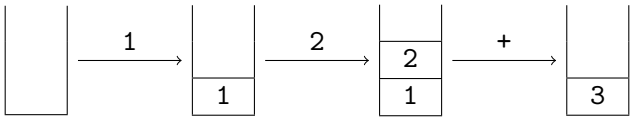

Figure 1: Visualizing stack-based calculation

1 Language Primer

citations for this history are fragmented across the internet; should consolidate some kernel of citation from it

Factor is a rather young language created by Slava Pestov in September of 2003. Its first incarnation targeted the Java Virtual Machine (JVM) as an embedded scripting language for a game. As such, its feature set was minimal. Factor has since evolved into a general-purpose programming language, gaining new features and redesigning old ones as necessary for larger programs. Today's implementation sports an extensive standard library and has moved away from the JVM in favor of native code generation. In this section, we cover the basic syntax and semantics of Factor for those unfamiliar with the language. This should be just enough to understand the later material in this thesis. More thorough documentation can be found via Factor's website, `http://factorcode.org`.

1.1 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 (being named by contiguous non-whitespace characters) are called *words*. 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.

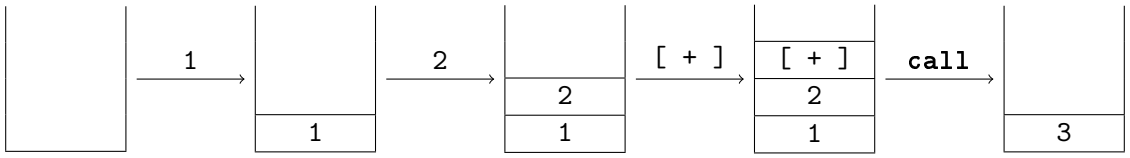

Figure 2: Quotations

- 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 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, sequence 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
```

Listing 1: Sequence 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 **`??`**.

1.2 Stack Effects

Everything else about Factor follows from the stack-based structure outlined in Section 1.1. 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 2 inputs.

For the purposes of documentation, of course, the names in stack effects do matter. In particular, the values 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 illustrated 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

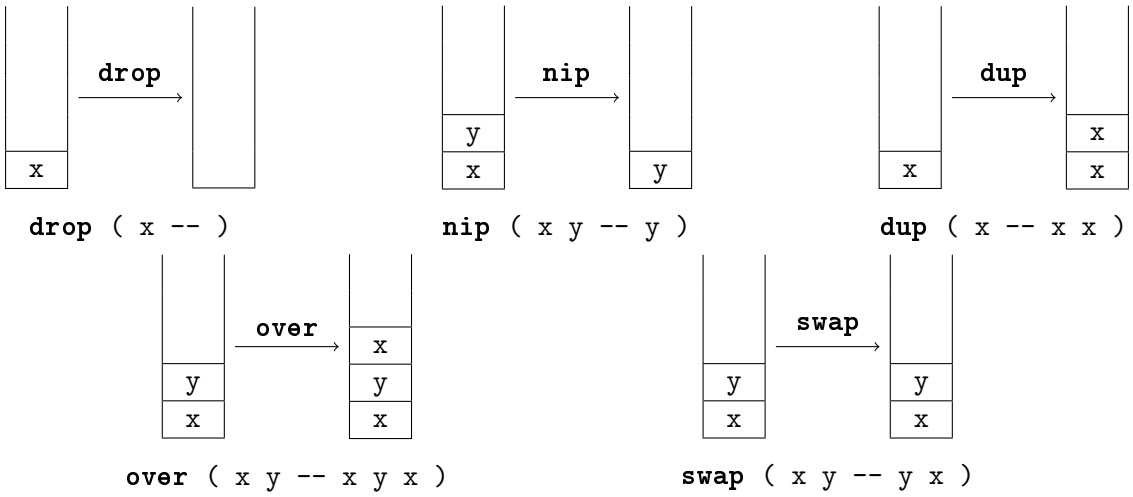

Figure 3: Stack shuffler words and their effects

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 3 permutes the top two elements of the stack, reversing their order. The stack effect `( x y -- y x )` indicates as much, since the left side denotes that two inputs are on the stack (the top is `y`, the next is `x`), and the outputs are the same, but swapped (so 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 keep track of more than a couple items on the stack at a time. We'll see how more complex data flow patterns are handled in **??**.

1.3 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 1.2, 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

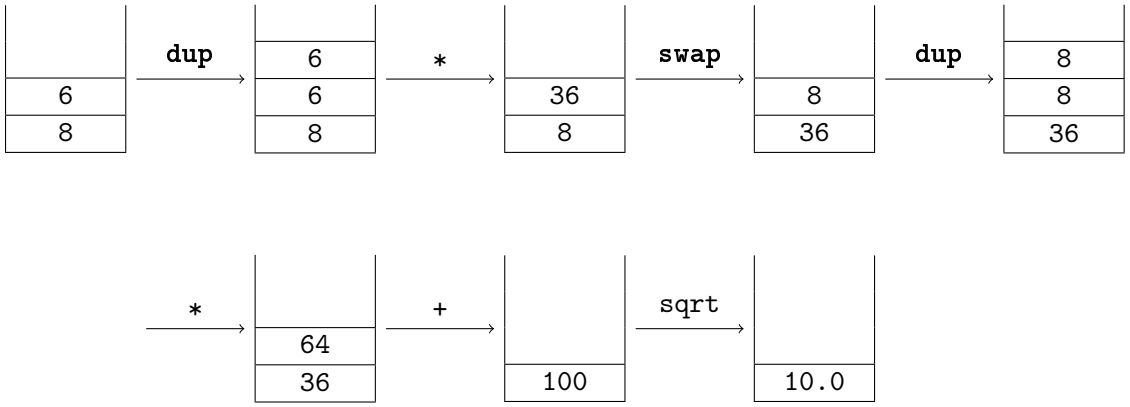

Figure 4: `norm` example

the definition. Thus, the classic example in Listing 2 on the previous page 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 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 actions, 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. Since concatenating any two programs denotes the composition of the functions that the programs individually denote, Factor is said to be a *concatenative* language. Taken together, these properties make 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.

cite Joy

```
: ^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.

You may have noticed that the examples thus far have not used type declarations. While Factor

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

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

Listing 5: Basic tuple definition syntax

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.

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, it defaults to the **`tuple`** class.

Slots are specified in several different 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 setter word naming template, 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".

```
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 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 the `utf8` word from the `io.encodings.utf8` vocabulary, corresponding to a particular Unicode format.