1 Language Primer

citations for this history are fragmented across the internet

Factor is a rather young language created by Slava Pestov in September of 2003. Its first incarnation targeted the Java Virtual Machine (JVM) as an embedded scripting language for a game. As such, its feature set was minimal. Factor has since evolved into a general-purpose programming language, gaining new features and redesigning old ones as necessary for larger programs. Today's implementation sports an extensive standard library and has moved away from the JVM in favor of native code generation. In this section, we cover the basic syntax and semantics of Factor for those unfamiliar with the language. This should be just enough to understand the later material in this thesis. More thorough documentation can be found via Factor's website, `http://factorcode.org`.

1.1 Combinators

Quotations, introduced in **??**, form the basis of both control flow and data flow in Factor. Not only are they the equivalent of anonymous functions, but the stack model also makes them syntactically lightweight enough to serve as blocks akin to the code between curly braces in C or Java. Higher-order words that make use of quotations on the stack are called *combinators*. It's simple to express familiar conditional logic and loops using combinators, as we'll show in Section 1.1.1. In the presence of explicit data flow via stack operations, even more patterns arise that can be abstracted away. Section 1.1.2 explores how we can use combinators to express otherwise convoluted stack-shuffling logic more succinctly.

1.1.1 Control Flow

```
5 even? [ "even" print ] [ "odd" print ] if

{ } empty? [ "empty" print ] [ "full" print ] if

100 [ "isn't f" print ] [ "is f" print ] if
```

Listing 1: Conditional evaluation in Factor

The most primitive form of control flow in typical programming languages is, of course, the **if** statement, and the same holds true for Factor. The only difference is that Factor's **if** isn't syntactically significant—it's just another word, albeit implemented as a primitive. For the moment, it will do to think of **if** as having the stack effect `( ? true false -- )`. The third element from the top of the stack is a condition, and it's followed by two quotations. The first quotation (second element from the top of the stack) is called if the condition is true, and the second quotation (the top of the stack) is called if the condition is false. Specifically, **f** is a special object in Factor for falsity. It is a singleton object—the sole instance of the **f** class—and is the only false value in the entire language. Any other object is necessarily boolean true. For a canonical boolean, there is the **t** object, but its truth value exists only because it is not **f**. Basic **if** use is shown in Listing 1. vref

The first example will print "odd", the second "empty", and the third "isn't f". All of them leave nothing on the stack.

```
: example1 ( x -- 0/x-1 )
  dup even? [ drop 0 ] [ 1 - ] if ;

: example2 ( x y -- x+y/x-y )
  2dup mod 0 = [ + ] [ - ] if ;

: example3 ( x y -- x+y/x )
  dup odd? [ + ] [ drop ] if ;
```

Listing 2: `if`'s stack effect varies

extra space after `?` with minted

However, the simplified stack effect for `if` is quite restrictive. `( ? true false -- )` intuitively means that both the `true` and `false` quotations can't take any inputs or produce any outputs—that their effects are `( -- )`. We'd like to loosen this restriction, but per `??`, Factor must know the stack height after the `if` call. We could give `if` the effect `( x ? true false -- y )`, so that the two quotations could each have the stack effect `( x -- y )`. This would work for the `example1` word in Listing 2, yet it's just as restrictive. For instance, the `example2` word would need `if` to have the effect `( x y ? true false -- z )`, since each branch has the effect `( x y -- z )`. Furthermore, the quotations might even have different effects, but still leave the overall stack height balanced. Only one item is left on the stack after a call to `example3` regardless, even though the two quotations have different stack effects: `+` has the effect `( x y -- z )`, while `drop` has the effect `( x -- )`.

In reality, there are infinitely many correct stack effects for `if`. Factor has a special notation for such *row-polymorphic* stack effects. If a token in a stack effect begins with two dots, like `..a` or `..b`, it is a *row variable*. If either side of a stack effect begins with a row variable, it represents any number inputs/outputs. Thus, we could give `if` the stack effect

```
( ..a ? true false -- ..b )
```

to indicate that there may be any number of inputs below the condition on the stack, and any number of outputs will be present after the call to `if`. Note that these numbers aren't necessarily equal, which is why we use distinct row variables in this case. However, this still isn't quite enough to capture the stack height requirements. It doesn't communicate that `true` and `false` must affect the stack in the same ways. For this, we can use the notation `quot: ( stack -- effect )`, giving quotations a nested stack effect. Using the same names for row variables in both the "inner" and "outer" stack effects will refer to the same number of inputs or outputs. Thus, our final (correct) stack effect for `if` is

```
( ..a ? true: ( ..a -- ..b ) false: ( ..a -- ..b ) -- ..b )
```

This tells us that the `true` quotation and the `false` quotation will each create the same relative change in stack height as `if` does overall.

```
{ "Lorem" "ipsum" "dolor" } [ print ] each

0 { 1 2 3 } [ + ] each

10 iota [ number>string print ] each

3 [ "Ho!" print ] times

[ t ] [ "Infinite loop!" print ] while

[ f ] [ "Executed once!" print ] do while
```

Listing 3: Loops in Factor

Though **if** is necessarily a language primitive, other control flow constructs are defined in Factor itself. It's simple to write combinators for iteration and looping as tail-recursive words that invoke quotations. Listing 3 showcases some common looping patterns. The most basic yet versatile word is **each**. Its stack effect is

```
( ... seq quot: ( ... x -- ... ) -- ... )
```

Each element `x` of the sequence `seq` will be passed to `quot`, which may use any of the underlying stack elements. Here, unlike **if**, we enforce that the input stack height is exactly the same as the output (since we use the same row variable). Otherwise, depending on the number of elements in `seq`, we might dig arbitrarily deep into the stack or flood it with a varying number of values. The first use of **each** in Listing 3 is balanced, as the quotation has the effect `( str -- )` and no additional items were on the stack to begin with. Essentially, it's equivalent to *"Lorem"* **print** *"ipsum"* **print** *"dolor"* **print**. On the other hand, the quotation in the second example has the stack effect `( total n -- total+n )`. This is still balanced, since there is one additional item below the sequence on the stack (namely `0`), and one element is left by the end (the sum of the sequence elements). So, this example is the same as `0 1 + 2 + 3 +`.

Any instance of the extensive **sequence** mixin will work with **each**, making it very flexible. The third example in Listing 3 shows **iota**, which is used here to create a *virtual* sequence of integers from 0 to 9 (inclusive). No actual sequence is allocated, merely an object that behaves like a sequence. In Factor, it's common practice to use **iota** and **each** in favor of repetitive C-like **for** loops.

Of course, we sometimes don't need the induction variable in loops. That is, we just want to execute a body of code a certain number of times. For these cases, there's the **times** combinator, with the stack effect

```
( ... n quot: ( ... -- ... ) -- ... )
```

This is similar to **each**, except that `n` is a number (so we needn't use **iota**) and the quotation doesn't expect an extra argument (i.e., a sequence element). Therefore, the example in Listing 3 is equivalent to *"Ho!"* **print** *"Ho!"* **print** *"Ho!"* **print**.

Naturally, Factor also has the **while** combinator, whose stack effect is

```
( ..a pred: ( ..a -- ..b ? ) body: ( ..b -- ..a ) -- ..b )
```

The row variables are a bit messy, but it works as you'd expected: the `pred` quotation is invoked on each iteration to determine whether `body` should be called. The **do** word is a handy modifier for **while** that simply executes the body of the loop once before leaving **while** to test the precondition as per usual. Thus, the last example in Listing 3 on the previous page executes the body once, despite the condition being immediately false.

```
{ 1 2 3 } [ 1 + ] map

{ 1 2 3 4 5 } [ even? ] filter

{ 1 2 3 } 0 [ + ] reduce
```

Listing 4: Higher-order functions in Factor

In the preceding combinators, quotations were used like blocks of code. But really, they're the same as anonymous functions from other languages. As such, Factor borrows classic tools from functional languages, like **map** and **filter**, as shown in Listing 4. **map** is like **each**, except that the quotation should produce a single output. Each such output is collected up into a new sequence of the same class as the input sequence. Here, the example produces `{ 2 3 4 }`. **filter** selects only those elements from the sequence for which the quotation returns a true value. Thus, the **filter** in Listing 4 outputs `{ 2 4 }`. Even **reduce** is in Factor, also known as a *left fold*. An initial element is iteratively updated by pushing a value from the sequence and invoking the quotation. In fact, **reduce** is defined as **swapd each**, where **swapd** is a shuffler word with the stack effect `( x y z -- y x z )`. Thus, the example in Listing 4 is the same as `0 { 1 2 3 } [ + ]` **each**, as in Listing 3 on the previous page.

These are just some of the control flow combinators defined in Factor. Several variants exist that meld stack shuffling with control flow, or can be used to shorten common patterns like empty false branches. An entire list is beyond the scope of our discussion, but the ones we've studied should give a solid view of what standard conditional execution, iteration, and looping looks like in a stack-based language.

1.1.2 Data Flow