GLOBAL VALUE NUMBERING IN FACTOR

A Thesis

Presented to the

Faculty of

California State Polytechnic University, Pomona

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

In

Computer Science

By

Alex Vondrak

2011

SIGNATURE PAGE

THESIS:	GLOBAL VALUE NUMBERING IN FACTOR
AUTHOR:	Alex Vondrak
DATE SUBMITTED:	Summer 2011
	Computer Science
Dr. Craig Rich Thesis Committee Chair Computer Science	____________________________________
Dr. Daisy Sang Computer Science	____________________________________
Dr. Amar Raheja Computer Science	____________________________________

Acknowledgments

To Lindsay—he is my rock

Abstract

Compilers translate code in one programming language into semantically equivalent code in another language—canonically from a high-level language to low-level machine primitives. Generally, the further removed a language's abstractions get from those of a computer, the harder it gets to compile code into an efficient representation. What isn't redundant in the source language may map to repetitive target instructions that waste time recomputing results. To combat this, compilers try to optimize away redundancies by looking for values that are provably equivalent when the program is run.

This thesis explores the theory and implementation of a particularly aggressive analysis called global value numbering in a particularly high-level language called Factor. Factor is a stack-based, dynamically-typed, object-oriented language born in late 2003. A baby among languages (now at version 0.94), its compiler craves all the optimizations it can get. By altering the existing local value numbering pass, redundancies can be identified and eliminated across entire programs, rather than isolated regions of code. This induces speedups as high as 45% across the majority of benchmarks. The results from these comparatively simple changes hold much promise for future improvements in making Factor programs more efficient.

Table of Contents

List of Figures

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 a phase 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., comparing variable names), 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 the current Factor version (as of this writing, 0.94) 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, Chapter 2 provides a short tutorial, laying a foundation for understanding the changes. Chapter 3 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.

In the unlikely event that you want to cite this thesis, you may use the following BibTeX entry:

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

2 Language Primer

Factor is a rather young language created by Slava Pestov in September 2003 [*Factor* 2010]. Its first incarnation was an embedded scripting language for a game that targeted the Java Virtual Machine (JVM). 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 chapter, 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 on the following page 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 [American National Standards Institute and Computer and Business Equipment Manufacturers Association 1994], Cat [Diggins 2007], and PostScript [Adobe Systems Incorporated 1999].

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,

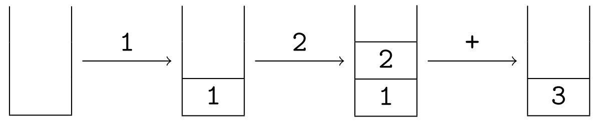

Figure 1: Visualizing stack-based calculation

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.
  - 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

```
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
```

Figure 2: Data structure literals in Factor

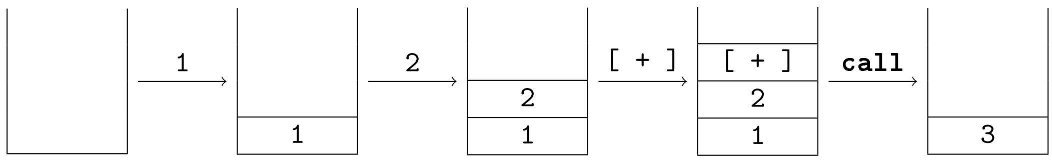

Figure 3: Quotations

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 those in Figure 2.

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 3 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 on page 16.

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 it requires at least three inputs but `+` might legitimately be called on only two.

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

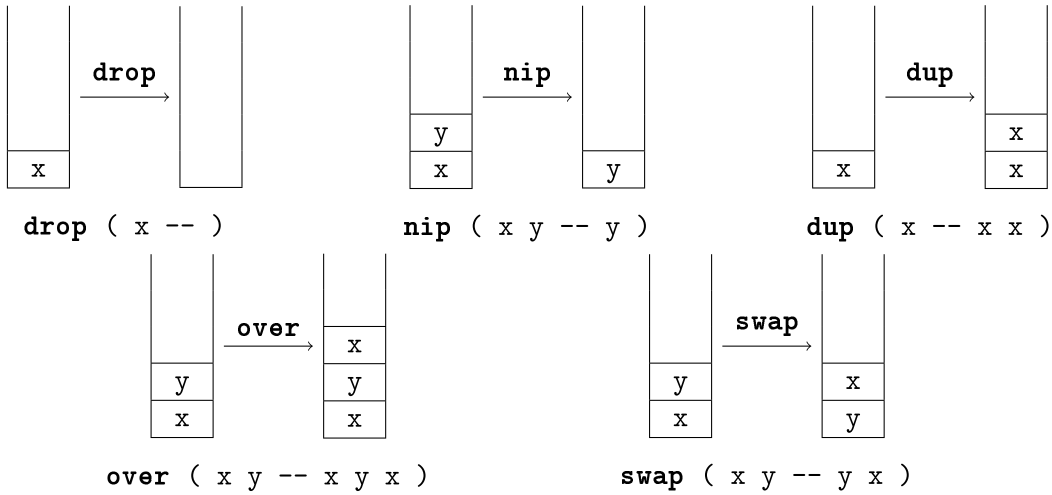

Figure 4: Stack shuffler words and their effects

on either side of the declaration names the top element of the stack. We can see this in Figure 4, 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 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 4 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 on page 16.

2.3 Definitions

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

Figure 5: Hello World in Factor

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 Figure 5 defines a word named `hello-world` which 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 Figure 6. This squares each of the top two numbers on the stack, adds them, then takes the square root of the sum. Figure 7 on the following page 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

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

Figure 6: The Euclidean norm, $\sqrt{x^2 + y^2}$

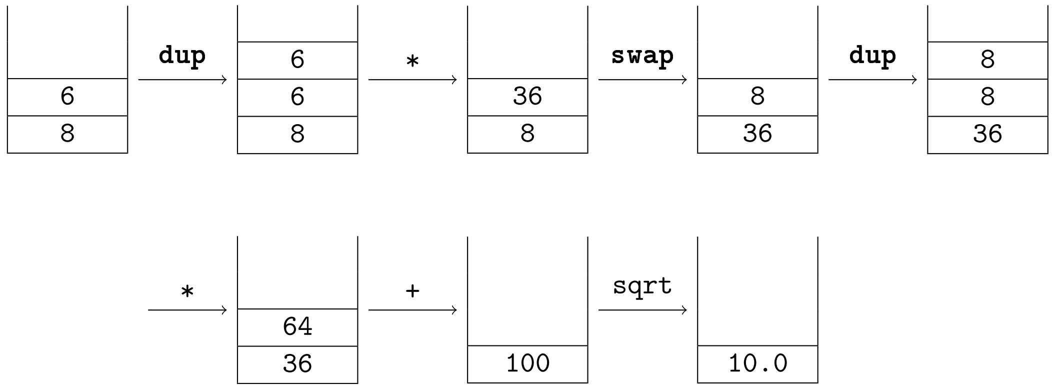

Figure 7: `norm` example

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

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

Figure 8: `norm` refactored

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.

As a simple case in point, we see the subexpression **`dup`** `*` appears twice in the definition of `norm` in Figure 6 on the previous page. We can easily factor that out into a new word and substitute it for the old expressions, as in Figure 8. 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.

Be that as it may, every once in awhile stack code gets too complicated to do away with more traditional notation. For these cases, Factor has a vocabulary called `locals`, which

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

Figure 9: `norm` with local variables

introduces syntax for defining words that use named lexical variables. Defining words with `::` instead of `:` turns the stack effect declaration into a full-fledged parameter list. The inputs are assigned to their corresponding names in the effect, which are used throughout the body in lieu of stack manipulation. The outputs just mean the same thing as before (i.e., the right side of the effect doesn't declare any variables like the left does). We can also assign local variables in the body of the word by using the syntax `:> destination`, which assigns `destination` to the value on the top of the stack. Figure 9 shows a version of `norm` that uses these features, though they aren't really necessary here. Interestingly, `locals` is implemented entirely in high-level Factor code, using parsing words to convert the syntax into equivalent stack manipulations.

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.

Tuples

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 Figure 10 on the following page. A class name is

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

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

Figure 10: Basic tuple definition syntax

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. Any number of slot specifiers follow, and the definition is terminated by the **`;`** token.

Tuple definitions automatically generate several different words, most of which depend on how slots are specified. There are various ways to specify slots, but we use only two basic forms in later code examples. We can see both in the first tuple of Figure 11 on the next page, which defines an object to represent regular expressions. The first three slots have the form `{ name read-only }`, which specifies a slot named `name` that can't be modified once initialized, akin to a **`final`** variable in Java. The next two specifiers are simpler, being just the names of the slots. Such slots can be modified freely. The following words are automatically defined for the first tuple:

- The `regexp` *class word* acts like a literal representing the class. This gets used for instantiation and method definitions, which we'll see later.

- The `regexp?` *class predicate* is a word with the stack effect `( object -- ? )`. That is, it returns a boolean (either **`t`** or **`f`**, conventionally written in stack effects as a single question mark) indicating whether the top of the stack is an instance of the `regexp` class. This is like a class-specific variant of Java's **`instanceof`**.

- Each slot has an associated *reader* word with the stack effect `( object -- value )`. These are analogous to "getter" methods in other languages. Each one is named after the slot whose value is extracted, so this example defines `raw>>`, `parse-tree>>`, `options>>`, `dfa>>`, and `next-match>>`.

```
TUPLE: regexp
    { raw read-only }
    { parse-tree read-only }
    { options read-only }
    dfa next-match ;

TUPLE: reverse-regexp < regexp ;
```

Figure 11: Sample tuple definitions from Factor's `regexp` vocabulary

- Similarly, any slot that is not marked `read-only` has a corresponding *writer* word with the stack effect `( value object -- )`. These destructively write the value into the eponymous slot of the object. Here, only two are defined, named `dfa<<` and `next-match<<`.

- Extra *setter* words are defined in terms of writers. These will have the stack effect `( object value -- object' )`, leaving the modified instance on top of the stack. The first tuple in Figure 11 defines `>>dfa` and `>>next-match`, which are equivalent to **`over`** `dfa<<` and **`over`** `next-match<<`, respectively. The shuffler duplicates `object` and pushes it to the top of the stack. More accurately, it duplicates a reference to `object`, as Factor's data stack is actually a stack of pointers. That way, changes to the new top of the stack with `dfa<<` or `next-match<<` will be reflected in the original `object`, which is left over at the end.

- *Changer* words are also created with the stack effect `( object quot -- object' )`. Here, `change-dfa` and `change-next-match` are defined. The quotation is called on the slot's current value in `object`. The result of calling the quotation is then stored in the slot. For instance, incrementing an integer slot named `foo` could be done with `[ 1 + ] change-foo`.

The second tuple in Figure 11 also defines a class word and predicate. Since it inherits from `regexp`, `reverse-regexp` gets the same five slots. If we had any other slot specifiers in the definition, it would have those in addition to the slots of its parent class. The reader, writer, setter, and changer methods will work on instances of `reverse-regexp`, since

```
TUPLE: color ;

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

TUPLE: rgb < color red green blue ;

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

Figure 12: Tuple constructors

inheritance establishes an "is-a" relationship from subclass to superclass—any instance of `reverse-regexp` is also an instance of `regexp`, though the reverse is not necessarily true. That is, `regexp?` will return **`t`** on instances of `reverse-regexp`, but `reverse-regexp?` will only return **`t`** on instances of `regexp` that are also `reverse-regexp`s. By viewing a class as the set of all objects that respond positively to the class predicate, we may partially order classes with the subset relationship. This fact will be important later.

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`**. For example, **`new`** is used in Figure 12 to define `<color>` (by convention, the constructor for `foo` is named `<foo>`). First, we push the class `color`, 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 (as long as the slots aren't `read-only`).

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 Figure 12. 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`.

Generics and Methods

Unlike more common object systems, we do not define individual methods that "belong" to particular tuples. In Factor, for a given generic word you define a method that specializes on a class. When the generic word is called on an object, it selects the method most specific to the object's class. This is determined by the aforementioned partial ordering of classes by their inheritance relationships.

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... ;
```

Factor's `sets` vocabulary gives us an accessible example of a generic word. `set` is a *mixin* class, defined by the **`MIXIN:`** parsing word. That is, the `set` class is a union of other classes, and users may extend the members of this union with the **`INSTANCE:`** word. We can this in Figure 13 on the following page, which shows the standard members of the `set` mixin. 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`.

Figure 14 on the next 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.

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

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

Figure 13: Set instances

```
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 ;
```

Figure 14: Set cardinality using Factor's object system

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`**.

We can see how the class ordering is used when `cardinality` selects the proper 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 on an `array`. 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 first. In the presence of explicit data flow via stack operations, even more patterns arise that can be abstracted away. The last half of this section explores how we can use combinators to express otherwise convoluted stack-shuffling logic more succinctly.

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 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 boolean condition, and it's followed by two quotations. The first quotation (`true`) is called if the condition is true, and the second quotation (`false`) 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`**.

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

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

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

Figure 15: Conditional evaluation in Factor

Basic **if** use is shown in Figure 15. The first example will print "odd", the second "empty", and the third "isn't f". All of them leave nothing on the stack.

However, the simplified stack effect for **if** is quite restrictive. Because the effect `( ? true false -- )` has no extra inputs and no outputs at all, it intuitively means that the `true` and `false` quotations both have the effect `( -- )`. 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 Figure 16 on the next page, 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 of inputs or 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 that any number of outputs will be present after the call to **if**. Note that these numbers

```
: 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 ;
```

Figure 16: **`if`**'s stack effect varies

aren't necessarily equal, which is why we use distinct row variables (`..a` and `..b`) 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, which has remained tacit to this point. For this, the notation `quot: ( stack -- effect )` gives 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 leave the stack at the same height as **`if`** does overall, and that neither expects any extra inputs.

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 recursive words that invoke quotations. Figure 17 on the following page 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 `quot`'s output stack height is exactly one less than the input. 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

```
{ "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
```

Figure 17: Loops in Factor

values. The first use of **each** in Figure 17 is balanced, as the quotation has the effect `( str -- )` and no additional items were on the stack to begin with (i.e., `...` stands in for 0 elements). 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 Figure 17 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

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

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

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

Figure 18: Higher-order functions in Factor

example in Figure 17 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 expect: 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 Figure 17 on the preceding page executes the body once, despite the condition being immediately false.

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 Figure 18. **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 Figure 18 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 Figure 18 is the same as `0 { 1 2 3 } [ + ]` **each**, as in Figure 17 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 such as 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.

Data Flow

While avoiding variables 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.

The first combinators we'll look at are **`dip`** and **`keep`**. These are used to preserve certain stack elements, do a computation, then restore said elements. 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 Figure 19 on the next 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 **`swap`** followed by **`swapd`** to rearrange the stack so we can call `-`, then **`swap`** it back to the desired order. This is even less clear, plus **`swapd`** is actually 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

```
: 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 )
    swap swapd - 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 ;
```

Figure 19: Preserving combinators

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 Figure 19, 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.

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. So,

```
x { [ a ] [ b ] [ c ] } cleave
```

takes the top element, `x`, and applies each quotation to it: `a` is applied to `x`, then `b` to `x`, then `c` to `x`. 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 three quotations. Cleave combinators are often used to extract multiple slots from a tuple. Figure 20 on the next 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 Figure 20 on the following 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.

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,

```
x y { [ a ] [ b ] } spread
```

```
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 ;
```

Figure 20: Cleave combinators

invokes `a` on `x` and `b` on `y`. 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 Figure 21 on the next page, 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` ("to uppercase") and `>lower` ("to lowercase") words are seen first, uninterrupted by an extra word. 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 Figure 22 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

```
: 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* ;
```

Figure 21: Spread combinators

```
: 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@ ;
```

Figure 22: Apply combinators

apply combinator. Thus, `with-bi@` cuts down on the duplicated `[ sq ]` in `without-bi@`. Similarly, we can replace the three repeated quotations passed 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.

This concludes our overview of Factor. Various other features are hidden away in different vocabularies, most of which are understandable based on the basics we've covered. For the purposes of this thesis, it's not important to know every little detail about Factor. Future code snippets will always be accompanied by further explanation, though they should generally be readable given this short tutorial.

3 The Factor Compiler

If we could sort programming languages by the fuzzy notions we tend to have about how "high-level" they are, toward the high end we'd find dynamically-typed languages like Python, Ruby, and PHP—all of which are generally more interpreted than compiled (though there has been compelling work on this front [e.g., Biggar 2009]). Despite being as high-level as these popular languages, Factor's implementation is driven by performance. Factor source is always compiled to native machine code using either its simple, non-optimizing compiler or (more typically) the optimizing compiler that performs several sorts of data and control flow analyses. In this chapter, we look at the general architecture of Factor's implementation, after which we place a particular emphasis on the transformations performed by the optimizing compiler.

References

Adobe Systems Incorporated. 1999. *PostScript Language Reference*. Third edition. Addison-Wesley. ISBN: 0-201-37922-8. URL: `http://partners.adobe.com/public/developer/en/ps/PLRM.pdf` (visited on August 15, 2011).

American National Standards Institute and Computer and Business Equipment Manufacturers Association. 1994. *American National Standard for information systems: programming languages: Forth: ANSI/X3.215-1994*. American National Standards Institute.

Biggar, P. 2009. "Design and Implementation of an Ahead-of-Time Compiler for PHP". PhD thesis. Trinity College, Dublin. URL: `http://www.paulbiggar.com/research/wip-thesis.pdf` (visited on August 15, 2011).

Diggins, C. 2007. *The Cat Programming Language*. URL: `http://cat-language.com/` (visited on August 15, 2011).

Factor. 2010. From Factor's documentation wiki. URL: `http://concatenative.org/wiki/view/Factor` (visited on August 15, 2011).