# 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. 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

```
: 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

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 leave nothing on the stack.

However, the simplified stack effect above is quite restrictive. It doesn't allow us to push any outputs. But to allow that, per **??**, Factor requires the stack height to be known at each point of the program—including after the **`if`** call. How do we know what the stack effect should be? Consider the words defined in Listing 2. In the `example1` word, both quotations passed to **`if`** have the stack effect `( a -- b )`. So, no matter which branch is taken, the stack height remains the same. `example2` shows another case of balanced input quotations, but unlike `example1`, they each have the effect `( a b -- c )`. For that matter, the third word has a balanced overall stack effect, since only one item is left on the stack after a call to `example3` regardless. Yet the two quotations have different stack effects: `+` has the effect `( a b -- c )`, while **`drop`** has the effect `( a -- )`.

### 1.1.2 Data Flow