any (vcon, k) pair, then evaluate the corresponding
          expression, otherwise evaluate the other expression *)
```

Update your KNF embedding (`knembed.sml`) to handle each of these forms with a call to `Impossible.exercise`.

```
| exp (K.BLOCK _) = Impossible.exercise "embed K.BLOCK"
| exp (K.SWITCH_VCON _) = Impossible.exercise "embed K.SWITCH_VCON"
```

And update your code generator (`codegen.sml`) to handle each of these forms with a call to `Impossible.exercise`.

```
| K.BLOCK _ => Impossible.exercise "codegen K.BLOCK"
| K.SWITCH_VCON _ => Impossible.exercise "codegen K.SWITCH_VCON"
```

And finally update `knrename.sml` to rename the two forms. With luck you may be able to use this code:

```
| K.BLOCK xs => K.BLOCK <$> errorList (map f xs)
| K.SWITCH_VCON (x, choices, fallthru) =>
    let fun choice (pat, e) = pair pat <$> mapx f e
    in  curry3 K.SWITCH_VCON <$> f x
                             <*> Error.mapList choice choices
                             <*> mapx f fallthru
    end
```

Ensure that your UFT compiles.

(6) *Match-compilation reading.* Pick up the handout on match compilation and the match-compilation paper by Kevin Scott and Norman Ramsey. Read enough so you have an idea what is going on with the `tree` type and with Figure 8. One of the core operations of Figure 8—refining constraints of the form (π, p)—is the first half of lab.

Lab: Refinement ~~and K-normalization~~

(7) *Refine constraints.* In file `match-compiler.sml`, implement function `refineConstraint`. Its type is

```
val refineConstraint :
 register -> labeled_constructor -> constraint -> constraint list compatibility
```

Aim for a clean function that type checks.

There is a full explanation in the match-compilation handout, but to refine a given constraint (π', p') using labeled constructor C/n at path π, you can use these guidelines:

- ~~If the pair (π', p') is unrelated to what's happening at path π, it~~ If $\pi' \neq \pi$ then π' and π refer to different locations. So the knowledge that C/n is seen at π doesn't affect constraint (π', p'); (π', p') just gets refined into the compatible, singleton list $[(\pi', p')]$.

- If $\pi' = \pi$ and the constructor's name and arity match, that is, if the pair (π', p') is $(\pi, C(p_1, \ldots, p_n))$, then it gets replaced by the list of compatible pairs $[(\pi.1, p_1), \ldots, (\pi.n, p_n)]$.[2]
- If $\pi' = \pi$ and p' is any other constructor application, then constraint (π', p') is not compatible with seeing C/n at path π, and `refineConstraint` returns INCOMPATIBLE.

Plan on spending about ~~half the lab~~ the whole lab on `refineConstraint` and `refineFrontier` here. ~~When you finish, either continue with function `refineFrontier` or move on to step (8).~~

After lab

K-normalization of case expressions

(8) *K-normalize case expressions using decision trees.* Return to your K-normalizer, and replace the call to `Impossible.exercise` with the following template (the template calls the match compiler and uses the resulting decision tree to generate K-normal form):

```
| F.CASE (e, choices) =>
  (... normalize e into a register, using the following continuation ...) (fn t =>
     let fun treeGen = Impossible.exercise "treeGen"
         val _ = treeGen : regset -> F.exp MC.tree -> reg K.exp
         val A' = ... available registers minus t ...
     in  treeGen A' (vizTree (MC.decisionTree (t, choices)))
     end)
```

The `CASE` form is defined in structure `Case` in file `case.sml`; variable `e` has type `F.exp` and variable choices has type `(Pattern.pattern * F.exp) list`.

Function `vizTree` acts like an identity function, except it may also write a visualiation to disk.

Define function `treeGen`.

Match compilation

(9) *Match compilation: Refinement.* Return to the match-compilation handout and the paper by Scott and Ramsey. In `match-compiler.sml`, if you did not finish functions `refineConstraint` and `refineFrontier` in lab, finish them now.

Now implement function `decisionTree`. The `TEST` and `MATCH` nodes are described in the paper. When your match compiler produces a node of the form `LET_CHILD ((`r`, `i`), `k`)`, you define continuation k. The continuation expects a new, temporary register, and it updates all the frontiers,

[2] The paper writes paths backwards, as $1.\pi$ and so on. We must have been insane.

substituting the new register for the old path `CHILD (`r`, `i`)`. Perform the substitution using function `forPath`.

Ensure that your UFT compiles.

(10) *If working locally, install Graphviz.* If you work on your own local machine, install the Graphviz package with its `dot` command. If you are on Halligan, `dot` is already installed in `/usr/sup/bin`.

(11) *Visualize results of match compilation.* The example in Figure 6 of the paper can reproduced using this eScheme code:

```
(define figure-6 (arg)
  (case arg
    [(C1 C2 C3) 'one]
    [(C1 x C4)  'two]
    [(C1 x C5)  'three]
    [_          'four]))

(check-expect (figure-6 (C1 3 C4)) 'two)
```

Place the code in file `fig6.scm`, and run

```
$ case-viz fig6.scm
```

Your UFT should crash with an uncaught `Impossible` exception, but before it does, it should write a visualization to file `fig6.dot`. The `case-viz` script should process the visualization into PDF and then open the PDF.

Notice that several leaves are duplicates. This duplication can be eliminated during code generation (for depth points).

K-normalization and embedding

(12) *K-normalization of constructed data.* K-normalize the `F.CONSTRUCTED` form. In the general case, the value constructor is turned into a string literal, and the whole lot go into a `K.BLOCK` form. But in order to continue using your legacy SVM representations of lists and Booleans, your K-normalizer should implement these special cases:

```
| F.CONSTRUCTED ("#t", []) => ... literal (BOOL true) ...
| F.CONSTRUCTED ("#f", []) => ... literal (BOOL false) ...
| F.CONSTRUCTED ("cons", [x, y]) => ... application of P.cons ...
| F.CONSTRUCTED ("'()", [])      => ... literal EMPTYLIST ...
```

(13) *Embed the* `SWITCH_VCON` *and* `BLOCK` *forms.* In file `knembed.sml`, update your K-normal form embedding to handle the new syntactic forms. Feel free to adapt my code:

```
| embed (K.BLOCK xs) = S.APPLY (S.VAR "block", map S.VAR xs)
| embed (K.SWITCH_VCON (x, choices, e)) =
```

```
let val lastQa = [(S.LITERAL (S.BOOLV true), embed e)]
    fun isCon (vcon, arity) =
          S.APPLY (S.VAR "matches-vcon-arity?",
            [S.VAR x, S.LITERAL (S.SYM vcon), (S.LITERAL o S.INT) arity])
    fun qa (c, e) = (isCon c, embed e)
in  S.COND (map qa choices @ lastQa)
end
```

(14) *Test your match compiler using the embedding.* Compile your UFT, then test the match compiler and embedding:

```
$ uft es-kn fig6.scm | vscheme
```

Fix a bug in closures and blocks.

(15) *Correct a bug that applies to* `CLOSURE` *and* `BLOCK` *forms.*

Assuming `n` is a local variable with value 99, the following code should set `n` to `(C1 99 100)` but instead it sets the second slot of `n` to point to `n` itself:[3]

```
(set n (C1 n 100)))
```

The same bug manifests with closures. Your UFT has the bug if it fails this test:

```
(define id (x) x)
(define simplify (x)
  (lambda ()
    (id (lambda () x))))

(check-expect (((simplify 99))) 99)
```

In both cases the difficulty is the same: the K-normalizer thinks that `BLOCK` and `CLOSURE` can be implemented in a single, atomic VM instruction, which implies that a `BLOCK` or `CLOSURE` value can be placed in a register that is also one of its arguments—just like an `add` instruction.

We can solve this problem during K-normalization by allocating a fresh register and assigning the BLOCK or CLOSURE form to it, which is done by the K-normal form `x := e`, where `e` is the normal form of the `BLOCK` or `CLOSURE`. The K-normalizer needs to allocate register `x`; I recommend this function, which does it in continuation-passing style:

```
val inLocalVar : regset -> (reg -> exp) -> exp =
  fn rs => fn k => let val t = smallest rs in K.SETLOCAL (t, k t) end
```

My new code for K-normalizing a closure now looks something like this:

[3] A value like this sends my printer into a tizzy. I might try to fix it later in the week.

```
| F.CLOSURE (lambda, captured) =>
    if null captured then
      K.FUNCODE (funcode lambda)
    else
      inLocalVar rs (fn t =>
         nbRegs bindAnyReg (rs -- t) captured (fn ys =>
           K.CLOSURE (funcode lambda, ys)))
```

Something similar needs to be done for the general case of K-normalizing `CONSTRUCTED` into `BLOCK`.

Code generation

(16) *Generate code for the* `BLOCK` *form.* In `codegen.sml`, the `K.BLOCK` form has to be handled only in function `toReg`. The code should be very similar to the code you generate for the `K.CLOSURE` form. In `forEffect`, a `K.BLOCK` generates no code, just like a `K.CLOSURE`. And `toReturn` can delegate to `toReg`.

(17) *Determine representations for value constructors in case expressions, including legacy value constructors.* In step (12) you wrote special-case code for the *introduction* form of constructed data. The special-case code enables your SVM to continue to use its special-case, efficient representations of constructed Booleans and lists. You also need to enable the Scheme `case` expression (the *elimination* form) to recognize these special-case representations.

To eliminate Boolean and list values using special-case code, you will use a trick that also allows integer literals to act like constructed data in patterns. The trick is to convert each *labeled* constructor to an appropriate literal value that can represent the constructor in assembly code and object code. The conversion can be done by this function:

```
val conLiteral : labeled_constructor -> A.literal
```

When value constructors `#t` and `#f` are used with arity 0, they should be converted to Boolean literals. Value constructor `'()`, when used with arity 0, should be converted to an empty-list literal. And when a constructor used with arity 0 has a name made entirely of digits, it should be converted to an integer literal. Every other constructor should be converted to a string literal.

Pitfall: Function `Int.fromString` can't be trusted: it works even if only a *prefix* of the string given is an integer. Validate the name using the predicate `CharVector.all Char.isDigit`.

In file `codegen.sml`, implement function `conLiteral`.[4]

[4] Although the elimination form is handled in `codegen.sml`, the introduction form is handled in `knormalize.sml`. This structure just about kills me, but I don't have time to fix it.

(18) *Generate code for* `SWITCH_VCON`. Like the `IFX` form, the `SWITCH_VCON` form has subexpressions whose code generation depends on context:

- If a `SWITCH_VCON` form puts its result in a register, each subexpression puts its result in that same register.
- If a `SWITCH_VCON` is executed for side effect, each subexpression is executed for side effect.
- And if a `SWITCH_VCON` is in tail position (returned), each subexpression is in tail position.

To avoid writing three copies of the same code, I defined an auxiliary function:

```
switchVcon :  (exp -> code)
           -> code
           -> reg * (labeled_constructor * exp) list * exp
           -> code
```

where `exp` is KNF and `code` is a Hughes list of assembly instructions.

Calling `switchVcon gen finish (r, choices, fallthru)` generates code as follows:

- Each subexpression is translated using `gen`, and its translation is followed by (a copy of) `finish`.
- Every subexpression except `fallthru` is preceded by the definition of a fresh label.
- The code returned by `switchVcon` starts with an `AssemblyCode.GOTO_VCON` form, which contains a jump table. That form is followed by the translation of `fallthru`. And that translation is followed by the translations of all the other expressions in any order, each of which is preceded by the definition of its corresponding label.

During the construction of the jump table. each `labeled_constructor` is converted to an `ObjectCode.literal` and an arity. To compute the literal, use the function `conLiteral` that you defined in step (17).

Implement `switchVcon` and use it in `toReg`, `forEffect`, and `toReturn`. In the first two cases, the `finish` parameter should contain an unconditional jump to an exit label. For `toReturn`, the `finish` parameter should be empty.

Assembly and disassembly

(19) *Unparse the* `GOTO_VCON` *form.* In file `asmparse.sml`, update your unparser to handle the `GOTO_VCON` form. Like `LOADFUNC`, `GOTO_VCON` should unparse to multiple lines. That's important for preserving indentation.

My code might help; it probably won't compile in your UFT, but it can probably be adapted:

```
| unparse (A.GOTO_VCON (r, choices) :: instructions) =
    let fun choice (v, arity, lbl) =
        spaceSep ["  case", lit v, "(" ^ int arity ^ "):", "goto", lbl]
   in  spaceSep ["switch", reg r, "{"] :: map choice choices @ "}" ::
        unparse instructions
    end
```

(20) *Test code generation.* Using your new unparser, check on the results of your code generator from step (18). For example, when I run mine on `fig6.scm`, I get these results:

```
$ uft es-vs fig6.scm
r0 := function (1 arguments) {
  switch r1 {
    case "C1" (2): goto L1
  }
  r0 := "four"
  return r0
  L1:
  r2 := block r1(1)
  switch r2 {
    case "C2" (0): goto L2
  }
  r3 := block r1(2)
  switch r3 {
    case "C5" (0): goto L3
    case "C4" (0): goto L4
  }
  r0 := "four"
  return r0
  L3:
  r0 := "three"
  return r0
  ... more where this came from ...
```

(21) *Assemble the* `GOTO_VCON` *form.* In file `assembler.sml`, update your `fold` function to account for the number of instructions emitted for a `GOTO_VCON` form: it is one plus twice the number of choices.

Update your `labelElim` function to assemble the `GOTO_VCON` form. It should generate a `GOTO_VCON` instruction from the `ObjectCode` module, followed by a pair of instructions for each choice: one `REGSLIT` instruction with the `if-vcon-match` opcode, and one `GOTO` instruction.

This code offers ample opportunity for off-by-one errors.

(22) *Prepare to disassemble the new instructions in* `svm-dis`. In your SVM directory, add new instructions to file `instructions.c`. You are welcome to copy my entries:

```
{ "mkblock", MkBlock, parseR2U8, "rX := block[rY,Z]" },
{ "getblockslot", GetBlockSlot, parseR2U8, "rX := block rY.Z" },
{ "setblockslot", SetBlockSlot, parseR2U8, "block rX.Z := rY" },
{ "goto-vcon", GotoVcon, parseR1U8, "goto-vcon rX [Y slots]" },
{ "if-vcon-match", IfVconMatch, parseU8LIT, "if vcon == LIT/X then" },
```

Add opcodes `MkBlock`, `GetBlockSlot`, `SetBlockSlot`, `GotoVcon`, and `IfVconMatch` to your `opcode.h` file.

Run `make`, which should rebuild both `svm` and `svm-dis`.

(23) *Inspect object code.* Run your object code through the disassembler and confirm that it looks as you expect. Pay special attention to PC-relative branches. My object code for `fig6.scm`, which I believe now respects the operational semantics,[5] disassembles like this:

```
$ uft es-vo fig6.scm | svm-dis -pc
  0: r0 := function 0x564793125d00
  1: global `figure-6` := r0
      ...
-----------------------------------
;    function at literal 15
  0: goto-vcon r1 [1 slots]
  1: if vcon == "C1"/2 then
  2: goto $PC + 2
  3: r0 := "four"
  4: return r0
  5: r2 := block r1.1
  6: goto-vcon r2 [1 slots]
  7: if vcon == "C2"/0 then
  8: goto $PC + 12
  9: r3 := block r1.2
 10: goto-vcon r3 [2 slots]
 11: if vcon == "C5"/0 then
 12: goto $PC + 4
 13: if vcon == "C4"/0 then
 14: goto $PC + 4
 15: r0 := "four"
 16: return r0
 17: r0 := "three"
 18: return r0
 19: r0 := "two"
 20: return r0
 21: r3 := block r1.2
 22: goto-vcon r3 [3 slots]
 23: if vcon == "C5"/0 then
```

[5] The branch is relative to the address *following* the `goto` instruction.

```
24: goto $PC + 6
25: if vcon == "C4"/0 then
26: goto $PC + 6
27: if vcon == "C3"/0 then
28: goto $PC + 6
29: r0 := "four"
30: return r0
31: r0 := "three"
32: return r0
33: r0 := "two"
34: return r0
35: r0 := "one"
36: return r0
37: halt
```

New SVM instructions

(24) *Put symbol* `cons` *where your* `vmrun` *can access it quickly.* To keep working with legacy representations of lists, your `vmrun` function needs access to the literal symbol `cons`. I made my life easy by adding this symbol's location to the VM state. I add the symbol in my `newstate` function:

```
vm->cons_sym_slot = literal_slot(vm, mkStringValue(Vmstring_newc("cons")));
```

My `vmrun` then places the symbol in a local variable:

```
Value cons_symbol = vm->literals[vm->cons_sym_slot]; // restore me after GC
```

(25) *Implement the block instructions.* In `vmrun.c`, implement instructions `mkblock`, `getblockslot`, and `setblockslot`. Functions `mkblock` and `setblockslot` can be modeled on existing closure instructions. But `getblockslot` needs to handle *both* blocks and cons cells. Mine makes a cons cell masquerade as a block with three slots:

```
case GetBlockSlot: {
    struct VMBlock *block = RY.block;
    if (RY.tag == ConsCell) {
        switch (Z) {
        case 0: RX = cons_symbol; break;
        case 1: case 2: RX = block->slots[Z - 1]; break;
        default: runerror("A cons cell does not have a slot %d", Z);
        }
    } else {
        assert(RY.tag == Block);
        RX = getrecord(vm, block->slots, block->nslots, Z);
    }
    break;
    }
```

This embarrassing masquerade can be eliminated for depth points.

(26) *Add a debugging flag.* At the top of your `vmrun` function, add this definition:

```
const char *dump_case = svmdebug_value("case");
```

(27) *Implement computed goto.* The `GotoVcon` instruction must first compute a labeled constructor from the value of the scrutinee, then search the jump table for a corresponding entry. The labeled constructor comprises a constructor value and an arity. It is computed by these rules:

- If the value of the scrutinee is a block b, then the constructor's value is b`->slots[0]` and its arity is b`->nslots - 1`.
- If the value of the scrutinee is a cons cell, then the constructor's value is the string `cons` and its arity is 2.
- If the value of the scrutinee is any other value v, then the constructor's value is v and its arity is zero.

Given the labeled constructor's value and arity, the `GotoVcon` instruction searches for a jump-table entry with a matching value and arity. Since each jump-table entry is coded as a pair of VM instructions, you can extract the expected arity in the X field of the first instruction. The value is found in the literal table at index YZ of that same instruction.

The next instruction specifies the address to which to transfer control if the value and arity match. That next instruction is a `Goto`, and it codes for the target address in the usual way (the 24-bit signed offset in the XYZ field, relative to the address immediately following the `Goto`).

Implement `GotoVcon`.

If `dump_case` is not `NULL`, show the value of the scrutinee, and show the constructor and arity you have computed for that value. Also, for each `IfVconMatch` instruction that is checked for a match, show the constructor and arity that are being checked.

Caution: Off-by-one errors can bite you here. If you run into trouble, try printing out target addresses when variable `dump_case` is not `NULL`. If you continue to have trouble, you could consider instead interpreting each `IfVconMatch` opcode as an independent instruction, as specified in the operational semantics. Once that's working, you can move the logic into `GotoVcon`, which will eliminate the extra interpretive overhead.

Testing

(28) *Recommendations for testing.* The eScheme parser understands ML code, and I've added predefined ML functions to file `predefs.es` in the

`src/vscheme` directory. You can read more about ML in chapter 8 of *Programming Languages: Build, Prove, and Compare.*

Alert: Because the ML interpreter has types, it can and does eta-expand every bare value constructor. But eScheme does not expand any bare value constructors; heedless of possible type definitions, eScheme will cheerfully use any value constructor at arity 0.

For testing, I recommend three phases:

1. As an initial test, the `split` function below does not depend on any predefined functions. It splits a list into two parts of equal size. (It's part of a merge sort.)

   ```
   (define split (xs half other-half)
                 (case xs
                    ['() (PAIR half other-half)]
                    [(cons y ys)
                        (split ys other-half (cons y half))]))

   (check-expect (split '(1 2) '() '()) (PAIR '(1) '(2)))
   (check-expect (split '(1 2 3 4) '() '()) (PAIR '(3 1) '(4 2)))
   ```

   You can run code with script `run-es-with-predef`. Or more likely,

   ```
   env SVMDEBUG=decode,case run-es-with-predef split.es
   ```

2. Next, look over the predefined functions. Quite a few are defined using `case` forms. Use some of these functions to Write some simple `check-expect` tests.

3. As an integration test, try the 2D-tree code described in the Supplement to *Build, Prove, and Compare* in section E.2, which starts on page S125. This code implements an elegant geometric algorithm which searches a map for the known point that is nearest to an arbitrary query point. To provide a test case, I've translated the `gis.uml` file to `gis.es`, which required only eta-expanding a few value constructors. I also provide the file `ne-city-halls.es`, which includes a `check-expect` test for the query shown on page S134. That code exercises value constructors and `case` expressions on non-list, non-Boolean data.

What and how to submit

(29) On Monday, *submit the homework.* In the `src/uft` directory you'll find a file `SUBMIT.12`. 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 hw12 src
```

or if you keep an additional `tests` directory,

```
provide cs106 hw12 src tests
```

(30) 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 reflection12 REFLECTION
```

Reading in depth

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

- A decision tree with jump tables at the leaves is used not only for efficient matching of constructed data, but even for `switch` statements that operate on integers. John Hennessy and Noah Mendelsohn (1982) describe an algorithm that is used in many compilers for C and C++. Yes, that Noah Mendelsohn.
- Robert Bernstein (1985) describes code generation for case statements in a production compiler. He pays significant attention to machine instructions, including a now-famous trick for implementing a signed range test with a single conditional branch.
- Luc Maranget has published a whole series of papers on pattern matching. He really nailed the decision-tree approach in a 2008 paper on producing good decision trees. In my opinion that paper is definitive.

Learning outcomes

Outcomes available for points

Learning outcomes available for project points:

1. *Craft.* You can extend both UFT and SVM with a new feature.
2. *Nested patterns.* You understand the capabilities of nested patterns.

3. *Pattern visualization.* You can use visualizations to analyze the result of compiling different `case` expressions.

4. *Inexhaustive pattern matches.* Your match compiler grasps the possibility that in any given `case` expression, there may be no match.

5. *Static types and pattern matching.* You understand the consequences of match compilation in a world where there are no static types.

6. *Representations of constructed data, part I.* You understand the run-time consequences of using a uniform representation for constructed data.

7. *Representations of constructed data, part II.* You understand the compile-time consequences of using a uniform representation for constructed data.

8. *Address computation.* Your assembler accurately computes the target of every branch in a computed `goto`.

9. *Efficient computed* `goto`. Your SVM interprets `GotoVcon` and its jump table together as one big instruction.

Learning outcomes available for depth points:

10. *Static types [2 points].* Suppose your UFT had static types, so any time it saw a list it would know the only possibilities were empty or nonempty. How would you exploit this knowledge to improve run-time performance?

    - What would you change about SVM instructions?
    - How would you change the code generated by the UFT?

11. *Code generation using a directed acyclic graph (DAG) [4 points].* In general, the match compiler produces a decision tree whose leaves contain multiple copies of values of type `'a`. Our simple K-normalizer and code generator will likely turn multiple copies of source expressions into multiple copies of assembly-code fragments. But by the time we reach assembly language, the `GOTO_VCON` form branches to a label—there is no need to copy the code. To earn these depth points, refactor your UFT so that it copies only labels, not code.

    This challenge could be approached in several ways:

    - Define a new KNF form that can represent a DAG using (scoped) labels. Match-compile with a unique label for each choice. If two leaves of the decision tree use the same label **with the same environment**, they can share code.
    - Define a decision-tree form in KNF. During code generation, use the standard algorithm to convert the tree to a DAG.
    - Leave the data structures exactly as they are. But during code generation, recover the decision tree from a nest of `SWITCH_VCON` and `LET` nodes, then use the standard algorithm to convert the tree to a DAG.

12. *Alternate run-time representations [7 points].* On an alternate Git branch, change your SVM to eliminate the `Boolean`, `Emptylist`, and `ConsCell` tags from `value.h`. Also eliminate all the code associated with these tags, and every related SVM instruction except `if`. In eScheme, reimplement `car`, `cdr`, `pair?`, and `null?` using `case` expressions. Then report on your experience:

    - How does this change simplify the SVM? In your answer, include measurements of the code savings. (Lines of code will be sufficient.)
    - How does this change affect run time? Devise a suitable benchmark that uses lists and Booleans, and quantify the cost (or savings).

13. *Native list operations [2 points].* In the SVM, I find it vaguely embarrassing that my `GetBlockSlot` instruction has to enable a cons cell to masquerade as a block. Eliminate the masquerade:

    - Extend the `path` type in the match compiler to include two new forms `CAR of register` and `CDR of register`.
    - Change the `LET_CHILD` form to take a path instead of a `register * int`, and change the code generator so it can emit `car` and `cdr` primitives for those paths. You'll need equations like this one:

      $$r[\![\texttt{LET\_CHILD (CAR } r\texttt{, } k\texttt{)}]\!] = \texttt{let } t \texttt{ = car(}r\texttt{) in } k\ t,$$

    - Remove the special case from the implementation of `GetBlockSlot` in `vmrun.c`.

How to claim the project points

1. To claim this point, submit a source tree in which `make` successfully builds a UFT and an SVM, when run in directories `src/uft` and `src/svm` respectively.

2. To claim this point, take two *useful* `case` expressions with nested patterns, either from your UFT or your 105 homework. In a file `nested.scm`, define functions `f` and `g` that contain eScheme `case` expressions that use the same patterns in the same order. Then define equivalent functions `f-unnested` and `g-unnested`, each of which performs the computation using nested `case` expressions—*not* nested patterns. Do not use `if`.

   The equivalence I wish to see is *observational* equivalence: two functions are equivalent if no program can tell the difference.

3. To claim this point, use the `case-viz` script to visualize functions `f` and `f-unnested` from the previous point. The visualization of `f-unnested` will contain two or more decision trees. Splice them together either by drawing pictures or by doctoring the `nested.dot` file. Compare the trees for the two files:

- Do you expect nested `case` expressions to amount to the same decision tree as one `case` expression with nested patterns? Or just an equivalent decision tree? Why or why not?
- Explain whether the trees produced by your match compiler do or do not meet your expectations.

4. To claim this point, submit a file `nomatch.scm` which evaluates a `case` expression in which no choice matches. Your submission earns the point if the code compiles and runs with your UFT and SVM, producing a suitable run-time error message.

5. In ML, the type of each scrutinee is known, and that type is associated with a finite, known set of value constructors. Explain the consequences for match compilation. Illustrate your explanation with the standard `map` function: how would it be compiled for eScheme, and how would it be compiled for ML? What is the key difference?

6. In steps (12) and (17), I've encouraged you to special-case representations of Booleans and lists. But in the ML interpreter, Booleans and lists are predefined, not primitive, and they use the same representations as all other constructed data. To claim this point, suppose you wished to implement the same plan in your own system. Identify the code that would have to change in `vmrun.c`, `value.h`, and `value.c`.

7. In steps (12) and (17), I've encouraged you to special-case representations of Booleans and lists. But in the ML interpreter, Booleans and lists are predefined, not primitive, and they use the same representations as all other constructed data. To claim this point, suppose you wished to implement the same plan in your own system. Identify the code that would have to change in your K-normalizer, your code generator, and your disambiguator.

8. To claim this point, copy `fig6.scm` into file `fig6-extended.scm`, and add `check-expect` tests that call function `figure-6` expecting results `'one`, `'three`, and `'four`. Your system earns the point if the code compiles and all the tests pass.

9. To claim this point, submit an implementation of `GotoVcon` that searches the jump table and chooses a target address, without returning to the main interpreter loop.