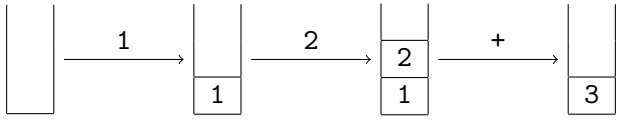

Figure 1: Visualizing stack-based calculation

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.

figure out section / chapter names

cite chapter?

cite factor-code.org

1.1 Stack-Based Programming

Like Reverse Polish Notation (RPN) calculators, Factor's essential evaluation model uses a global *stack* upon which operands are pushed before operators are called. This lends itself to *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. Parsing becomes very flexible, since whitespace is essentially the only thing that separates tokens. In the Forth tradition, functions (being single tokens delineated by whitespace) 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.
- Otherwise, scan ahead for a single token.
  - If the token is the name of an *ordinary word*, it's added to the parse tree.
  - If the token is the name of a *parsing word*, it's invoked with the parser's current information.
  - Otherwise, try to parse the token as a numeric literal.

Notice too that evaluation (as in Figure 1 on the previous page) simply goes left-to-right: literals are pushed to the stack, words are invoked.

Ordinary words correspond to functions. While infix languages tend to distinguish syntactically significant operators (e.g., for arithmetic) from normal user-defined functions, there is no such difference here. Since `1 2 +` and `1 2 foo` are tokenized the same way, other languages' special operators are simply words in Factor.

More ordinary word examples

Parsing words serve as hooks into the parser, letting Factor users extend its 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).

Other parsing words act as sided delimiters. The parsing word for the left-hand delimiter will scan ahead for the right-hand delimiter, using whatever was read in between to add objects to the parse tree. For example, array literals are created by the parsing word `{` parsing ahead until it finds a `}` token, collecting the results into an array object that's added to the parse tree. As it parses recursively, parsing words and literals in between the braces are collected while ordinary words are not called, since the tokens are collected at parse time.

- Basic syntax & semantics
  - Example ordinary words: stack shufflers
  - Example parsing words: quotations
- Concatenative programming
  - "Pipeline code"
  - Whitespace = function composition
  - Point-free style
  - Origin of name "Factor"
- Stack effects
  - Basic stack effects: stack shufflers illustrated
  - Complex stack effects: row polymorphism & types
  - Stack checker
- Control flow
  - Combinators
    - if
    - each
    - while
  - Dataflow combinators
    - Dip/keep
    - Cleave

        * Spread
        * Apply

- Object system
    - tuples
    - generics & methods
- Libraries & metaprogramming
    - Results of evolution
    - locals?
    - fry?
    - macros?
    - functors?
    - ffi?