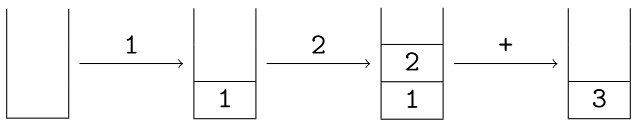

Figure 1: Visualizing stack-based calculation

## 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.

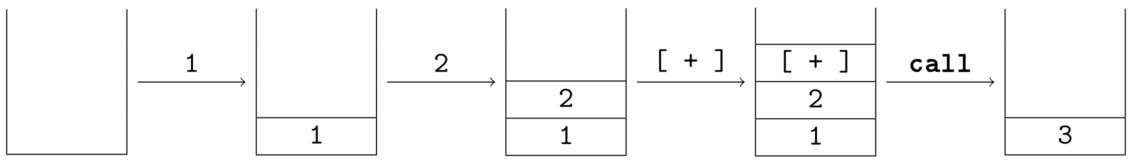

Figure 2: Quotations

- 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 of the two original elements, sandwiching `y` between two `x`s. Other shuffler words that duplicate

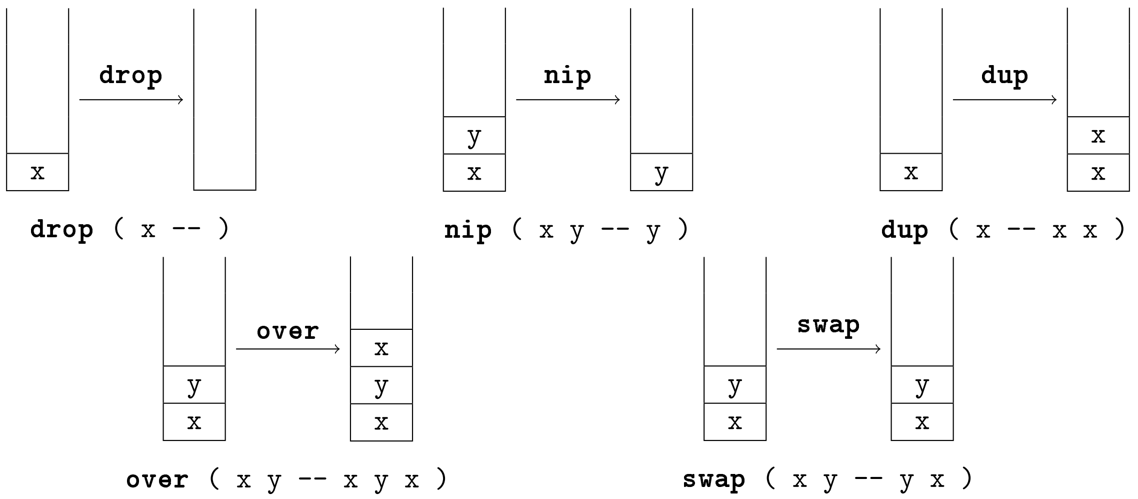

Figure 3: Stack shuffler words and their effects

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

We've now covered enough material to see how to define words.

- 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?