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. Section 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.

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},
}
```

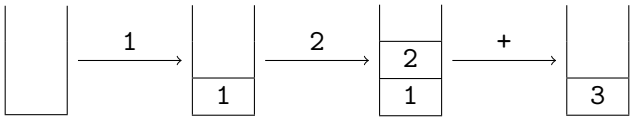

Figure 1: Visualizing stack-based calculation

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.

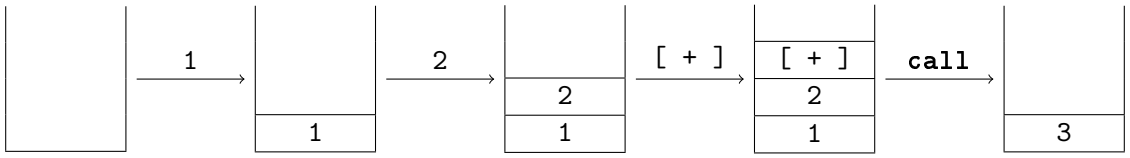

Figure 2: Quotations

- 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

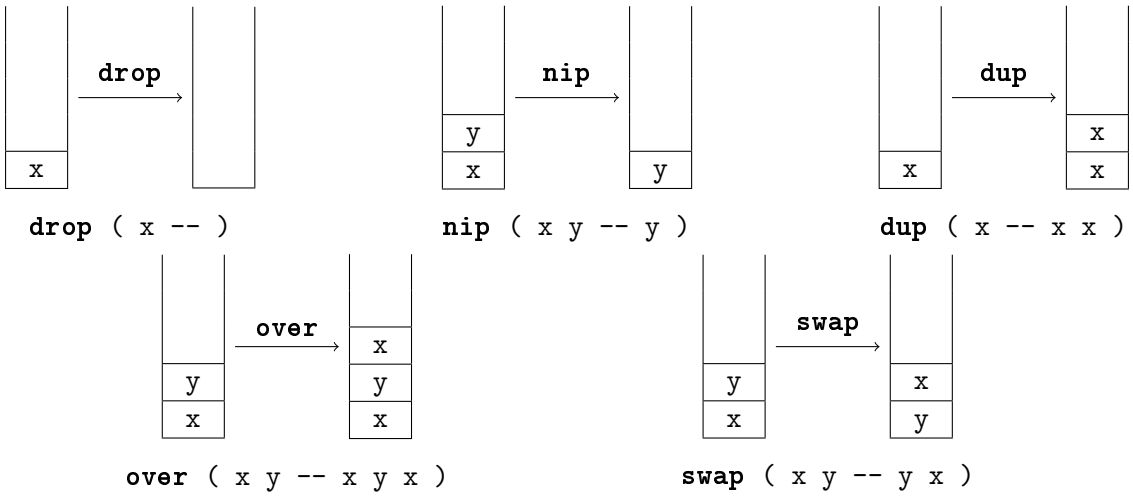

Figure 3: Stack shuffler words and their effects

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

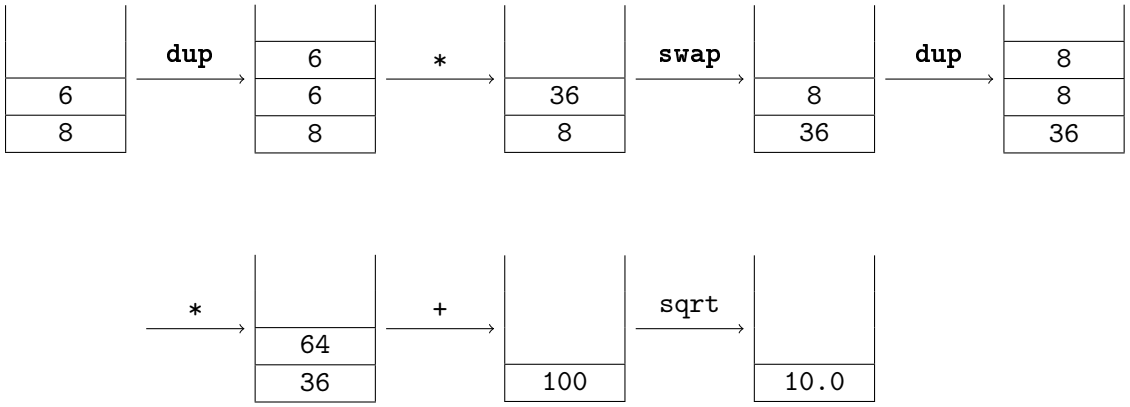

Figure 4: `norm` example

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.

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. 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 section, 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.

Though there are projects for this

3.1 Organization

At the lowest level, Factor is written atop a C++ virtual machine (VM) that is responsible for basic runtime services. This is where the non-optimizing base compiler is implemented. It's the base compiler's job to compile the simplest primitives: operations that push literals onto the data stack, **call**, **if**, **dip**, words that access tuple slots as laid out in memory, stack shufflers, math operators, functions to allocate/deallocate call stack frames, etc. The aim of the base compiler is to generate native machine code as fast as possible. To this end, these primitives correspond to their own stubs of assembly code. Different stubs are generated by Factor depending on the instruction set supported by the particular machine in use. Thus, the base compiler need only make a single pass over the source code, emitting these assembly instructions as it goes.

This compiled code is saved in an *image file*, which contains a complete snapshot of the current state of the Factor instance, similar to many Smalltalk and Lisp systems. As code is parsed and compiled, the image is updated, serving as a cache for compiled code. This modified image can be saved so that future Factor instances needn't recompile vocabularies that are already contained in the image.

cite?

The VM also handles method dispatch and memory management. Method dispatch incorporates a *polymorphic inline cache* to speed up generic words. Each generic word's call site is associated with a state:

- In the *cold* state, the call site's instruction computes the right method for the class being dispatched upon, which is the operation we're trying to avoid. As it does this, a polymorphic inline cache stub is generated, thus transitioning it to the next state.
- In the *inline cache* state, a stub has been generated that caches the locations of methods for classes that have already been seen. This way, if a generic word at a particular call site is invoked often upon only a small number of classes (as is often in the case in loops, for example), we don't need to waste as much time doing method lookup. By default, if more than three different classes are dispatched upon, we transition to the next state.
- In the *megamorphic* state, the call instruction points to a larger cache that is allocated for the specific generic word (i.e., it is shared by all call sites). While not as efficient as an inline cache, this can still improve the performance of method dispatch.

To manage memory, the Factor VM uses a generational garbage collector (GC), which carves out sections of space on the heap for objects of different ages. Garbage in the oldest generation is collected with a mark-sweep-compact algorithm, while younger generations rely on a copying collector. This way, the GC is specialized for large numbers of short-lived objects that will stay in the younger generations without being promoted to the older generation. In the oldest space, even compiled code can be compacted. This is to avoid heap fragmentation in applications that must call the compiler at runtime, such as Factor's interactive development environment.

cite?

Values are referenced by tagged pointers, which use the three least significant bits of the pointer's address to store type information. This is possible because Factor aligns objects on an eight-byte boundary, so the three least significant bits of an address are always 0. These bits give us eight unique tags, but since Factor has more than eight data types, two tags are reserved to indicate that the type information is stored elsewhere. One is for VM types without their own tag, and the other is for user-defined tuples, each of which has its own type. Sufficiently small integers (e.g., 29-bit integers on a 32-bit machine, since the other 3 bits are used for the type tag) are stored directly in the pointer, so they needn't be heap-allocated. Larger integers and floating point numbers are boxed, but the optimizing compiler may unbox them to store floats in registers.

The VM is meant to be minimal, as Factor is mostly *self-hosting*. That is, the real workhorses of the language are written in Factor itself, including the standard libraries, parser, object system, and the optimizing compiler. It's possible for the compiler to be written in Factor because of the *bootstrapping* process that creates a new image from scratch. First, a minimal *boot image* is created from an existing *host* Factor instance. When the VM runs the boot image, it initiates the bootstrapping process. Using the host's parser, the base compiler will compile the core vocabularies

necessary to load the optimizing compiler. Once the optimizing compiler can itself be compiled, it is used to recompile (and thus optimize) all of the words defined so far. With the basic vocabularies recompiled, any additional vocabularies can be loaded using the optimized compiler and saved into a new, working image.

Thus, while the Factor VM is important, it is a small part of the code base. Since the bootstrapping process allows the optimizing compiler (hereafter just "the compiler") to be written in the same high-level language it's compiling, we can avoid the fiddly low-level details of the C++ backend. This is more conducive to writing advanced compiler optimizations, which are often complicated enough without having a concise, dynamically-typed, garbage-collected language like Factor to help us.

3.2 High-level Optimizations

To manipulate source code abstractly, we must have at least one intermediate representation (IR)—a data structure representing the instructions. It's common to convert between several IRs during compilation, as each form offers different properties that facilitate particular analyses. The Factor compiler optimizes code in passes across two different IRs: first at a high-level using the `compiler.tree` vocabulary, then at a low-level with the `compiler.cfg` vocabulary.

The high-level IR arranges code into a vector of `node` objects, which may themselves have children consisting of vectors of node—a tree structure that lends to the name `compiler.tree`. This ordered sequence of nodes represents control flow in a way that's effectively simple, annotated stack code. Listing 18 on the next page shows the definitions of the tuples that represent the "instruction set" of this stack code. Each object inherits (directly or indirectly) from the `node` class, which itself inherits from `identity-tuple`. This is a tuple whose `equal?` method is defined to always return `f` so that no two instances are equivalent unless they are the same instance.

Notice that most nodes define some sort of `in-d` and `out-d` slots, which mark each of them with the input and output data stacks. This represents the flow of data through the program. Here, stack values are denoted simply by integers, giving each value a unique identifier. An `#introduce` instance is inserted wherever the next node requires stack values that have not yet been named. Thus, while `#introduce` has no `in-d`, its `out-d` introduces the necessary stack values. Similarly, `#return` is inserted at the end of the sequence to indicate the final state of the data stack with its `in-d` slot.

The most basic operations of a stack language are, of course, pushing literals and calling functions. The `#push` node thus has a `literal` slot and an `out-d` slot, giving a name to the single element it pushes to the data stack. `#call`, of course, is used for normal word invocations. The `in-d` and `out-d` slots effectively serve as the stack effect declaration. In later analyses, data about the word's definition may be stored across the `body`, `method`, `class`, and `info` slots.

The word `build-tree` takes a Factor quotation and constructs the equivalent high-level IR form. In Listing 19 on page 21, we see the output of the simple example `[ 1 + ] build-tree`. Note that `T{ class { slot1 value1 } { slot2 value2 } ... }` is the syntax for tuple literals. The first node is a `#push` for the `1` literal. Since `+` needs two input values, an `#introduce` pushes a new "phantom" value. `+` gets turned into a `#call` instance. Notice the `in-d` slot refers to the values in the order that they're passed to the word, not necessarily the order they've been introduced in the

```
TUPLE: node < identity-tuple ;

TUPLE: #introduce < node out-d ;
TUPLE: #return < node in-d info ;

TUPLE: #push < node literal out-d ;
TUPLE: #call < node word in-d out-d body method class info ;

TUPLE: #renaming < node ;
TUPLE: #copy < #renaming in-d out-d ;
TUPLE: #shuffle < #renaming mapping in-d out-d in-r out-r ;

TUPLE: #declare < node declaration ;

TUPLE: #terminate < node in-d in-r ;

TUPLE: #branch < node in-d children live-branches ;
TUPLE: #if < #branch ;
TUPLE: #dispatch < #branch ;

TUPLE: #phi < node phi-in-d phi-info-d out-d terminated ;

TUPLE: #recursive < node in-d word label loop? child ;
TUPLE: #enter-recursive < node in-d out-d label info ;
TUPLE: #call-recursive < node label in-d out-d info ;
TUPLE: #return-recursive < #renaming in-d out-d label info ;

TUPLE: #alien-node < node params ;
TUPLE: #alien-invoke < #alien-node in-d out-d ;
TUPLE: #alien-indirect < #alien-node in-d out-d ;
TUPLE: #alien-assembly < #alien-node in-d out-d ;
TUPLE: #alien-callback < node params child ;
```

Listing 18: High-level IR nodes

```
V{
    T{ #push { literal 1 } { out-d { 6256273 } } }
    T{ #introduce { out-d { 6256274 } } }
    T{ #call
        { word + }
        { in-d V{ 6256274 6256273 } }
        { out-d { 6256275 } }
    }
    T{ #return { in-d V{ 6256275 } } }
}
```

Listing 19: `[ 1 + ] build-tree`

IR. The sum is pushed to the data stack, so the `out-d` slot is a singleton that names this value. Finally, `#return` indicates the end of the routine, its `in-d` containing the value left on the stack (the sum pushed by `#call`).

```
V{
    T{ #introduce { out-d { 6256132 6256133 } } }
    T{ #shuffle
        { mapping { { 6256134 6256133 } { 6256135 6256132 } } }
        { in-d V{ 6256132 6256133 } }
        { out-d V{ 6256134 6256135 } }
    }
    T{ #return { in-d V{ 6256134 6256135 } } }
}
```

Listing 20: `[ swap ] build-tree`

The next tuples in Listing 18 on the previous page reassign existing values on the stack to fresh identifiers. The `#renaming` superclass has the two subclasses `#copy` and `#shuffle`. The former represents the bijection from elements of `in-d` to elements of `out-d` in the same position; corresponding values are copies of each other. The latter represents a more general mapping. Stack shufflers are translated to `#shuffle` nodes with `mapping` slots that dictate how the fresh values in `out-d` correspond to the input values in `in-d`. For instance, Listing 20 shows how **`swap`** takes in the values `6256132` and `6256133` and outputs `6256134` and `6256135`, where the former is mapped to the second element (`6256133`) and the latter to the first (`6256132`). Thus, `out-d` swaps the two elements of `in-d`, mapping them to fresh identifiers. The `in-r` and `out-r` slots of `#shuffle` correspond to the *retain* stack, which is an implementation detail beyond the scope of this discussion.

`#declare` is a miscellaneous node used for the `declare` primitive. It simply annotates type information to stack values, as in Listing 21 on the next page. `#terminate` is another one-off node, but a much more interesting one. While Factor normally requires a balanced stack, sometimes we

```
V{
    T{ #introduce { out-d { 6256069 } } }
    T{ #declare { declaration { { 6256069 fixnum } } } }
    T{ #return { in-d V{ 6256069 } } }
}
```

Listing 21: `[ { fixnum } declare ] build-tree`

```
V{
    T{ #push { literal "Error!" } { out-d { 6256051 } } }
    T{ #call
        { word throw }
        { in-d V{ 6256051 } }
        { out-d { } }
    }
    T{ #terminate { in-d V{ } } { in-r V{ } } }
    T{ #return { in-d V{ } } }
}
```

Listing 22: `[ "Error!" throw ] build-tree`

purposefully want to throw an error. `#terminate` is introduced where the program halts prematurely. When checking the stack height, it gets to be treated specially so that *terminated* stack effects unify with any other effect. That way, branches will still be balanced even if one of them unconditionally throws an error. Listing 22 shows `#terminate` being introduced by the `throw` word.

Next, Listing 18 on page 20 defines nodes for branching based off the superclass `#branch`. The `children` slot contains vectors of nodes representing different branches. `live-branches` is filled in during later analyses to indicate which branches are alive so that dead ones may be removed. For instance, `#if` will have two elements in its `children` slot representing the true and false branches. On the other hand, `#dispatch` has an arbitrary number of children. It corresponds to the `dispatch` primitive, which is an implementation detail of the generic word system used to speed up method dispatch.

You may have noted the emphasis on introducing new values, instead of reassigning old ones. Even `#shuffle`s output fresh identifiers, letting their values be determined by the `mapping`. The reason for this is that `compiler.tree` uses static single assignment (SSA) form, wherein every variable is defined by exactly one statement. This simplifies the properties of variables, which helps optimizations perform faster and with better results. By giving unique names to the targets of each assignment, the SSA property is guaranteed. However, `#branch`es introduce ambiguity: after, say, an `#if`, what will the `out-d` be? It depends on which branch is taken. To remedy this problem, after any `#branch` node, Factor will place a `#phi` node—the classical SSA "phony function", ϕ. While it doesn't perform any literal computation, conceptually ϕ selects between its inputs, choosing the "correct" argument depending on control flow. This can then be assigned to

cite?

a unique value, preserving the SSA property. In Factor, this is represented by a `phi-in-d` slot, which is a sequence of sequences. Each element corresponds to the `out-d` of the child at the same position in the `children` of the preceding `#branch` node. The `#phi`'s `out-d` gives unique names to the output values.

```
V{
    T{ #introduce { out-d { 6256247 } } }
    T{ #if
        { in-d { 6256247 } }
        { children
            {
                V{
                    T{ #push
                        { literal 1 }
                        { out-d { 6256248 } }
                    }
                }
                V{
                    T{ #push
                        { literal 2 }
                        { out-d { 6256249 } }
                    }
                }
            }
        }
    }
    T{ #phi
        { phi-in-d { { 6256248 } { 6256249 } } }
        { out-d { 6256250 } }
        { terminated V{ f f } }
    }
    T{ #return { in-d V{ 6256250 } } }
}
```

Listing 23: `[ [ 1 ] [ 2 ] if ] build-tree`

For example, the `#phi` in Listing 23 will select between the `6256248` return value of the first child or the `6256249` output of the second. Either way, we can refer to the result as `6256250` afterwards. The `terminated` slot of the `#phi` tells us if there was a `#terminate` in any of the branches.

The `#recursive` node encapsulates *inline recursive* words. In Factor, words may be annotated with simple compiler declarations, which guide optimizations. If we follow a standard colon definition with the **`inline`** word, we're saying that its definition can be spliced into the call-site, rather than generating code to jump to a subroutine. Inline words that call themselves must additionally be declared **`recursive`**. For example, we could write `: foo ( -- ) foo ;` **`inline recursive`**.

The nodes `#enter-recursive`, `#call-recursive`, and `#return-recursive` denote different stages of the recursion—the beginning, recursive call, and end, respectively. They carry around a lot of metadata about the nature of the recursion, but it doesn't serve our purposes to get into the details. Similarly, we gloss over the final nodes of Listing 18 on page 20 correspond to Factor's foreign function interface (FFI) vocabulary, called `alien`. At a high level, `#alien-node`, `#alien-invoke`, `#alien-indirect`, `#alien-assembly`, and `#alien-callback` are used to make calls to C libraries from within Factor.

```
: optimize-tree ( nodes -- nodes' )
  [
      analyze-recursive
      normalize
      propagate
      cleanup
      dup run-escape-analysis? [
          escape-analysis
          unbox-tuples
      ] when
      apply-identities
      compute-def-use
      remove-dead-code
      ?check
      compute-def-use
      optimize-modular-arithmetic
      finalize
  ] with-scope ;
```

Listing 24: Optimization passes on the high-level IR

Shouldn't bold "cleanup" in Listing 24

Now that we're familiar with the structure of the high-level IR, we can turn our attention to optimization. Listing 24 shows the passes performed on a sequence of nodes by the word `optimize-tree`. Before optimization can begin, we must gather some information and clean up some oddities in the output of `build-tree`. `analyze-recursive` is called first to identify and mark loops in the tree. Effectively, this means we detect tail-recursion introduced by `#recursive` nodes. Future passes can then use this information for data flow analysis. Then, `normalize` makes the tree more consistent by doing two things:

- All `#introduce` nodes are removed and replaced by a single `#introduce` at the beginning of the whole program. This way, further passes needn't handle `#introduce` nodes.
- As constructed, the `in-d` of a `#call-recursive` will be the entire stack at the time of the call. This assumption happens because we don't know how many inputs it needs until the `#return-recursive` is processed, because of row polymorphism. So, here we figure out

exactly what stack entries are needed, and trim the `in-d` and `out-d` of each `#call-recursive` accordingly.

Once these passes have cleaned up the tree, `propagate` performs probably the most extensive analysis of all the phases. In short, it performs an extended version of sparse conditional constant propagation (SCCP). The traditional data flow analysis combines global copy propagation, constant propagation, and constant folding in a *flow-sensitive* way. That is, it will propagate information from branches that it knows are definitely taken (e.g., because `#if` is always given a true input). Instead of using the typical single-level (numeric) constant value lattice, Factor uses a lattice augmented by information about classes, numeric value ranges, array lengths, and tuple slots' classes. Classes can be used in the lattice with the partial-order protocol described briefly in Section 2.4. Additionally, the transfer functions are allowed to inline certain calls if enough information is present. This occurs in the transfer function since generic words' inline expansions into particular methods provide more information, thus giving us more opportunities for propagation. This is particularly useful for arithmetic words. In Factor, words like `+` and `*` are generics that work across all sorts of numeric representations, be they **`fixnum`**s, **`float`**s, **`bignum`**s, etc. If the operation overflows, the values are automatically cast up to larger representations. But iterated refinement of the inputs' classes can let the compiler select more specific, efficient methods (e.g., if both arguments are **`fixnum`**s).

cite

Interval propagation also helps propagate class information. By refining the range of possible values a particular item can have, we might discover that, say, it's small enough to fit in a **`fixnum`** rather than a **`bignum`**. There are plenty more things that interval propagation can tell us, too. For example, it may give us enough information to remove overflow checks performed by numeric words. And if the interval has zero length, we may replace the value with a constant. This then continues getting propagated, contributing to constant folding and so forth.

`propagate` iterates through the nodes collecting all of this data until reaching a stable point where inferences can no longer be drawn. Technically, this information doesn't alter the tree at all; we merely store it so that speculative decisions may be realized later. The next word in Listing 24 on the previous page, `cleanup`, does just this by inlining words, folding constants, removing overflow checks, deleting unreachable branches, and flattening inline-recursive words that don't actually wind up calling themselves (e.g., because the calls got constant-folded).

The next major pass is `escape-analysis`, whose information is used for the actual transformation `unbox-tuples`. This discovers tuples that *escape* by getting passed outside of a word. For instance, the inputs to `#return` obviously escape, as they are passed to the world outside of the word in question. Similarly, inputs to the `#call` of another word escape. So, though the tuples in `escaping-via-#return` and `escaping-via-#call` in Listing 25 on the following page both escape, we can see the one in `non-escaping` does not. In fact, the last allocation is unnecessary. By identifying this, `unbox-tuples` can then rewrite the code to avoid allocating a `data-struct` altogether, instead manipulating the slots' values directly. Note that this only happens for immutable tuples, all of whose slots are `read-only`. Otherwise, we would need to perform more advanced pointer analyses to discover aliases.

`apply-identities` follows to simplify words with known identity elements. If, say, an argument to `+` is `0`, we can simply return the other argument. This converts the `#call` to `+` into a simple `#shuffle`. These identities are defined for most arithmetic words.

```
TUPLE: data-struct
  { a read-only }
  { b read-only } ;

: escaping-via-#return ( -- data-struct )
  1 2 data-struct boa ;

: escaping-via-#call ( -- )
  1 2 data-struct boa pprint ;

: non-escaping ( -- )
  1 2 data-struct [ a>> ] [ b>> ] bi + ;
```

Listing 25: Escaping vs. non-escaping tuple allocations

Another simple few passes come next in Listing 24 on page 24. True to its name, `compute-def-use` computes where SSA values are defined and used. Values that are never used are eliminated by `remove-dead-code`. `?check` conditionally performs some consistency checks on the tree, mostly to make sure that no errors were introduced in the stack flow. If a global variable isn't toggled on, this part is skipped. We run `compute-def-use` again to update the information after altering the tree with dead code elimination.

Finally, `optimize-modular-arithmetic` performs a form of strength-reduction on artihmetic words that only use the low-order bits of their inputs/results, which may also remove more unnecessary overflow checks. `finalize` cleans up a few random miscellaneous bits of the tree (removing empty shufflers, deleting `#copy` nodes, etc.) in preparation for lower-level optimizations.

Double-check zealous syntax-highlighting

3.3 Low-level Optimizations

The low-level IR in `compiler.cfg` takes the more conventional form of a control flow graph (CFG). A CFG (not to be confused with "context-free grammar") is an arrangement of instructions into *basic blocks*: maximal sequences of "straight-line" code, where control does not transfer out of or into the middle of the block. Directed edges are added between blocks to represent control flow—either from a branching instruction to its target, or from the end of a basic block to the start of the next one. Construction of the low-level IR proceeds by analyzing the control flow of the high-level IR and converting the nodes of Section 3.2 into lower-level, more conventional instructions modeled after typical assembly code. There are over a hundred of these instructions, but many are simply different versions of the same operation. For instance, while instructions are generally called on *virtual registers* (represented in Factor simply by integers), there are *immediate* versions of instructions. The `##add` instruction, as an example, represents the sum of the contents of two registers, but `##add-imm` sums the contents of one register and an integer literal. Other instructions are inserted to make stack reads and writes explicit, as well as to balance the height. Below is a categorized list of all the instruction objects (each one is a subclass of the `insn` tuple).

cite

Is the complete list really necessary?

- Loading constants: `##load-integer`, `##load-reference`
- Optimized loading of constants, inserted by representation selection: `##load-tagged`, `##load-float`, `##load-double`, `##load-vector`
- Stack operations: `##peek`, `##replace`, `##replace-imm`, `##inc-d`, `##inc-r`
- Subroutine calls: `##call`, `##jump`, `##prologue`, `##epilogue`, `##return`
- Inhibiting tail-call optimization (TCO): `##no-tco`
- Jump tables: `##dispatch`
- Slot access: `##slot`, `##slot-imm`, `##set-slot`, `##set-slot-imm`
- Register transfers: `##copy`, `##tagged>integer`
- Integer arithmetic: `##add`, `##add-imm`, `##sub`, `##sub-imm`, `##mul`, `##mul-imm`, `##and`, `##and-imm`, `##or`, `##or-imm`, `##xor`, `##xor-imm`, `##shl`, `##shl-imm`, `##shr`, `##shr-imm`, `##sar`, `##sar-imm`, `##min`, `##max`, `##not`, `##neg`, `##log2`, `##bit-count`
- Float arithmetic: `##add-float`, `##sub-float`, `##mul-float`, `##div-float`, `##min-float`, `##max-float`, `##sqrt`
- Single/double float conversion: `##single>double-float`, `##double>single-float`
- Float/integer conversion: `##float>integer`, `##integer>float`
- SIMD operations: `##zero-vector`, `##fill-vector`, `##gather-vector-2`, `##gather-int-vector-2`, `##gather-vector-4`, `##gather-int-vector-4`, `##select-vector`, `##shuffle-vector`, `##shuffle-vector-halves-imm`, `##shuffle-vector-imm`, `##tail>head-vector`, `##merge-vector-head`, `##merge-vector-tail`, `##float-pack-vector`, `##signed-pack-vector`, `##unsigned-pack-vector`, `##unpack-vector-head`, `##unpack-vector-tail`, `##integer>float-vector`, `##float>integer-vector`, `##compare-vector`, `##test-vector`, `##test-vector-branch`, `##add-vector`, `##saturated-add-vector`, `##add-sub-vector`, `##sub-vector`, `##saturated-sub-vector`, `##mul-vector`, `##mul-high-vector`, `##mul-horizontal-add-vector`, `##saturated-mul-vector`, `##div-vector`, `##min-vector`, `##max-vector`, `##avg-vector`, `##dot-vector`, `##sad-vector`, `##horizontal-add-vector`, `##horizontal-sub-vector`, `##horizontal-shl-vector-imm`, `##horizontal-shr-vector-imm`, `##abs-vector`, `##sqrt-vector`, `##and-vector`, `##andn-vector`, `##or-vector`, `##xor-vector`, `##not-vector`, `##shl-vector-imm`, `##shr-vector-imm`, `##shl-vector`, `##shr-vector`
- Scalar/vector conversion: `##scalar>integer`, `##integer>scalar`, `##vector>scalar`, `##scalar>vector`

- Boxing and unboxing aliens: `##box-alien`, `##box-displaced-alien`, `##unbox-any-c-ptr`, `##unbox-alien`
- Zero-extending and sign-extending integers: `##convert-integer`
- Raw memory access: `##load-memory`, `##load-memory-imm`, `##store-memory`, `##store-memory-imm`
- Memory allocation: `##allot`, `##write-barrier`, `##write-barrier-imm`, `##alien-global`, `##vm-field`, `##set-vm-field`
- The FFI: `##unbox`, `##unbox-long-long`, `##local-allot`, `##box`, `##box-long-long`, `##alien-invoke`, `##alien-indirect`, `##alien-assembly`, `##callback-inputs`, `##callback-outputs`
- Control flow: `##phi`, `##branch`
- Tagged conditionals: `##compare-branch`, `##compare-imm-branch`, `##compare`, `##compare-imm`
- Integer conditionals: `##compare-integer-branch`, `##compare-integer-imm-branch`, `##test-branch`, `##test-imm-branch`, `##compare-integer`, `##compare-integer-imm`, `##test`, `##test-imm`
- Float conditionals: `##compare-float-ordered-branch`, `##compare-float-unordered-branch`, `##compare-float-ordered`, `##compare-float-unordered`
- Overflowing arithmetic: `##fixnum-add`, `##fixnum-sub`, `##fixnum-mul`
- GC checks: `##save-context`, `##check-nursery-branch`, `##call-gc`
- Spills and reloads, inserted by the register allocator: `##spill`, `##reload`

```
: optimize-cfg ( cfg -- cfg' )
    optimize-tail-calls
    delete-useless-conditionals
    split-branches
    join-blocks
    normalize-height
    construct-ssa
    alias-analysis
    value-numbering
    copy-propagation
    eliminate-dead-code ;
```

Listing 26: Optimization passes on the low-level IR

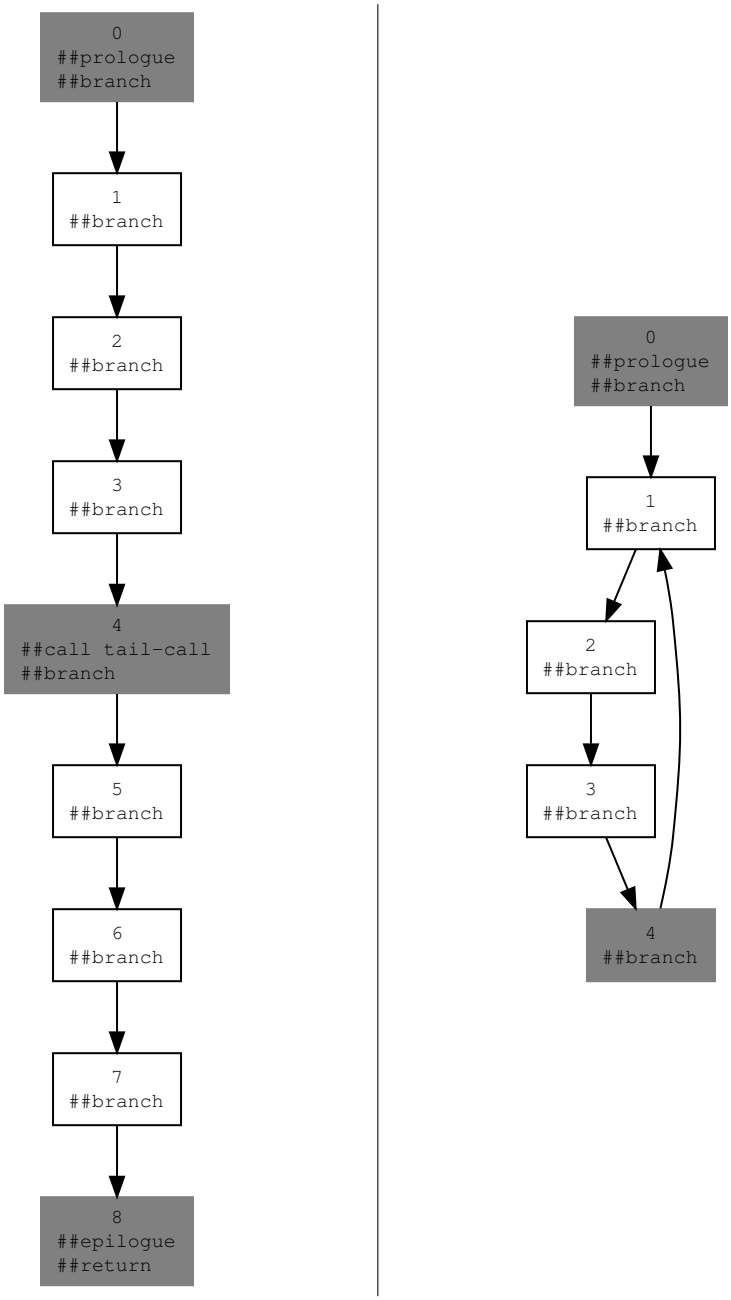

Figure 5: `tail-call` before and after `optimize-tail-calls`

By translating the high-level IR into instructions that manipulate registers directly, we reveal further redundancies that can be optimized away. The `optimize-cfg` word in Listing 26 on page 28 shows the passes performed in doing this. The first word, `optimize-tail-calls`, performs tail call elimination on the CFG. *Tail calls* are those that occur within a procedure and whose results are immediately returned by that procedure. Instead of allocating a new call stack frame, we may convert tail calls into simple jumps, since afterwards the current procedure's call frame isn't really needed. In the case of recursive tail calls, we can convert special cases of recursion into loops in the CFG, so that we won't trigger call stack overflows. For instance, consider Figure 5 on the previous page, which shows the effect of `optimize-tail-calls` on the following definition:

used in Section 3.2, not defined

```
: tail-call ( -- ) tail-call ;
```

Note the recursive call (trivially) occurs at the end of the definition, just before the return point. When translated to a CFG, this is a `##call` instruction, as seen in block 4 to the left of Figure 5 on the preceding page. This is also just before the final `##epilogue` and `##return` instructions in block 8, as blocks 5–7 are effectively empty (these excessive `##branch`es will be eliminated in a later pass). Because of this, rather than make a whole new subroutine call, we can convert it into a `##branch` back to the beginning of the word, as in the CFG to the right.

The next pass in Listing 26 on page 28 is `delete-useless-conditionals`, which removes branches that go to the same basic block. This situation might occur as a result of optimizations performed in the high-level IR. To see it in action, Figure 6 on the following page shows the transformation on a purposefully useless conditional, `[ ] [ ] if`. Before removing the useless conditional, the CFG `##peek`s at the top of the data stack (`D 0`), storing the result in the virtual register `1`. This value is popped, so we decrement the stack height (`##inc-d -1`). Then, `##compare-imm-branch` in block 2 compares the value in the virtual register `2` (which is a copy of `1`, the top of the stack) to the immediate value `f` to see if it's not equal (signified by `cc/=`). However, both branches jump through several empty blocks and merge at the same destination. Thus, we can remove both branches and replace `##compare-imm-branch` with an unconditional `##branch` to the eventual destination. We see this on the right of Figure 6 on the next page.

In order to expose more opportunities for optimization, `split-branches` will actually duplicate code. We use the fact that code immediately following a conditional will be executed along either branch. If it's sufficiently short, we copy it up into the branches individually. That is, we change `[ A ] [ B ] if C` into `[ A C ] [ B C ] if`, as long as `C` is small enough. Later analyses may then, for example, more readily eliminate one of the branches if it's never taken. Figure 7 on page 32 shows what such a transformation looks like on a CFG. The example `[ 1 ] [ 2 ] if dup` is essentially changed into `[ 1 dup ] [ 2 dup ] if`, thus splitting the block with two predecessors (block 9) on the left.

The next pass, `join-blocks`, compacts the CFG by joining together blocks involved in only a single control flow edge. Mostly, this is to clean up the myriad of empty or short blocks introduced during construction, like sequences of a bunch of `##branch`es. Figure 8 on page 33 shows this pass on the CFG of `0 100 [ 1 fixnum+fast ] times`. `fixnum+fast` is a specialized version of `+` that suppresses overflow and type checks. We use it here to keep the CFG simple. We'll be using this particular code to illustrate all but one of the remaining optimization passes in Listing 26 on page 28, as it's a motivating example for the work in this thesis. The passes before `join-blocks`

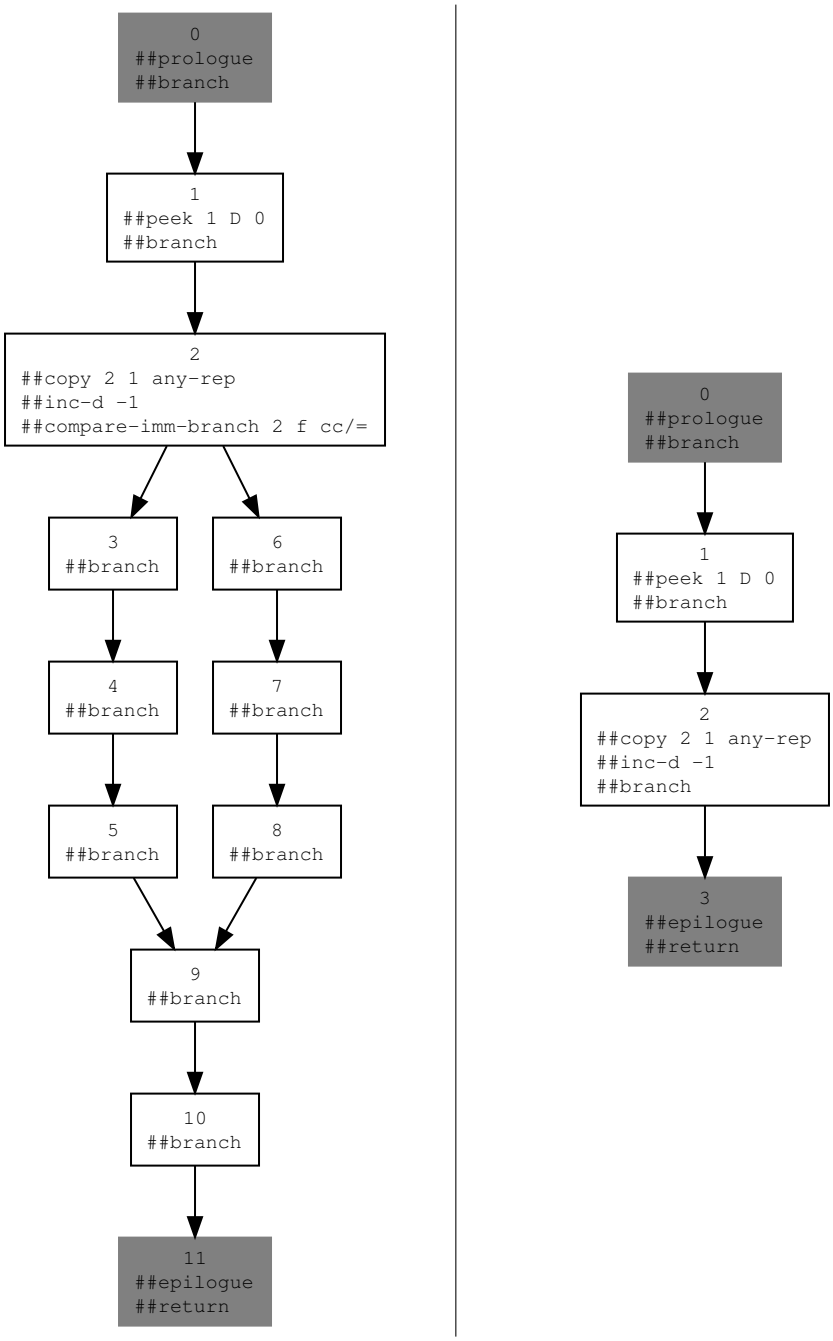

Figure 6: `[ ] [ ] if` before and after `delete-useless-conditionals`

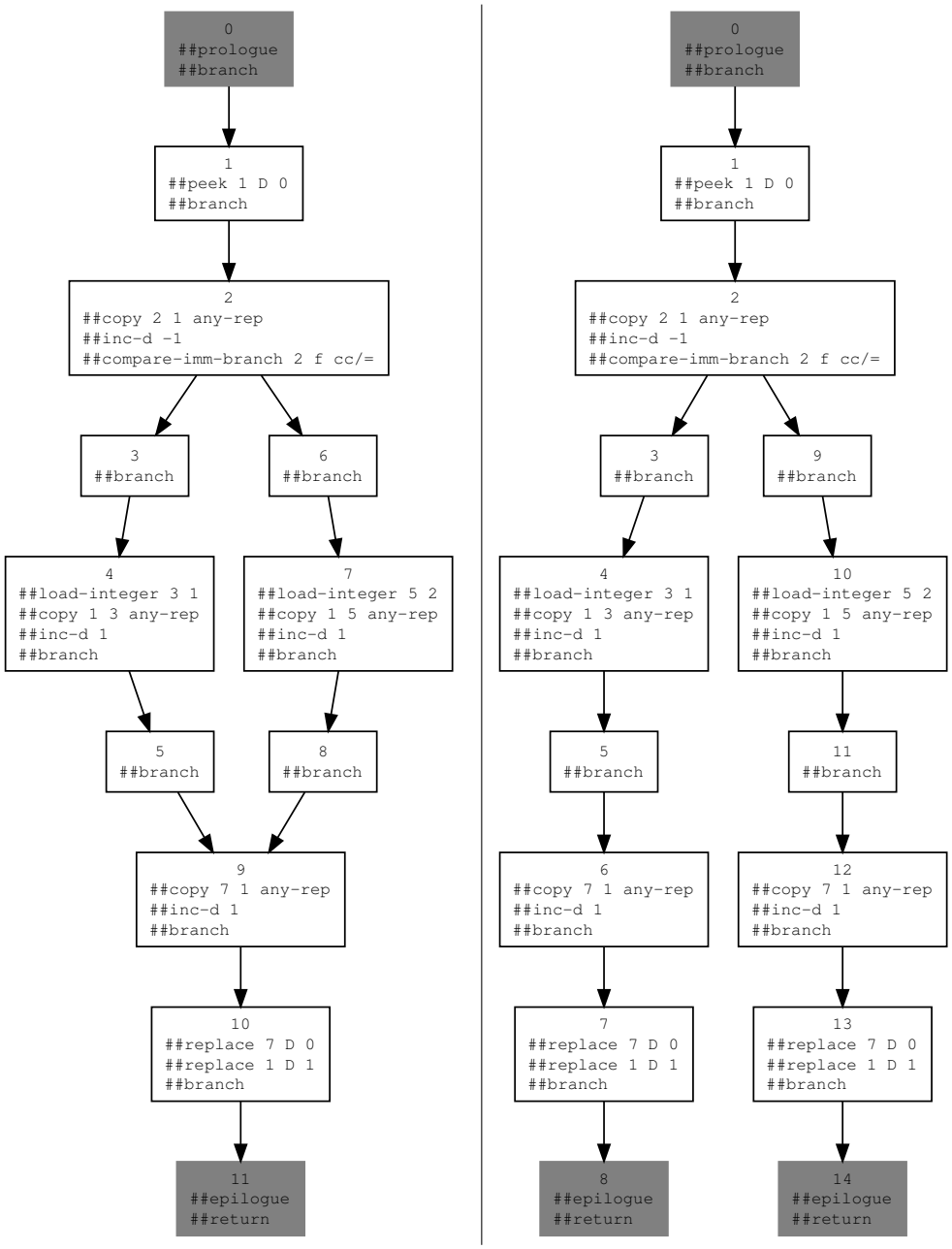

Figure 7: `[ 1 ] [ 2 ]` **`if dup`** before and after `split-branches`

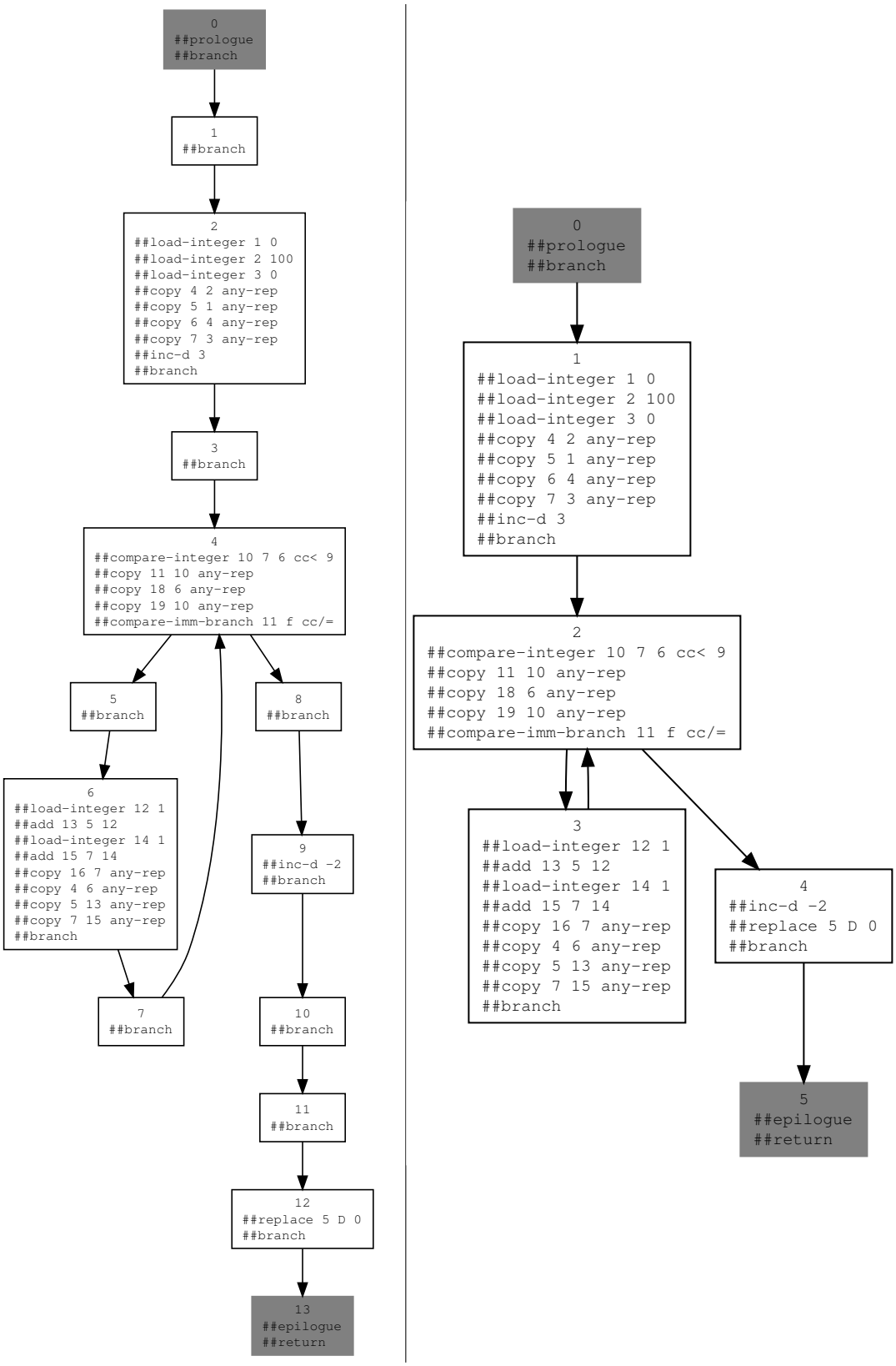

Figure 8: `0 100 [ 1 fixnum+fast ] times` before and after `join-blocks`

don't change the CFG seen on the left in Figure 8 on the previous page, but we get rid of the useless `##branch` blocks in the CFG on the right.

Figure 9 on the following page shows the result of applying `normalize-height` to the result of `join-blocks`. This phase combines and canonicalizes the instructions that track the stack height, like `##inc-d`. While the shuffling in this example isn't complex enough to be interesting, neither is this phase. It amounts to more cleanup: multiple height changes are combined into single ones at the beginnings of the basic blocks. In Figure 9 on the next page, this means that `##inc-d` is moved to the top of block 1, as compared to the right of Figure 8 on the preceding page.

In converting the high-level IR to the low-level, we actually lose the SSA form of `compiler.tree`. Not only does the construction do this, but `split-branches` also copies basic blocks verbatim, so any value defined will have a duplicate definition site, violating the SSA property. `construct-ssa` recomputes a so-called *pruned* SSA form, wherein ϕ functions are inserted only if the variables are live after the insertion point. This cuts down on useless ϕ functions. Figure 10 on page 36 shows the reconstructed SSA form of the CFG from Figure 9 on the following page.

cite TDMSC and construction algorithm

The next pass, `alias-analysis`, doesn't change the CFG of `0 100 [ 1 fixnum+fast ] times`, so we won't have an accompanying figure. At a high level, `alias-analysis` is easy to understand: it eliminates redundant memory loads and stores by rewriting certain patterns of memory access. If the same location is loaded after being stored, we convert the latter load into a `##copy` of the value we stored. Two reads of the same location with no intermittent write gets the second read turned into a `##copy`. Similarly, if we see two writes without a read in the middle, the first write can be removed.

`value-numbering` is the key focus of this thesis. It will be detailed in **??** on page ??. For now, it does to think of it as a combination of common subexpression elimination and constant folding. In Figure 11 on page 37, we see several changes:

- `##load-integer 23 0` in block 1 of Figure 10 on page 36 (which assigns the value `0` to the virtual register `23`) is redundant, so is replaced by `##copy 23 21`.
- In block 2, the last instruction `##compare-imm-branch 32 f cc/=` is the same as `##compare-integer-branch 30 26 cc<`. The source register (`32`) of the original is a `##copy` of `31`, which itself is computed by `##compare-integer 31 30 26 cc< 9`. So, the `##compare-imm-branch` is equivalent to a simple `##compare-integer-branch`, which doesn't use the temporary virtual register `9` and doesn't waste time comparing against the `f` object.
- The second operands in both `##add`s of block 3 are just constants stored by `##load-integer`s. So, these are turned into `##add-imm`s.
- Also, the second `##load-integer` in block 3 just loads `1` like the first instruction. Therefore, it's replaced by a `##copy`.

In **??**, we'll see how and why this pass fails to identify other equivalences.

Following `value-numbering`, `copy-propagation` performs a global pass that eliminates `##copy` instructions. Uses of the copies are replaced by the originals. So, in Figure 12 on page 38, we can

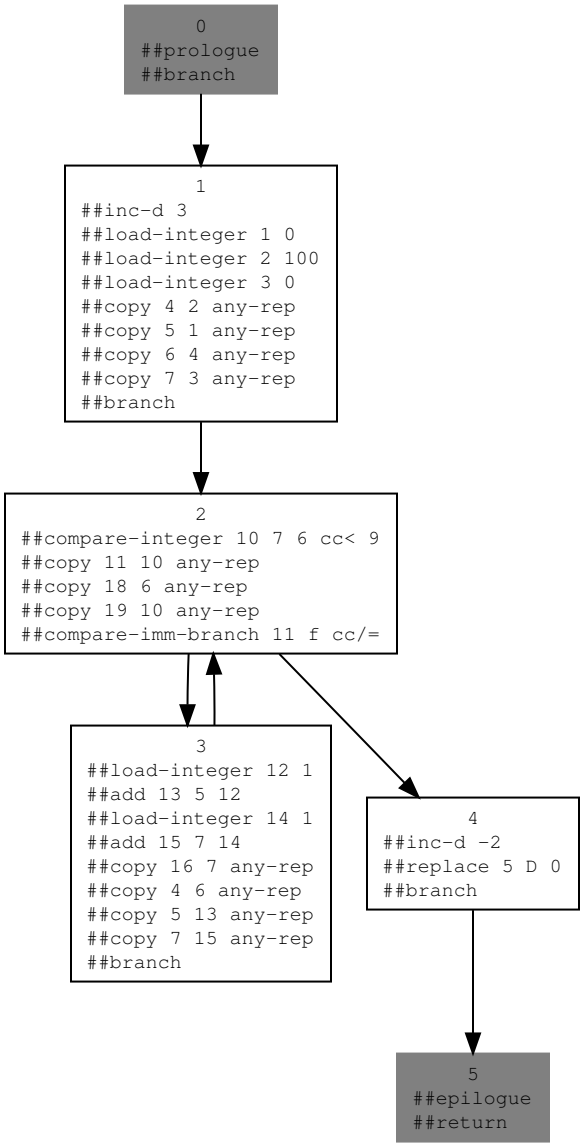

Figure 9: `0 100 [ 1 fixnum+fast ] times` after `normalize-height`

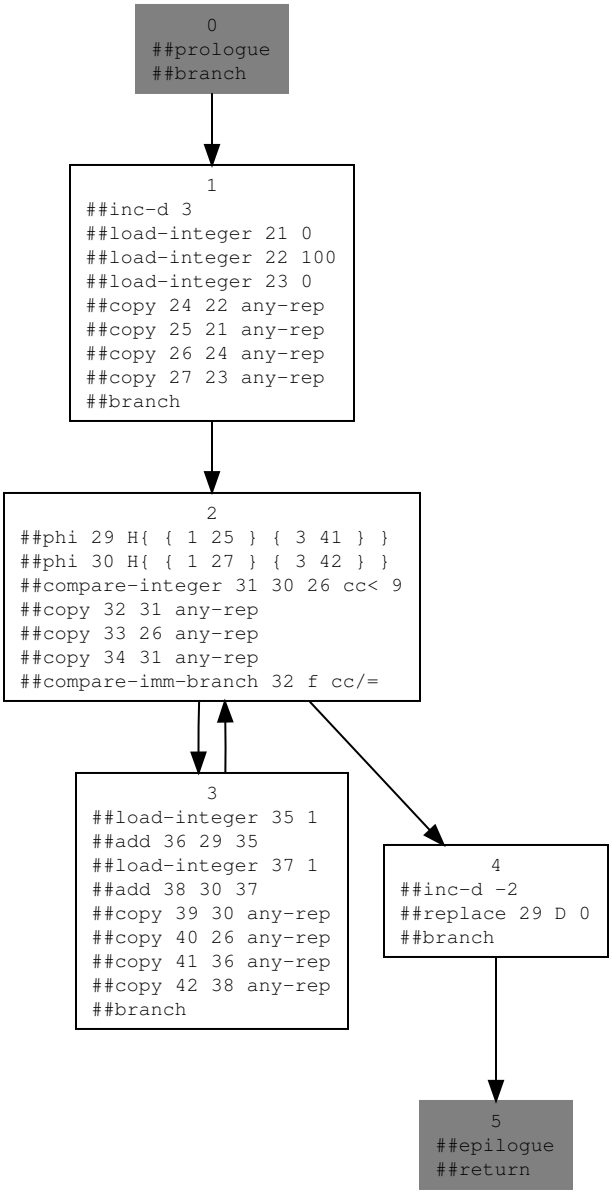

Figure 10: `0 100 [ 1 fixnum+fast ]` **`times`** after `construct-ssa`

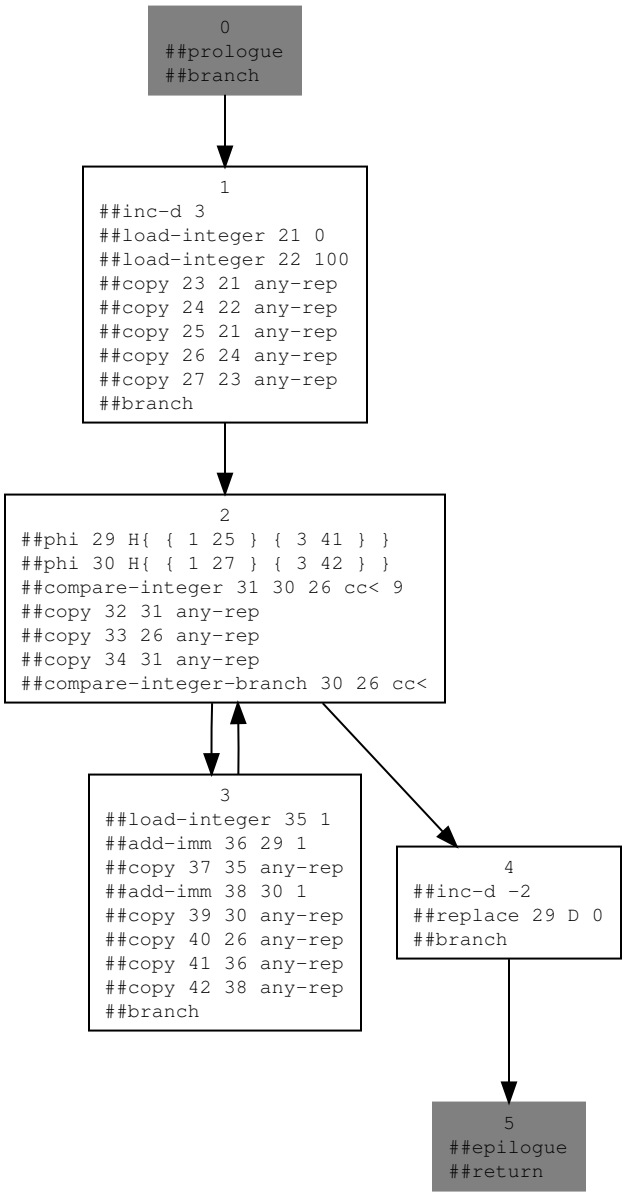

Figure 11: `0 100 [ 1 fixnum+fast ] times` after `value-numbering`

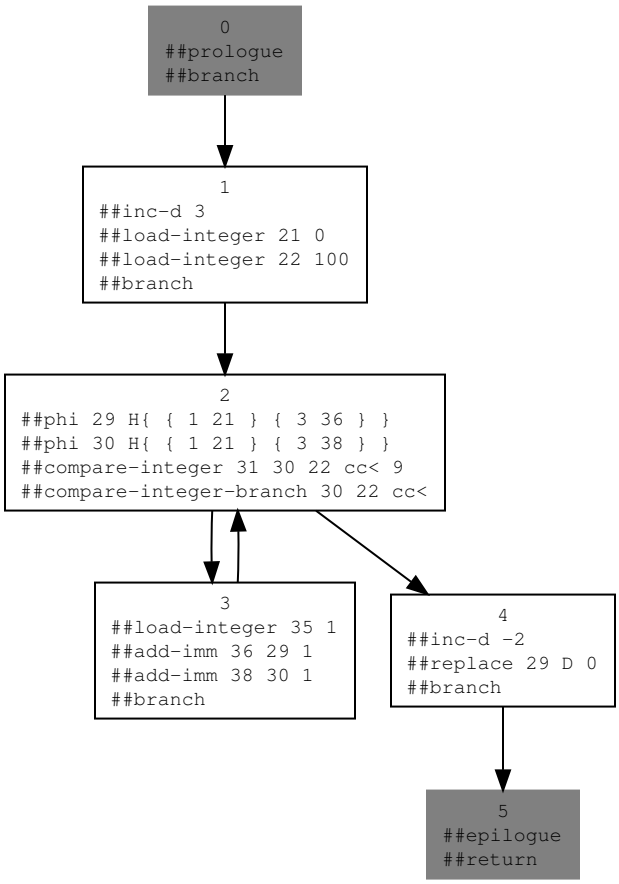

Figure 12: `0 100 [ 1 fixnum+fast ] times` after `copy-propagation`

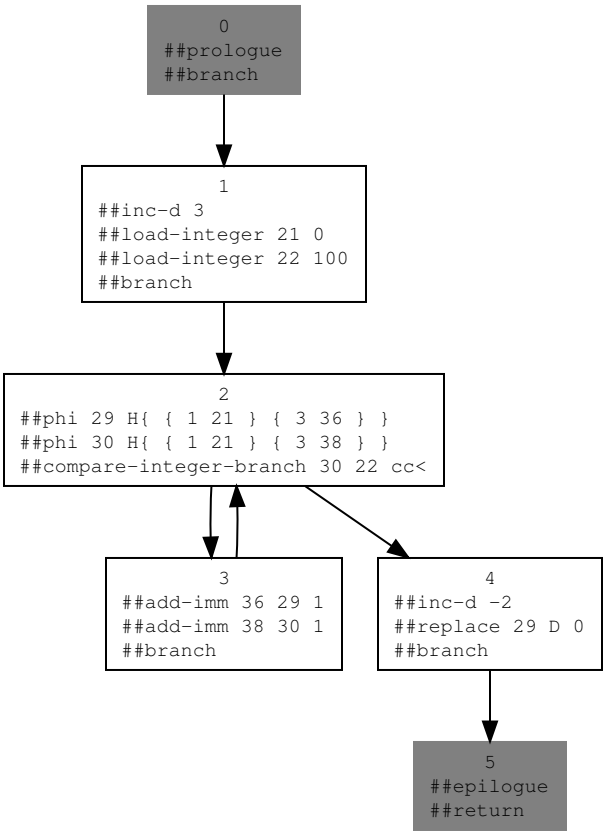

Figure 13: `0 100 [ 1 fixnum+fast ] times` after `eliminate-dead-code`

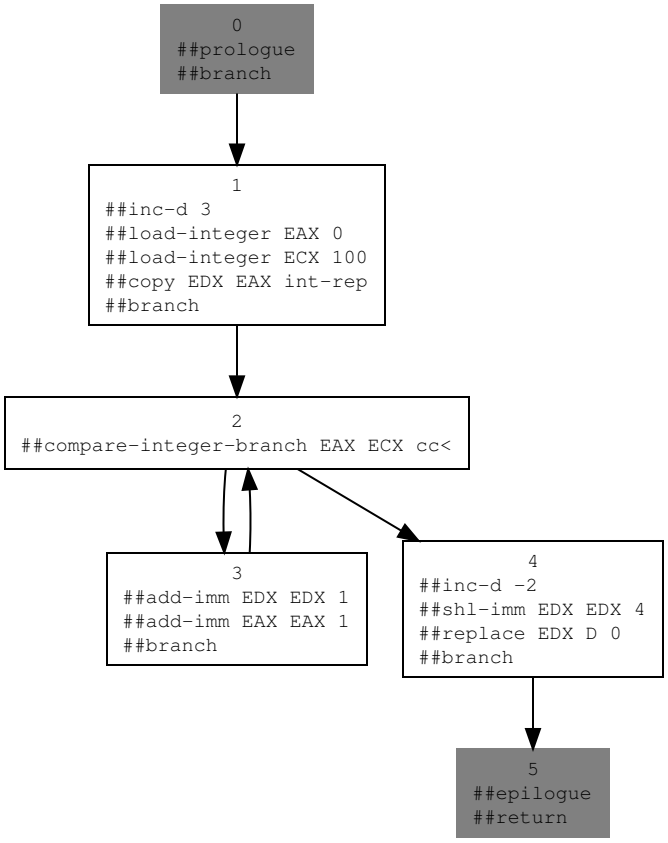

Figure 14: `0 100 [ 1 fixnum+fast ] times` after `finalize-cfg`

see that all of the `##copy` instructions have been removed and, for instance, the use of the virtual register `25` in block 2 has been replaced by `21`, since `25` was a copy of it.

Next, dead code is removed by `eliminate-dead-code`. Figure 13 on the previous page shows that the `##compare-integer` in block 2 and the `##load-integer` in block 3 were removed, since they defined values that were never used.

The final pass in Listing 26 on page 28, `finalize-cfg`, itself consists of several more passes. We will not get into many details here, but at a high level, the most important passes figure out how virtual registers should map to machine registers. We first figure out when certain values can be unboxed. Then, instructions are reordered in order to reduce *register pressure*. That is, we try to schedule instructions around each other so that we don't need to store more values than we have machine registers. That way, we avoid *spilling* registers onto the heap, which wastes time. After leaving SSA form, we perform a *linear scan* register allocation, which replaces virtual registers with machine registers and inserts `##spill` and `##reload` instructions for the cases we can't avoid. Figure 14 shows an example on an Intel x86 machine, which has enough registers that we needn't spill anything.