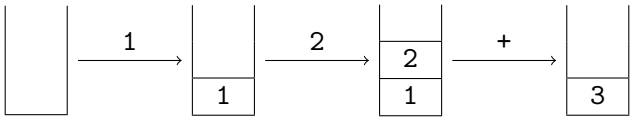

Figure 1: Visualizing stack-based calculation

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.

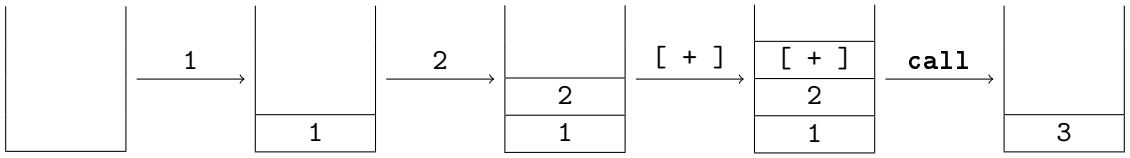

Figure 2: Quotations

- 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.

1.3 Concatenative Programming

The biggest step needed to understand Factor isn't really big: everything else follows from the stack-based structure outlined in Section 1.1. With some input on the stack, consecutive words transform the data in discrete steps, thereby shaping a result. This breeds a fairly stylized way of writing code that resembles a pipeline. Postfix code like `1 2 +` often seems to read "backwards", but when viewed in this pipelined fashion, it can read more clearly than standard syntax. For instance, suppose we're given a value `x` to manipulate by applying three functions, `f`, `g`, and `h`, in order. Typical function application notation looks like `h(g(f(x)))`, which must be read from the inside out because of the nested function calls. In Factor, we'd write `x f g h`, which is flat and reads left-to-right. Having such a whitespace-centered syntax reduces noise: instead of grouping arguments, the order of evaluation is the order you write your code.

In a way, this makes whitespace equivalent to function composition: just treat every word as a function from stacks to stacks—"before" to "after". Even literals (numbers, strings, etc.) can be

thought of as functions that take in a stack and return that stack with an extra element pushed onto it. Since concatenating any two programs denotes the composition of the functions that the programs individually denote, Factor is said to be a *concatenative* language. Note that being stack-based is unnecessary for being concatenative, though it's common.

cite and talk about Joy

This functional interpretation is useful for formally reasoning about concatenative languages, but it also highlights how Factor parallels more traditional functional languages. In particular, languages like Haskell encourage *point-free style*, wherein calculations are performed using function composition rather than application. Using this idiom, code is more compact and references fewer variables (the "points" of point-free style). Since data is passed around the stack in Factor, there's no need for variables—point-free style is the default. In the absence of named values, new operators spring up to name different patterns in code. These make data flow explicit, since you can't use intermediate variables or tricks of syntax as a crutch. Such operators are discussed in **??**.

list other concatenative languages

cite Joy

cite

Taken together, concatenative idioms emphasize shorter, clearer code. Since data flow is explicit, complicated patterns become painfully evident. There's little syntactic noise, so repetitive bits of code are easily recognized. Because whitespace is composition, these complicated, repetitive sections of code are easily "factored" (extracted) into new word definitions, hence the very name of the Factor language. By contrast, an applicative program laden with variables and syntactic noise requires 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 produce readable programs.

- Stack effects
  - Basic stack effects: stack shufflers illustrated
  - Complex stack effects: row polymorphism & types
  - Stack checker
- Combinators
  - Control flow
    - if
    - each
    - while
  - Data flow
    - Dip/keep
    - Cleave
    - Spread
    - Apply
- Object system
  - tuples
  - generics & methods

- Libraries & metaprogramming
  - Results of evolution
  - locals?
  - fry?
  - macros?
  - functors?
  - ffi?