1 Introduction

Compilers translate programs written in a source language (e.g., Java) into semantically equivalent programs in some target language (e.g., assembly code). They let us make our source language arbitrarily abstract so we can write programs in ways that humans understand while letting the computer execute programs in ways that machines understand. In a perfect world, such translation would be straightforward. Reality, however, is unforgiving. Straightforward compilation results in clunky target code that performs a lot of redundant computations. To produce efficient code, we must rely on less-than-straightforward methods. Typical compilers go through a stage of *optimization*, whereby a number of semantics-preserving transformations are applied to an *intermediate representation* of the source code. These then (hopefully) produce a more efficient version of said representation. Optimizers tend to work in *phases*, applying specific transformations during any given phase.

Global value numbering (GVN) is such an analysis performed by many highly-optimizing compilers. Its roots run deep through both the theoretical and the practical. Using the results of this analysis, the compiler can identify expressions in the source code that produce the same value—not just by lexical comparison (i.e., variables having the same name), but by proving equivalences between what's actually computed at runtime. These expressions can then be simplified by further algorithms for redundancy elimination. This is the very essence of most compiler optimizations: avoid redundant computation, giving us code that runs as quickly as possible while still following what the programmer originally wrote.

High-level, dynamic languages tend to suffer from efficiency issues: they're often interpreted rather than compiled, and perform no heavy optimization of the source code. However, the Factor language (`http://factorcode.org`) fills an intriguing design niche, as it's very high-level yet still fully compiled. It's still young, though, so its compiler craves all the improvements it can get. In particular, while Factor currently has a *local* value numbering analysis, it is inferior to GVN in several significant ways.

In this thesis, we explore the implementation and use of GVN in improving the strength of optimizations in Factor. Because Factor is a young and relatively unknown language, Section 2 provides a short tutorial, laying a foundation for understanding the changes. **??** describes the overall architecture of the Factor compiler, highlighting where the exact contributions of this thesis fit in. Finally, **??** goes into detail about the existing and new value numbering passes, closing with a look at the results achieved and directions for future work.

All the code for the GVN phase was written atop Factor version 0.94, and a copy of it can be found in the appendix. In the unlikely event that you want to cite this thesis, you may use the following BibTeX entry:

ref

make sure it's single-page

```
@mastersthesis{vondrak:11,
  author = {Alex Vondrak},
  title  = {Global Value Numbering in Factor},
  school = {California Polytechnic State University, Pomona},
  month  = sep,
  year   = {2011},
}
```

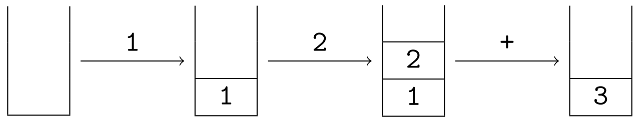

Figure 1: Visualizing stack-based calculation

2 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`.

2.1 Stack-Based Languages

Like Reverse Polish Notation (RPN) calculators, Factor's evaluation model uses a global stack upon which operands are pushed before operators are called. This naturally facilitates postfix notation, in which operators are written after their operands. For example, instead of `1 + 2`, we write `1 2 +`. Figure 1 shows how `1 2 +` works conceptually:

- `1` is pushed onto the stack
- `2` is pushed onto the stack
- `+` is called, so two values are popped from the stack, added, and the result (`3`) is pushed back onto the stack

Other stack-based programming languages include Forth, Joy, Cat, and PostScript.

cite

cite

cite

cite

The strength of this model is its simplicity. Evaluation essentially goes left-to-right: literals (like `1` and `2`) are pushed onto the stack, and operators (like `+`) perform some computation using values currently on the stack. This "flatness" makes parsing easier, since we don't need complex grammars with subtle ambiguities and precedence issues. Rather, we basically just scan left-to-right for tokens separated by whitespace. In the Forth tradition, functions are called *words* since they're made up of any contiguous non-whitespace characters. This also lends to the term *vocabulary* instead of "module" or "library". In Factor, the parser works as follows.

- If the current character is a double-quote (`"`), try to parse ahead for a string literal.
- Otherwise, scan ahead for a single token.

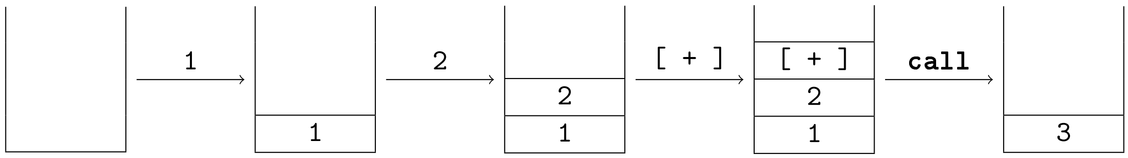

Figure 2: Quotations

  - If the token is the name of a *parsing word*, that word is invoked with the parser's current state.
  - If the token is the name of an ordinary (i.e., non-parsing) word, that word is added to the parse tree.
  - Otherwise, try to parse the token as a numeric literal.

Parsing words serve as hooks into the parser, letting Factor users extend the syntax dynamically. For instance, instead of having special knowledge of comments built into the parser, the parsing word `!` scans forward for a newline and discards any characters read (adding nothing to the parse tree).

Similarly, there are parsing words for what might otherwise be hard-coded syntax for data structure literals. Many act as sided delimeters: the parsing word for the left-delimiter will parse ahead until it reaches the right-delimiter, using whatever was read in between to add objects to the data structure. For example, `{ 1 2 3 }` denotes an array of three numbers. Note the deliberate spaces in between the tokens, so that the delimeters are themselves distinct words. In `{␣1␣2␣3␣}` (with spaces as marked), the parsing word `{` parses objects until it reaches `}`, collecting the results into an array. The `{` word would not be called if not for that space, whereas `{1␣2␣3}` parses as the word `{1`, the number `2`, and the word `3}`—not an array. Further, since the left-delimeter words parse recursively, such literals can be nested, contain comments, etc. Other literals include the following.

```
V{ 1 2 3 }                         ! vector
B{ 1 2 3 }                         ! byte array
BV{ 1 2 3 }                        ! byte vector
HS{ 1 2 3 }                        ! hash set
H{ { key1 val1 } { key2 val2 } } ! hash table
```

Listing 1: Data structure literals in Factor

A particularly important set of parsing words in Factor are the square brackets, `[` and `]`. Any code in between such brackets is collected up into a special sequence called a *quotation*. Essentially, it's a snippet of code whose execution is suppressed. The code inside a quotation can then be run with the **`call`** word. Quotations are like anonymous functions in other languages, but the stack model makes them conceptually simpler, since we don't have to worry about variable binding and the like. Consider a small example like `1 2 [ + ]` **`call`**. You can think of **`call`** working by "erasing" the brackets around a quotation, so this example behaves just like `1 2 +`. Figure 2 shows its evaluation: instead of adding the numbers immediately, `+` is placed in a quotation, which is pushed to the stack. The quotation is then invoked by **`call`**, so `+` pops and adds the two numbers and pushes the result onto the stack. We'll show how quotations are used in Section 2.5.

2.2 Stack Effects

Everything else about Factor follows from the stack-based structure outlined in Section 2.1. Consecutive words transform the stack in discrete steps, thereby shaping a result. In a way, words are functions from stacks to stacks—from "before" to "after"—and whitespace is effectively function composition. Even literals (numbers, strings, arrays, quotations, etc.) can be thought of as functions that take in a stack and return that stack with an extra element pushed onto it.

With this in mind, Factor requires that the number of elements on the stack (the *stack height*) is known at each point of the program in order to ensure consistency. To this end, every word is associated with a *stack effect* declaration using a notation implemented by parsing words. In general, a stack effect declaration has the form

```
( input1 input2 ... -- output1 output2 ... )
```

where the parsing word `(` scans forward for the special token `--` to separate the two sides of the declaration, and then for the `)` token to end the declaration. The names of the intermediate tokens don't technically matter—only how many of them there are. However, names should be meaningful for clarity's sake. The number of tokens on the left side of the declaration (before the `--`) indicates the minimum stack height expected before executing the word. Given exactly this number of inputs, the number of tokens on the right side is the stack height after executing the word.

For instance, the stack effect of the `+` word is `( x y -- z )`, as it pops two numbers off the stack and pushes one number (their sum) onto the stack. This could be written any number of ways, though. `( x x -- x )`, `( number1 number2 -- sum )`, and `( m n -- m+n )` are all equally valid. Further, while the stack effect `( junk x y -- junk z )` has the same relative height change, this declaration would be wrong, since `+` might legitimately be called on only two inputs.

For the purposes of documentation, of course, the names in stack effects do matter. They correspond to elements of the stack from bottom-to-top. So, the rightmost value on either side of the declaration names the top element of the stack. We can see this in Figure 3 on the next page, which shows the effects of standard *stack shuffler* words. These words are used for basic data flow in Factor programs. For example, to discard the top element of the stack, we use the **`drop`** word, whose effect is simply `( x -- )`. To discard the element just below the top of the stack, we use **`nip`**, whose effect is `( x y -- y )`. This stack effect indicates that there are at least two elements on the stack before **`nip`** is called: the top element is `y`, and the next element is `x`. After calling the word, `x` is removed, leaving the original `y` still on top of the stack. Other shuffler words that remove data from the stack are **`2drop`** with the effect `( x y -- )`, **`3drop`** with the effect `( x y z -- )`, and **`2nip`** with the effect `( x y z -- z )`.

The next stack shufflers duplicate data. **`dup`** copies the top element of the stack, as indicated by its effect `( x -- x x )`. **`over`** has the effect `( x y -- x y x )`, which tells us that it expects at least two inputs: the top of the stack is `y`, and the next object is `x`. `x` is copied and pushed on top of the two original elements, sandwiching `y` between two `x`s. Other shuffler words that

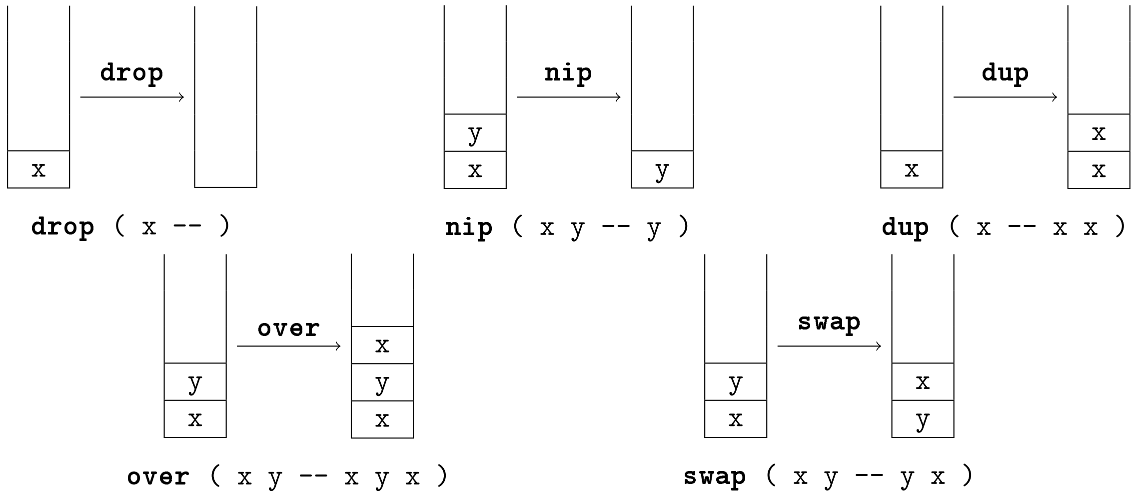

Figure 3: Stack shuffler words and their effects

duplicate data on the stack are **`2dup`** with the effect `( x y -- x y x y )`, **`3dup`** with the effect `( x y z -- x y z x y z )`, **`2over`** with the effect `( x y z -- x y z x y )`, and **`pick`** with the effect `( x y z -- x y z x )`.

True to the name **`swap`**, the final shuffler in Figure 3 permutes the top two elements of the stack, reversing their order. The stack effect `( x y -- y x )` indicates as much. The left side denotes that two inputs are on the stack (the top is `y`, the next is `x`), and the right side shows the outputs are swapped (the top element is `x` and the next is `y`). Factor has other words that permute elements deeper into the stack. However, their use is discouraged because it's harder for the programmer to mentally keep track of more than a couple items on the stack. We'll see how more complex data flow patterns are handled in Section 2.5.2.

2.3 Definitions

```
: hello-world ( -- )
  "Hello, world!" print ;
```

Listing 2: Hello World in Factor

```
: norm ( x y -- norm )
  dup * swap dup * + sqrt ;
```

Listing 3: The Euclidean norm, $\sqrt{x^2 + y^2}$

Using the basic syntax of stack effect declarations described in Section 2.2, we can now understand how to define words. Most words are defined with the parsing word `:`, which scans for a name, a stack effect, and then any words up until the **`;`** token, which together become the body of the definition. Thus, the classic example in Listing 2 defines a word named `hello-world` which

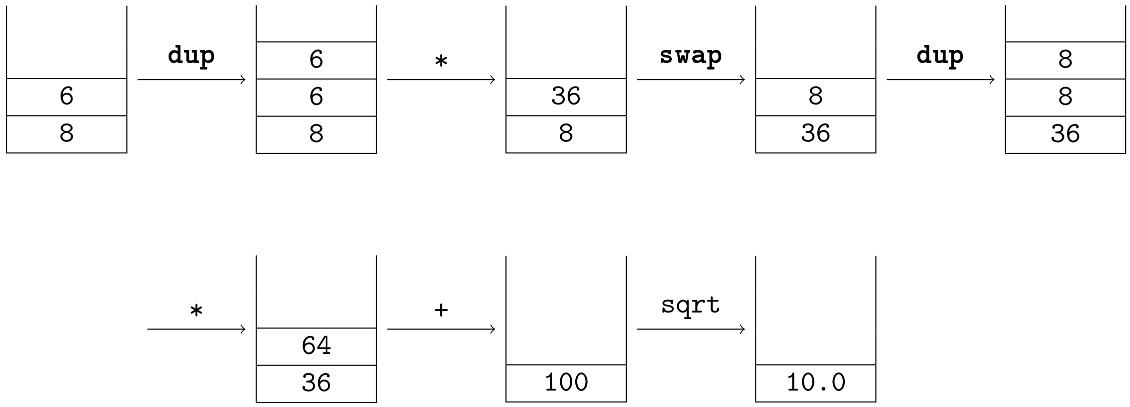

Figure 4: `norm` example

expects no inputs and pushes no outputs onto the stack. When called, this word will display the canonical greeting on standard output using the **`print`** word.

A slightly more interesting example is the `norm` word in Listing 3 on the preceding page. This squares each of the top two numbers on the stack, adds them, then takes the square root of the sum. Figure 4 shows this in action. By defining a word to perform these steps, we can replace virtually any instance of **`dup`** `*` **`swap dup`** `* + sqrt` in a program simply with `norm`. This is a deceptively important point. Data flow is made explicit via stack manipulation rather than being hidden in variable assignments, so repetitive patterns become painfully evident. This makes identifying, extracting, and replacing redundant code easy. Often, you can just copy a repetitive sequence of words into its own definition verbatim. This emphasis on "factoring" your code is what gives Factor its name.

```
: ^2 ( n -- n^2 )
  dup * ;

: norm ( x y -- norm )
  ^2 swap ^2 + sqrt ;
```

Listing 4: `norm` refactored

As a simple case in point, we see the subexpression **`dup`** `*` appears twice in the definition of `norm` in Listing 3 on the preceding page. We can easily factor that out into a new word and substitute it for the old expressions, as in Listing 4. By contrast, programs in more traditional languages are laden with variables and syntactic noise that require more work to refactor: identifying free variables, pulling out the right functions without causing finicky syntax errors, calling a new function with the right variables, etc. Though Factor's stack-based paradigm is atypical, it is part of a design philosophy that aims to facilitate readable code focusing on short, reusable definitions.

2.4 Object Orientation

You may have noticed that the examples in Section 2.3 did not use type declarations. While Factor is dynamically typed for the sake of simplicity, it does not do away with types altogether. In fact, Factor is object-oriented. However, its object system doesn't rely on classes possessing particular methods, as is common. Instead, it uses *generic words* with methods implemented for particular classes. To start, though, we must see how classes are defined.

2.4.1 Tuples

```
TUPLE: class
  slot-spec1 slot-spec2 slot-spec3 ... ;

TUPLE: subclass < superclass
  slot-spec1 slot-spec2 slot-spec3 ... ;
```

Listing 5: Basic tuple definition syntax

The central data type of Factor's object system is called a *tuple*, which is a class composed of named *slots*—like instance variables in other languages. Tuples are defined with the `TUPLE:` parsing word as shown in Listing 5. A class name is specified first; if it is followed by the `<` token and a superclass name, the tuple inherits the slots of the superclass. If no superclass is specified, the default is the **`tuple`** class.

Slots can be specified in several ways. The simplest is to just provide a single token, which is the name of the slot. This slot can then hold any type of object. Using the syntax `{ name class }`, a slot can be limited to hold only instances of a particular class, like **`integer`** or **`string`**. There are other forms of slot specifiers, which we will cover after some examples.

```
TUPLE: color ;

: <color> ( -- color )
  color new ;

TUPLE: rgb < color red green blue ;

: <rgb> ( r g b -- rgb )
  rgb boa ;
```

Listing 6: Simple tuple examples

Consider the two tuples defined in Listing 6. The first, `color`, has no slots. With every tuple, a class predicate is defined with the stack effect `( object -- ? )` whose name is the class suffixed by a question mark. Here, the word `color?` is defined, which pushes a boolean (in Factor, either **`t`** or **`f`**) indicating whether the top element of the stack is an instance of the `color` class. The second tuple, `rgb`, inherits from the `color` class. While this doesn't give `rgb` any different slots, it does mean that an instance of `rgb` will return **`t`** for `color?` due to the "is-a" relationship between subclass and superclass. The word `rgb?` is similarly defined.

Notice that the `rgb` tuple declares three slots named `red`, `green`, and `blue`. Since the slots' classes aren't declared, any sort of object can be stored in them. A set of methods are defined to manipulate an `rgb` instance's slots. Three *reader* words are defined (one for each slot), analogous to "getter" methods in other languages. Following the template for naming reader words, this example defines `red>>`, `green>>`, and `blue>>`. Each word has the stack effect `( object -- value )`, and extracts the value corresponding to the eponymous slot. Similarly, the *writer* words `red<<`, `green<<`, and `blue<<` each have the stack effect `( value object -- )`, and store values in the corresponding `rgb` slots destructively. To leave the modified `rgb` instance on the stack while setting slots, the *setter* words `>>red`, `>>green`, and `>>blue` are also defined, each with the stack effect `( object value -- object' )`. These words are defined in terms of writers. For instance, `>>red` is the same as **`over`** `red<<`, since **`over`** copies a reference to the tuple (i.e., it doesn't make a "deep" copy).

To construct an instance of a tuple, we can use either **`new`** or **`boa`**. **`new`** will not initialize any of the slots to a particular input value—all slots will default to Factor's canonical false value, **`f`**. **`new`** is used in Listing 6 on the preceding page to define `<color>` (by convention, the constructor for `foo` is named `<foo>`). First, we push the class `color` onto the stack (this word is also automatically defined by `TUPLE:`), then just call **`new`**, leaving a new instance on the stack. Since this particular tuple has no slots, using **`new`** makes sense. We might also use it to initialize a class, then use setter words to only assign a particular subset of slots' values.

However, we often want to initialize a tuple with values for each of its slots. For this, we have **`boa`**, which works similarly to **`new`**. This is used in the definition of `<rgb>` in Listing 6 on the previous page. The difference here is the additional inputs on the stack—one for each slot, in the order they're declared. That is, we're constructing the tuple **b**y **o**rder of **a**rguments, giving us the fun pun "**`boa`** constructor". So, `1 2 3 <rgb>` will construct an `rgb` instance with the `red` slot set to `1`, the `green` slot set to `2`, and the `blue` slot set to `3`.

```
TUPLE: email
    { from string }
    { to array }
    { cc array }
    { bcc array }
    { subject string }
    { content-type string initial: "text/plain" }
    { encoding word initial: utf8 }
    { body string } ;
```

Listing 7: Special slot specifiers

Now that we've seen the various words defined for tuples, we can explore more complex slot specifiers. Using the array-like syntax from before, slot specifiers may be marked with certain *attributes*—both with the class declared (like `{ name class attributes... }`) and without the class declared (as in `{ name attributes... }`). In particular, Factor recognizes two different attributes. If a slot marked `read-only`, the writer (and thus setter) for the slot will not be defined, so the slot cannot be altered. A slot may also provide an initial value using the syntax `initial: some-literal`. This will be the slot's value when instantiated with **`new`**.

For example, Listing 7 on the preceding page shows a tuple definition from Factor's `smtp` vocabulary that defines an `email` object. The `from` address, `subject`, and `body` must be instances of `string`, while `to`, `cc`, and `bcc` are **`array`**s of destination addresses. The `content-type` slot must also be a **`string`**, but if unspecified, it defaults to *`"text/plain"`*. The `encoding` must be a `word` (in Factor, even words are first-class objects), which by default is `utf8`, a word from the `io.encodings.utf8` vocabulary for a Unicode format.

2.4.2 Generics and Methods

Unlike more common object systems, we do not define individual methods that "belong" to particular tuples. In Factor, you define a method that specializes on a class for a particular generic word. That way, when the generic word is called, it dispatches on the class of the object, invoking the most specific method for the object.

Generic words are declared with the syntax **`GENERIC:`** `word-name ( stack -- effect )`. Words defined this way will then dispatch on the class of the top element of the stack (necessarily the rightmost input in the stack effect). To define a new method with which to control this dispatch, we use the syntax **`M: class`** `word-name definition...` **`;`**.

```
USING: bit-sets hash-sets sequences ;
IN: sets

MIXIN: set
INSTANCE: sequence set
INSTANCE: hash-set set
INSTANCE: bit-set set
```

Listing 8: Set instances

An accessible example of a generic word is in Factor's `sets` vocabulary. **`set`** is a *mixin* class—a union of other classes whose members may be extended by the user. We can see the standard definition in Listing 8. Note that the **`USING:`** form specifies vocabularies being used (like Java's **`import`**), and `IN:` specifies the vocabulary in which the definitions appear (like Java's **`package`**). We can see here that instances of the **`sequence`**, `hash-set`, and `bit-set` classes are all instances of **`set`**, so will respond **`t`** to the predicate `set?`. Similarly, **`sequence`** is a mixin class with many more members, including **`array`**, **`vector`**, and **`string`**.

Listing 9 on the following page shows the `cardinality` generic from Factor's `sets` vocabulary, along with its methods. This generic word takes a **`set`** instance from the top of the stack and pushes the number of elements it contains. For instance, if the top element is a `bit-set`, we extract its `table` slot and invoke another word, `bit-count`, on that. But if the top element is **`f`** (the canonical false/empty value), we know the cardinality is 0. For any **`sequence`**, we may offshore the work to a different generic, **`length`**, defined in the `sequences` vocabulary. The final method gives a default behavior for any other **`set`** instance, which simply uses `members` to obtain an equivalent **`sequence`** of set members, then calls **`length`**.

```
IN: sets
GENERIC: cardinality ( set -- n )

USING: accessors bit-sets math.bitwise sets ;
M: bit-set cardinality table>> bit-count ;

USING: kernel sets ;
M: f cardinality drop 0 ;

USING: accessors assocs hash-sets sets ;
M: hash-set cardinality table>> assoc-size ;

USING: sequences sets ;
M: sequence cardinality length ;

USING: sequences sets ;
M: set cardinality members length ;
```

Listing 9: Set cardinality using Factor's object system

By viewing a class as a set of all objects that respond positively to the class predicate, we may partially order classes with the subset relationship. Method dispatch will use this ordering when `cardinality` is called to select the most specific method for the object being dispatched upon. For instance, while no explicit method for **`array`** is defined, any instance of **`array`** is also an instance of **`sequence`**. In turn, every instance of **`sequence`** is also an instance of **`set`**. We have methods that dispatch on both **`set`** and **`sequence`**, but the latter is more specific, so that is the method invoked. If we define our own class, `foo`, and declare it as an instance of **`set`** but not as an instance of **`sequence`**, then the **`set`** method of `cardinality` will be invoked. Sometimes resolving the precedence gets more complicated, but these edge-cases are beyond the scope of our discussion.

2.5 Combinators

Quotations, introduced in Section 2.1, 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 2.5.1. In the presence of explicit data flow via stack operations, even more patterns arise that can be abstracted away. Section 2.5.2 explores how we can use combinators to express otherwise convoluted stack-shuffling logic more succinctly.

2.5.1 Control Flow

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

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

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

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

Listing 10: Conditional evaluation in Factor

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

vref

```
: 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 11: **`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 Section 2.2, 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 11, 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 12: 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 12 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 12 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 12 on the preceding page 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 12 on the previous page 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 12 on the preceding 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 13: 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 13. **`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 13 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 13 is the same as `0 { 1 2 3 } [ + ]` **`each`**, as in Listing 12 on the preceding 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.

2.5.2 Data Flow

While avoiding variables and additional syntax makes it easier to refactor code, keeping mental track of the stack can be taxing. If we need to manipulate more than the top few elements of the stack, code gets harder to read and write. Since the flow of data is made explicit via stack shufflers, we actually wind up with redundant patterns of data flow that we otherwise couldn't identify. In Factor, there are several combinators that clean up common stack-shuffling logic, making code easier to understand.

```
: without-dip1 ( x y -- x+1 y )
  swap 1 + swap ;

: with-dip1 ( x y -- x+1 y )
  [ 1 + ] dip ;

: without-dip2 ( x y z -- x-y z )
  2over - swapd nip swapd nip swap ;

: with-dip2 ( x y z -- x-y z )
  [ - ] dip ;

: without-keep1 ( x -- x+1 x )
  dup 1 + swap ;

: with-keep1 ( x -- x+1 x )
  [ 1 + ] keep ;

: without-keep2 ( x y -- x-y y )
  swap over - swap ;

: with-keep2 ( x y -- x-y y )
  [ - ] keep ;
```

Listing 14: Preserving combinators

The first combinators we'll look at are **dip** and **keep**. These are used to preserve elements of the stack. When working with several values, sometimes we don't want to use all of them at quite the same time. Using **drop** and the like wouldn't help, as we'd lose the data altogether. Rather, we want to retain certain stack elements, do a computation, then restore them. For an uncompelling but illustrative example, suppose we have two values on the stack, but we want to increment the second element from the top. `without-dip1` in Listing 14 on the previous page shows one strategy, where we shuffle the top element away with **`swap`**, perform the computation, then **`swap`** the top back to its original place. A cleaner way is to call **`dip`** on a quotation, which will execute that quotation just under the top of the stack, as in `with-dip1`. While the stack shuffling in `without-dip1` isn't terribly complicated, it doesn't convey our meaning very well. Shuffling the top element out of the way becomes increasingly difficult with more complex computations. In `without-dip2`, we want to call `-` on the two elements below the top. For lack of a more robust stack shuffler, we use **`2over`** to isolate the two values so we can call `-`. The rest of the word consists of shuffling to get rid of excess values on the stack. It's also worth noting that **`swapd`** is a deprecated word in Factor, since its use starts making code harder to reason about. Alternatively, we could dream up a more complex stack shuffler with exactly the stack effect we wanted in this situation. But this solution doesn't scale: what if we had to calculate something that required more inputs or produced more outputs? Clearly, **`dip`** provides a cleaner alternative in `with-dip2`.

`keep` provides a way to hold onto the top element of the stack, but still use it to perform a computation. In general, `[ ... ]` **`keep`** is equivalent to **`dup`** `[ ... ]` **`dip`**. Thus, the current top of the stack remains on top after the use of **`keep`**, but the quotation is still invoked with that value. In `with-keep1` in Listing 14 on the preceding page, we want to increment the top, but stash the result below. Again, this logic isn't terribly complicated, though `with-keep1` does away with the shuffling. `without-keep2` shows a messier example where a simple **`dup`** will not save us, as we're using more than just the top element in the call to `-`. Rather, three of the four words in the definition are dedicated to rearranging the stack in just the right way, obscuring the call to `-` that we really want to focus on. On the other hand, `with-keep2` places the subtraction word front-and-center in its own quotation, while **`keep`** does the work of retaining the top of the stack.

```
TUPLE: coord x y ;

: without-bi ( coord -- norm )
  [ x>> sq ] keep y>> sq + sqrt ;

: with-bi ( coord -- norm )
  [ x>> sq ] [ y>> sq ] bi + sqrt ;

: without-tri ( x -- x+1 x+2 x+3 )
  [ 1 + ] keep [ 2 + ] keep 3 + ;

: with-tri ( x -- x+1 x+2 x+3 )
  [ 1 + ] [ 2 + ] [ 3 + ] tri ;
```

Listing 15: Cleave combinators

The next set of combinators apply multiple quotations to a single value. The most general form of these so-called *cleave* combinators is the word **`cleave`**, which takes an array of quotations as input, and calls each one in turn on the top element of the stack. Of course, for only a couple of quotations, wrapping them in an array literal becomes cumbersome. The word **`bi`** exists for the two-quotation case, and **`tri`** for the three quotations. Cleave combinators are often used to extract multiple slots from a tuple. Listing 15 on the previous page shows such a case in the `with-bi` word, which improves upon using just **`keep`** in the `without-bi` word. In general, a series of **`keep`**s like `[ a ] keep [ b ] keep c` is the same as `{ [ a ] [ b ] [ c ] } cleave`, which is more readable. We can see this in action in the difference between `without-tri` and `with-tri` in Listing 15 on the preceding page. In cases where we need to apply multiple quotations to a set of values instead of just a single one, there are also the variants **`2cleave`** and **`3cleave`** (and the corresponding **`2bi`**, **`2tri`**, **`3bi`**, and **`3tri`**), which apply the quotations to the top two and three elements of the stack, respectively.

```
: without-bi* ( str1 str2 -- str1' str2' )
  [ >upper ] dip >lower ;

: with-bi* ( str1 str2 -- str1' str2' )
  [ >upper ] [ >lower ] bi* ;

: without-tri* ( x y z -- x+1 y+2 z+3 )
  [ [ 1 + ] dip 2 + ] dip 3 + ;

: with-tri* ( x y z -- x+1 y+2 z+3 )
  [ 1 + ] [ 2 + ] [ 3 + ] tri* ;
```

Listing 16: Spread combinators

To apply multiple quotations to multiple values, Factor has *spread* combinators. Whereas cleave combinators abstract away repeated instances of **`keep`**, spread combinators replace nested calls to **`dip`**. The archetypical combinator, **`spread`**, takes an array of quotations, like **`cleave`**. However, instead of applying each one to the top element of the stack, each one corresponds to a separate element of the stack. Thus, `{ [ a ] [ b ] } spread` invokes `b` on the top element, and `a` on the element beneath the top. Much like **`cleave`**, there are shorthand words for the two- and three-quotation cases. These are suffixed with asterisks to indicate the spread variants, so we have **`bi*`** and **`tri*`**. In Listing 16, the `without-bi*` word shows the simple **`dip`** pattern that **`bi*`** encapsulates. Here, we're converting the string `str1` (the second element from the top) into uppercase and `str2` (the top element) to lowercase. In `with-bi*`, the `>upper` and `>lower` words are seen first, uninterrupted by an extra word, making the code easier to read. More compelling is the way that **`tri*`** replaces the **`dip`**s that can be seen in `without-tri*`. In comparison, `with-tri*` is less nested and easier to comprehend at first glance. While there are **`2bi*`** and **`2tri*`** variants that spread quotations to two values apiece on the stack, they are uncommon in practice.

Finally, *apply* combinators invoke a single quotation on multiple stack entries in turn. While there is a generalized word, it's more common to use the corresponding shorthands. Here, they are suffixed with at-signs, so **`bi@`** applies a quotation to each of the top two stack values, and **`tri@`** to each of the top three. This way, rather than duplicate code for each time we want to call a word, we need only specify it once. This is demonstrated clearly in Listing 17 on the following page. In `without-bi@`, we see that the quotation `[ sq ]` (for squaring numbers) appears twice for the call to **`bi*`**. In general, we can replace spread combinators whose quotations are all the same with a single quotation and an apply combinator. Thus, `with-bi@` cuts down on the duplicated `[ sq ]` in `without-bi@`. Similarly, we can replace the three repeated quotations passed

```
: without-bi@ ( x y -- norm )
  [ sq ] [ sq ] bi* + sqrt ;

: with-bi@ ( x y -- norm )
  [ sq ] bi@ + sqrt ;

: without-tri@ ( x y z -- x+1 y+1 z+1 )
  [ 1 + ] [ 1 + ] [ 1 + ] tri* ;

: with-tri* ( x y z -- x+1 y+1 z+1 )
  [ 1 + ] tri@ ;
```

Listing 17: Apply combinators

to **`tri*`** in `without-tri@` with a single instance passed to **`tri@`** in `with-tri@`. Like other data flow combinators, we have the numbered variants. **`2bi@`** has the stack effect `( w x y z quot -- )`, where `quot` expects two inputs, and is thus applied to `w` and `x` first, then to `y` and `z`. Similarly, **`2tri@`** applies the quotation to the top six objects of the stack in groups of two. Like their spread counterparts, they are not used very much.

Some wrap-up that isn't completely lame.