1 Language Primer

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. 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. 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

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**. For these situations, 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 give each row variable different names in this case. However, this still isn't quite enough to capture the stack height requirements. Specifically, 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. We can use the same names for row variables in both the "inner" and "outer" stack effects in order to 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. Since quotations are basically just anonymous functions, looping combinators just need to recursively invoke them. Listing 3 showcases some common looping patterns. The most basic, yet most versatile, is **each**, whose stack effect is

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

Here, unlike with **if**, we enforce that the input stack height is exactly the same as the output (since we use the same row variable). So, each element `x` of `seq` will be passed to `quot`, which may use any of the underlying stack elements. As long as the quotation doesn't push or pop any excess elements, the effect of **each** is balanced. Thus, the first example in Listing 3 is equivalent to

```
"Lorem" print "ipsum" print "dolor" print
```

while the second example is the same as

```
0 1 + 2 + 3 +
```

Both leave the stack the same height as at the start: the first with no items, the second with one (the total sum of the sequence elements).

Any instance of the extensive **sequence** mixin will work with **each**, making it very flexible. The third example shows **iota**, which is used 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. So, 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 counted 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 (like a sequence element) on the stack. Therefore, the example in Listing 3 on the previous page executes *"Ho!"* **print** three times.

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

$$(..a \text{ pred: } (..a -- ..b \text{ ? }) \text{ body: } (..b -- ..a) -- ..b)$$

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.

1.1.2 Data Flow