An Introduction to Factor

Zackery L. Arnold
Department of Electrical and Computer Engineering
University of Dayton
Dayton, Ohio 45469–0232 USA
arnoldz3@udayton.edu

ABSTRACT

We provide a brief overview of Factor—a stack-based and concatenative, object-oriented programming language.

KEYWORDS

Concatenative language, higher-order functions, stack language.

Zackery L. Arnold. 2017. An Introduction to Factor. *CPS 499-03: Emerging Languages, University of Dayton, Ohio 45469–0232 USA, Spring 2017,* 4 pages.

1 INTRODUCTION

Factor (http://factorcode.org/) is a uniquely blended concatenative programming language that combines the core concepts of object-oriented programming (e.g., classes, polymorphism) with the flexibility and power of the functional programming paradigm. Factor programs are written using chains of higher-order functions that manipulate data on a data stack. Programs are developed using the interactive development environment called the Factor Listener. Code written by the user is immediately compiled by Factor's optimizing compiler and stored within the currently operating image of the language mixing the ease of experimentation of interpreted languages with the performance of a compiled one. With all of these features in place and rich libraries of useful functions Factor presents itself as an emerging language worthy of wider consideration.

2 CONCATENATIVE PROGRAMMING

Concatenative programming languages, in the tradition of their predecessor language Forth [2], are hallmarked by the composition of multiple functions that cooperate to transform data. The concept of concatenative programming, including in Factor, greatly resembles the concept of *pipelining* in Linux/UNIX systems. Programs in Factor are made up of many higher-order functions that are written from left to right. Functions are identified using any sequence of non-whitespace characters, resulting in a line of code that nearly resembles a sentence in a natural language. Hence, in Factor, functions are typically referred to as *words* with whitespace serving as the *concatenation operator* [3]. The following is an example of concatenative Factor code that calculates `12!`.

```
12 [1,b] 1 [ * ] reduce
```

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
CPS 499-03: Emerging Languages, University of Dayton, Ohio 45469–0232 USA
© 2017 Copyright held by the owner/author(s).

3 THE DATA STACK

Factor is also a stack-based programming language. This means that all data in the program is loaded onto a shared data stack. Words that use this data pop off the stack and push back the results of the operation. Concatenated words, then, work together through the use of the shared stack to morph the input data into the desired output. It is worth noting that the shared data stack is entirely unrelated to the traditional concept of the runtime stack used in function calls and memory allocation [1]. Furthermore, the data stack itself is not directly related to a typical stack data structure, as the data stack does not encapsulate any methods like `push` or `pop`. Instead, the data stack is fully managed by the words used in the program.

3.1 Stack Shuffling and Combinators

One of the greater learning curves that comes with studying Factor is understanding how to properly manipulate the data stack. There are multiple words that are readily available to the programmer, such as `swap` (which swaps the two topmost elements), `drop` (which throws away the top element), and `dup` (which duplicates the top element). These words are vital to the success of more complicated Factor programs but make the code bulky and less readable. As a result, the programmer is encouraged to adopt more powerful higher-order functions that abstract some of the busy work associated with the language. Words in Factor that achieve this goal, such as `reduce` and `fold`, are called *combinators*.

One commonly used combinator is the word `bi`, which takes a single value off of the stack and applies two separate functions to the value, leaving two results on the stack. For example, `12 [ 1 + ] [ 1 - ] bi` will leave the two separate values `13` and `11` on the top of the stack. The word `bi` may also be used for the evaluation of an `or` word as in the program `[ even? ] [ odd? ] bi or`. Other useful combinators include `cleave` (which applies an arbitrary number of words to a given value), `each` (which applies a word to all items of a list), `filter` (which returns the elements of a list that pass a given filter), and `map` (which applies a word to a list to receive a new list). Fig. 1 illustrates some of these common stack shuffling and combinator words in action.

3.2 Stack Effect and the Stack Checker

Another challenge with programming in a stack-based language comes from the large amount of bugs that can arise when words that manipulate the stack leave a varying number of items on to the stack after completion. As a result, words defined in Factor are accompanied with an explicit declaration that shows how many items are taken from and placed on the stack. This declaration, called a *stack effect*, is notated with `( a -- b )`, where the words `a`

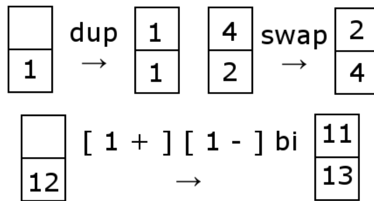

Figure 1: A visual representation of a few commonly used words in Factor code.

and b are representative of the *effect* that the word has on the stack. Stack effect declarations serve both a documentation purpose as well as a crucial part of compilation — words can only be compiled if they strictly follow the notation of their corresponding stack effect. Stack effects, in general, perform a simple role that is similar to pattern matching in other languages, such as Haskell. However, unlike Haskell, the words used to represent items on the stack are purely symbolic, meaning that they cannot be referenced in the body of the word like a lexically-scoped variable[1].

The *stack checker* is the name given to the tool that acts similar to a type system in Factor [3]. The stack checker performs a brief simulation of the program before passing it to the compiler—checking that each branch of control in the program leaves the stack at equal heights. If the word does not leave the stack with a consistent height, then a compile-time error is raised and the program is rejected. However, Factor does make some exceptions to this rule, specifically with regards to *row polymorphism.* In some situations the programmer may wish to use a combinator with sets of words that have different stack effects. Row polymorphism in Factor permits this, so long as the quotations are only operating on data *below* itself on the stack—resulting in the same stack height as the other words used in control flow.

4 QUOTATIONS AND VOCABULARIES

As mentioned above, functions in Factor are idiomatically called words. Words can be anonymously constructed from the composition of other words. Anonymous words are called *quotations* and are wrapped in the `[` and `]` *parsing words* that enclose the quotation. Table 1 illustrates some of the most common parsing words encountered in Factor code. Some words, such as `if`, take a quotation as an argument and use that quotation to transform the stack. Quotations can also be constructed using values already on the stack when using a pair of alternate words. For instance, `10 '[ _ = ]` results in the equality quotation `[ 10 = ]`.

Words can also be defined with formal names. New words are defined with a sequence of the beginning `:` parsing word, the given name to the word, the word's stack effect, the body of the word, and the enclosing `;` parsing word that marks the end of a definition. For example, consider the following word `product` which multiplies the contents of a sequence:

```
: product ( seq -- p ) 1 [ * ] reduce ;
```

[1]Factor does have library-support for lexically scoped variables, but they are unrelated to the notion of stack effect.

Table 1: Commonly encountered parsing words.

Parsing Word	Semantics
`: ... ;`	Denotes the start and end of a named word.
`( ... -- ... )`	Denotes a word's stack effect.
`[ ... ]`	Denotes a quotation or anonymous word.
`{ ... }`	Denotes an array or vector sequence.
`!`	Denotes a comment until end of line.

Multiple words like `product` above can be defined in a collection known as a *vocabulary*—the Factor analog of a library. Vocabularies are stored within a source code file that rests in a hierarchal subdirectory structure within the Factor installation directory. Each vocabulary has to explicitly reference the vocabularies that it relies upon for operation. For example, a separate vocabulary that wishes to use the `product` word above would need to include the vocabulary in which product is defined. Explicit declaration of necessary vocabularies allow Factor's compiler to ensure minimization of the compiled code. Compiled Factor programs only include the bare minimum code needed for successful program execution.

5 OBJECT-ORIENTED IMPLEMENTATION

Factor is a *purely* object oriented programming language. This means that every value used is an object. Classes are divided into three primary types: *primitive classes*, *tuple classes*, and *derived classes* [3]. Primitive classes cannot be subclassed and are used for primitive classes like strings, numbers, and words. Tuple classes are more advanced and may contain instance variables that, unlike generic words, are owned by the class. Derived classes are built from other preexisting classes and can take multiple forms. For example, predicate, union, and intersection classes can be created in the same way one might interpret sets from set theory. A special example is a *mixin* class that is used often in Factor to collect groups of classes under a common interface [3].

Another unique feature of Factor is the way in which objects and object-specific words interact. In many object-oriented languages a given object has direct access to methods that are intrinsic to that object. Factor instead uses *generic words* that are redefined as needed to handle a given object-specific implementation—serving a purpose similar to template class functions in other languages, but with less restrictive implementation due to the absence of the concept of ownership.

6 FACTOR'S DEVELOPMENT WORKFLOW

Programming in Factor necessitates familiarity with Factor's interactive development environment—the *Factor Listener.* The Listener provides an interpretive playground for the programmer to test new ideas and words on the fly without the use of dedicated source code. When the programmer is ready to commit to a source file, they can create a new vocabulary using the `scaffold` tools vocabulary and the `scaffold-work` word. A new directory is created with an empty source file and ready for the programmer. Once the file has been successfully edited, the programmer can update Factor automatically with the new vocabulary by using either the `F2` key or the `refresh-all` word.

Other convenient features are provided by the Factor IDE. Unit tests can be defined and automatically evaluated with the use of the `unit-test` vocabulary. Documentation files can be created with the use of the `scaffold-docs` vocabulary and multiple words from the auto-included `help` vocabulary. In addition to user documentation all of the Factor documentation is available from within the IDE through the help button at the top of the interface. Word execution can be benchmarked at any time with the use of either the `CTRL + T` keystroke or the use of the `time` word. Because Factor uses an image-based compilation system, the `save-image` word can be used to save the state of the current Factor instance to a file. Images can then be loaded when the IDE is launched, resulting in a faster development experience without the need of reloading key vocabularies for every launch. Factor is also capable of deployment as a standalone executable with support for Windows, Linux, and Mac OS, through the use of the `deploy` word. Moreover, Factor itself also has many capable vocabularies that implement concepts from foreign function interfaces and UI toolkits to direct HTTP and SMTP support [3].

7 EXERCISES

The following are some programming exercises that incorporate some essential Factor concepts:

(1) Define a word `caesar` in a vocabulary `homework` that takes a string of alphabetical characters and an integer and applies the integer to each character in the string. Your solution must handle both positive and negative offset values. Only include uppercase and lowercase alphabetical characters in the output. For example, `ABCD` with an offset of 4 should produce `EFGH`. Factor your solution into a primary word `caesar` and set any helper words as private. This problem can be solved in less than 10 lines of code.
Examples:

```
> "SEESPOTRUN" 26 caesar .
"seespotrun"

> "SEESPOTRUN" -26 caesar .
"seespotrun"

> "ABCDEFG" 7 caesar .
"HIJKLMN"
```

(2) Define a new tuple class `novel` that represents a fictional literary work. Include member variables that correspond to the novel's title, author, genre, publisher, year of publication, and an identification number. Use strings for the first four variables and integers for the last pair. Include a constructor `<novel>` that takes values for each variable as arguments and sets them automatically. Also include a word `book-print` that takes a novel as an argument and prints the novel's details in a simple format.
Examples:

```
> "Narnia" "C. S. Lewis" "Fantasy"
  "Geoffrey Bles" 1952 1 <novel>

--- Data stack:
T{ novel f "The Lion, the Witch,
and the Wardrobe" "C. S. Lewis"
"Fantasy" "Geoffrey Bles" 1950...

> book-print
Class: Novel
Title: The Lion, the Witch, and
the Wardrobe
Author: C. S. Lewis
Genre: Fantasy
Publisher: Geoffrey Bles
Year: 1950
ID: 1
```

(3) Extend the solution to problem 2 to two new classes of books—`textbook` and `article`. For textbooks, change the genre field to subject. For article, change the genre field to discipline and add journal and volume fields. Then, with the three classes, define a `mixin` class called `library` that will represent the union of different types of books. Adjust the `book-print` word from before to be generic with templates for each type of book. Publish the `library` and each class in a `library` vocabulary.
Examples:

```
> USE: library
> "The C Programming Language"
  "Ritchie, D. and  Kernighan, B."
  "Computer Science" "Prentice
  Hall" 1988 2 <textbook>

--- Data stack:
T{ textbook f "The C Programming
Language"...

> book-print
Class: Novel
Title: The C Programming Language
Author: Ritchie, D. and Kernighan, B.
Subject: Computer Science
Publisher: Prentice Hall
Year: 1988
ID: 2

> "Factor: A Dynamic Stack-based
  Programming Language"
  "Pestov, S." "Computer Science"
  "ACM SIGPLAN Notices"
  45 "ACM Press" 2010 3 <article>

--- Data stack:
T{ article f...
```

```
> book-print
Class: Novel
Title: Factor: A Dynamic Stack-based
Programming Language
Author: Pestov, S.
Discipline: Computer Science
Journal: ACM SIGPLAN Notices
Volume: 45
Publisher: ACM Press
Year: 2010
ID: 3
```

8 CONCLUSION

The strengths of both object-oriented and functional languages are blended in Factor. Abstract classes, higher-order functions, and Factor's expressive syntax give the programmer the flexibility needed to solve many complex problems in an elegant way. Factor's wide set of mature vocabularies and features allow the programmer to focus on the greater details of a given problem. If the curious programmer is able to overcome the learning curve that comes with writing concatenatively they will find a rich and rewarding language awaits them. In summary, or if you will, in a word:

```
: Factor ( -- is ) "Object Oriented" "Functional Execution" + ;
```

REFERENCES

[1] Concatenative language. Available: http://concatenative.org/wiki/view/Concatenative%20language [Last accessed: 24 March 2017]. (????).

[2] P. Koopman. 1993. A brief introduction to Forth. In *Proceedings of the Second ACM SIGPLAN conference on History of Programming Languages*. ACM Press, New York, NY, 357–358. Also appears in *ACM SIGPLAN Notices*, 28(3), 1993.

[3] S. Pestov, D. Ehrenberg, and J. Groff. 2010. Factor: A Dynamic Stack-based Programming Language. In *Proceedings of the Sixth Symposium on Dynamic Languages*. ACM Press, New York, NY, 43–58. Also appears in *ACM SIGPLAN Notices*, 45(12), 2010.