Background Report: Factor as a Practical Language for Software Applications

Reggie Gillett
27133123
w6k8
reggie.gillett@gmail.com

Graham St-Laurent
23310121
i5l8
gstlaurent@gmail.com

Lynsey Haynes
12686119
a4h8
lynseyahaynes@gmail.com

Brittany Roesch
22015119
r1d8
brroesch@gmail.com

Gord Minaker
34615112
u4s8
gordonminaker@hotmail.com

ABSTRACT

Factor is a concatenative and object-oriented programming language. This paper explores some of the unique features of contatenative programming languages and Factor in particular. We investigate features of Factor necessary for the implementation of an interactive game, and discuss the implications of their use.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages

Keywords

Factor, stack-based languages, concatenative languages

1. INTRODUCTION

Our discussion centers around an exploration of Factor as a programming language. The impetus of our exploration will be an evaluation of Factor's elements and features in the context of developing a computer game. We will focus on what makes Factor unique and highlight aspects with particular value to the creation of a substantial programming project.

1.1 Related Languages

Our motivation for studying Factor stems from three principal goals: 1. We want to explore a stack-based language with postfix syntax to see if it provides an effective way to describe and solve problems, 2. we want to use a new language that implements features from more recent language research (e.g., the increasing trends towards functionalism), and 3. we want a language that is intended to solve arbitrary real-world problems—that is, a practical, full-featured language. Factor satisfies these goals, and in this section we describe how it relates to other languages in its family and why they weren't chosen. Specific details relating to terminology and syntax will be explained in later sections.

Factor's two closest relatives are Joy[13] and Forth[10], which are both stack-based languages using primarily postfix syntax. Forth is an older language, first appearing in the late 1960s, and intended, by its creator Charles Moore, to be a tiny portable language.[21] Because of this, Moore chose a very simple, sequential (as opposed to generating a tree), whitespace-based parse/interpret mechanism, supported by implicit passing of typeless data via a *parameter stack*. In Forth, functions are called *words*; here's an example of a word that squares a number and adds one to the result, followed by its application to a stack whose top element is a 3 (thus leaving a 10 on the stack):

```
: squareinc ( n -- n' )   dup * 1 + ;
3 squareinc
```

Factor has adopted much of its syntax and terminology directly from Forth. Here is the same word and application in Factor:

```
: squareinc ( n -- n' )   dup * 1 + ;
3 squareinc
```

It hardly takes an astute reader to notice that the two examples are exactly the same. In spite of that, there is one syntactic difference, but it is invisible: the parenthetical *stack comment* in Forth is just that—a comment, thereby optional and ignored by the compiler—whereas Factor statically verifies that its words affect the stack in the way the programmer indicates; in Factor, the parenthetically-delimited *stack effect* is required. That is an example of how Forth, although a powerful, stack-based language, is old and minimal. It does not natively provide any helpful static checking, nor does it provide list or closure mechanisms, and it dangerously splays out memory addresses so explicitly it would make a seasoned C-programmer blush. It is not a language informed by the latest researches in programming so it is insufficient for us.

Joy, on the other hand, is a research language created by produced by Manfred von Thun dating from the turn of the century.[23] Not only is it stack-based and concatenative, but it is also a purely functional programming language. It represents a new functional paradigm: *compositional* functionalism based on function composition, as opposed to the well-known functional paradigms based on the lambda calculus; von Thun calls those *applicative* functionalism. Joy provides native list mechanisms; it is so pure that it has no state nor environment; and it makes extensive use of

quotations, which, as discussed more deeply in later sections, are essentially anonymous functions and can be passed via the stack just like any other object. Passing numerous quotations around requires a rich vocabulary of what von Thun calls *combinators*—functions that manipulate quotations. For example, the `ifte` combinator ("if-then-else"), takes three quotations from the stack, applies the first (i.e., bottom-most), and, based on the resultant stack, applies one of the other two quotations. Here's an example that uses `ifte` in the definition of a function that will be familiar to many readers:[24]

```
factorial  ==
    [0 =] [pop 1] [dup 1 - factorial *] ifte
```

Factor inherits the combinator/quotation pattern from Joy, although it implements its `if` in a slightly different way: [1]

```
: factorial ( n -- n' )
   dup 1 = [  ] [ dup 1 - factorial * ] if ;
```

(Note how Joy does not rely on whitespace to parse brackets, whereas Factor, in the Forth tradition, does. We revisit this designer's decision in the discussion at the end of the paper.) Ultimately, although Joy does provide fascinating new concepts in modern language development, it is still just a research language, and doesn't have the full-featured library needed to implement an arbitrarily-broad program like ours. So it is also insufficient for us. (Note also that, unlike Joy, Factor *does* provide state and environments, as well as semantics for several different types of closures.)

Nevertheless, at Joy's heart is a thesis that defines what it means to be a *concatenative language*:

> The concatenation of two programs denotes the composition of the functions denoted by the two programs. [25]

Factor has entirely adopted this philosophy. It is less applicable however to Forth as well as another older postfix language: PostScript.[1]

PostScript[20], introduced in 1985, is probably the most common stack-based, postfix language in use today, but it is written and read almost exclusively by machines; it is the language of printers and PDF documents. Nevertheless, it is a Turing-complete language, but, designed t handle practical efficiency constraints like Forth, its designers likewise to use chose a stack-based, point-free, and postfix model. The biggest difference here though is that its problem domain is almost exclusively that of graphical document description; it is a very specialized language so it is not rich enough for our project.

PostScript is not in the same family as the other languages we have discussed, in spite of each language's postfix and stack-based interpretive aspect. For example, here's how squareinc and its application could be coded in PostScript:

```
/squareinc {
    dup mul 1 add
} def
3 squareinc
```

Unlike the other three, PostScript even uses postfix syntax for function definition!

2. FACTOR'S CONCATENATIVE FEATURES

2.1 Stack Paradigm

A stack-based language uses an operand stack to pass function arguments and return function values.[22]. Literals, like numbers and strings, can be thought of as functions with no arguments that push themselves onto the stack. Operators and functions are applied to values already pushed onto the stack. The stack gets used in place of, and possibly in addition to, named variables. Thus function calls are referred to as "words", since they are not followed by their arguments.

2.1.1 Postfix Syntax

Factor uses a postfix syntax, with operands being written first, followed by their operator or word. The order of operations is not held, rather values and operators are evaluated in the order in which they come. The combination of an operand stack and postfix syntax can make algebraic expressions look complicated and unintuitive.[1] For example, the relatively simple mathematical expression:

$$f(x, y, z) = y^2 + x^2 - |y| \tag{1}$$

could be written as:

```
f = drop dup dup * swap abs rot3 dup * swap - +
```

[1] in a postfix language such as Factor. Despite its initially steep learning curve, postfix syntax combined with a stack base are easy for a computer to compile and generally have good performance. And once accustomed to it, according to Factor creator Slava Pestov, it becomes quite readable. In his words:

> Learning a stack language is like learning to ride a bicycle: it takes a bit of practice and you might graze your knees a couple of times, but once you get the hang of it, it becomes second nature.[8]

2.1.2 Dataflow Combinators

The stack manages the data flow of the program.[9] Since all operators work on the items at the top of the stack, dataflow combinators can rearrange or remove elements on the stack as the programmer needs. They can also be used to perform an operation while keeping its input intact, or apply a single operator or sets of operators to multiple values.[3] For example, the dip combinators will ignore a specified number of values on the top of the stack to invoke an operator further down on the stack:

```
dip2 (x quot -- x )
```

[1] For the sake of completeness, here's how factorial could look in Forth, which does not have combinators:

```
:  factorial ( n - n' ) dup 0= if drop 1 else dup
1- recurse * then ;
```

And reciprocatively, here's how `squareinc` could look in Joy:

```
squareinc == dup * 1 +
3 squareinc
```

This hides the first element of the stack temporarily to apply the second element. The most prevalent dataflow combinators in Factor are dup (duplicates a stack element), drop (pops a stack element) , and swap (exchanges the first and second elements).[7] Dataflow combinators would no doubt be essential to a game built in Factor.

2.2 Lexical Scope

. In a lexically-scoped language, the range of functionality of a variable may only be called or referenced from within the block of code in which it is defined. This means the analysis can be done statically. Lexical resolution is determined at compile time and is also known as early binding. Most modern languages today are lexically scoped, as they make the language more readable and easier to understand, as well as removes the unpredictability of dynamic scoping. There are still a few dynamically scoped languages in common usage though; PostScript, the programming language which runs on printers, is perhaps the most commonly used of them.[18] Factor initially did not provide support for lexical scoping, but it has since been implemented as default. Adopting lexical scope enables two major changes in the programming language. It makes it easy to construct function closures and to construct objects with mutable state.[12]

2.3 Quotations as Anonymous Functions

In Factor, a quotation is enclosed within square brackets and contains a sequence of literals and words. The following example demonstrates that the quotation is passed as the single object onto the stack.

```
[ "hello" . ]

--- Data stack:
[ "hello" . ]
```

Because the quotation itself is pushed onto the stack as a quotation, it demonstrates inferred evaluation and behaves similar to an anonymous function. As such, quotations do not keep track of the environment. In Pestov's words:

> Quotations do not close over any lexical environment; they are entirely self-contained, and their evaluation semantics only depend on their elements, not any state from the time they were constructed. So quotations are anonymous functions but not closures.[5]

Only when a word retrieves a quotation from the stack is when it will be exectued and evaluated.

2.4 Lexical Closures

A lexical closure is a function that can have free variables which are bound to a value by the environment in which the function is defined.

To define a word which uses lexically scoped variables, you use `::` to define a word which internally performs all the necessary rewriting to make closure conversion and lexical scoping work.[22] (Normal function definition uses single semicolon `:`, instead.) You follow the word name with a list of locals enclosed in pipe ('|') characters and use these locals anywhere in the body of the word. Normally, the names of input stack parameters (parameters in which they are retrieved from the stack) have no meaning within the scope of the word. However, a word with named parameters makes those names available in the lexical scope of the word's definition.

The following is an example word using locals:

```
:: add-test | x y z | x y + z + ;
1 2 3 add-test .
  6
```

[6] Once the word 'add-test' is called, the parameters x, y and z will be bound to a value. This means that 'add-test' has become a closure since all of its free variables (x, y and z) are now closed (ie. bound to a value) by the environment in which 'add-test' was defined.

Also within the scope of a :: definition, additional lexical variables can be bound using the :> operator, which binds either a single value from the data stack to a name, or multiple stack values to a list of names surrounded by parentheses.[3] For example, if you were to provide the following code within a word body,

```
1 2 + :> x
```

[6] the name 'x' would be bound to the value 3 and is in scope within the body anywhere after the initial binding.

All locals are stored on the retain stack, which is separate from the data stack. This means that internally it forms all of the inputs of a lambda into an array, which is then moved to the retain stack. Named parameter accesses are rewritten into the code which moves this parameter array to the data stack, pulls out the right element and moves it back to the retain stack.[5]

In a stack-based language, lexical variables, and in particular lexically-scoped closures are a useful extension of the concatenative paradigm. They can particularly become helpful when there is no obvious solution to a problem in terms of re-arranging operands at the top of the stack [22].

2.5 Point Free Style

As stated above, Factor provides local variables, but they are used in only a small minority of procedures because its language features allow most code to be comfortably written in a point-free style. Point-free style is utilized by all known concatenative languages and is a programming paradigm in which function definitions do not identify the arguments (or "points") on which they operate. Instead the definitions merely compose other functions, among which are combinators that manipulate the arguments.[15]

For example, consider this sequence of operations in a hypothetical applicative language (that closely resembles Python):

```
def example(x):
   y = foo(x)
   z = bar(y)
   w = baz(z)
   return w
```

Another way of writing that emphasizes the function composition:

```
def example(x):
    return baz(bar(foo(x)))
```

Notice how `baz` is the last thing that function to get called, but it occurs first—i.e., it is the leftmost. Now let's see how the same series of functions would look in Haskell using point-free style and function composition:

```
example = baz . bar . foo
```

Notice the lack of parameters (the `.` is Haskell's way of signifying composition), as well as the fact that the lastmost function to be applied is the first one to appear. This can intuitively be understood as saying "`example` is the composition of `baz`, `bar`, and `foo`." In a concatenative language like Factor, however, this same sequence would look something like this:

```
: example   foo bar baz ;
```

Once again, no parameter appears so it is the function composition that is emphasized. But in this case, the functions appear in the order that they are called, so the intuitive reading is simply "`example` is a `baz`, a `bar`, and a `foo`", which is a subtly different perspective. We find that this postfix perspective emphasizes the process, whereas the other, prefix perspective emphasizes the product as a whole.

Regardless of the perspective, however, there are clear benefits to implementing code in point-free style. One advantage is that it allows the programmer to be succinct. It also allows the user to think of what the code is doing with the image of *composing functions* rather than imagining the process of arguments flowing through.

3. FEATURES REQUIRED FOR EFFECTIVE SOFTWARE DESIGN

3.1 Control Flow Structures

Central to the programming of any game are control flow structures. Loops and conditional expressions are integral to the construction of any game, simple or complex. For example, in virtually all games there exists code to interpret user input. Typically this consists of a single or series of if statements which execute particular actions based on a key pressed by the user.[11] For even the simplest games this is fundamental to their implementation. Take for an example a simple number guessing game, at some point the program will have to evaluate if the number entered by the user matches the number to be guessed. This control pattern is often included in a *game loop*, another essential piece of most games rooted in control flow.[11]. The *game loop* executes continuously processing user input, updating the game state and rendering the game on each turn.[11]

All control flow structures in Factor are expressed using higher-order functions, functions which take functions as arguments. In the tradition of other stack-based languages such as Joy, these are referred to as *combinators*. Combinators operate like any other word in Factor, taking their arguments from the stack, however unlike other words, these arguments are quotations rather than literals.[22] The previously-mentioned example of a number guessing game gives the example illustrated below, which illustrates the use of the if combinator:

```
: guessed-number ( n -- )
   secret-number  =
   [ "You guessed it!" print ]
   [ "Nope, that's not it" print ]
   if ;
```

It assumes our secret number is a constant defined elsewhere. We can see that `if` takes 3 arguments, a boolean and two quotations, one that will be evaluated in the true case, and one that will be evaluated in the false case. Any object in the place of the boolean other than the special object f (Factor's representation of false) evaluates to true[2]. In this case we are performing a comparison between our secret number and the guessed number using the equals operator. This will evaluate to a boolean which will act as the first argument of our call to `if`. We then provide two quotations, one for the true case which prints "You Guessed it!" and one for the false which prints "Nope, that's not it".

All control Flow in Factor is achieved using the if combinator and recursion.[22]. Factor contains several looping combinators in its standard library which give the effect of iteration (`each`, `filter`, `map`, etc). All of these use recursion or branching and recursion to achieve their effects.[7]

3.2 User Interface

Any game will require the user to interact with the machine, so there will always be some sort of user interface. Factor provides a vocabulary implementing a cross-platform, object-based Graphical User Interface,[16] so using Factor for our game should be relatively painless.

In Factor's User Interface vocabulary, graphical control elements are of the type `gadget`, with the outermost gadget—the window itself—being subtype of gadget called a `world`. Ultimately, the world is simply the topmost node in a tree of gadgets. The UI is implemented using OpenGL, and Factor does provide a vocabulary with bindings to OpenGL, so full-featured 3D graphics are possible, as well as more basic 2D graphics. Although the gadget class is open-ended and there are many developer options for the UI, it does come with some basic types such as the `label`, `button`, or `editor`. Depending on time, we may try to implement some graphics, but simply providing a text area to read from, and an area that includes text output may be all we get to do.

Consequently, since Factor also provides built-in words that access *standard output* and *standard input*, instead of taking a GUI route, we may alternatively end up creating a text-based console application.

3.3 Unit Testing

In the development of any non-trivial software application, unit testing is a necessary first step to ensure error-free code. A programming language which lacks a unit-testing framework should be quickly forgotten. Factor has a fully adequate unit-testing framework so that testing can begin quickly without installing or configuring external frameworks.

By simply typing "<*vocabulary-name*> `scaffold-tests`" into the listener, a test file is created. Within the test file,

tests are typically written in the following format:

```
[<expected-output>] [ <input> ] unit-test
```

Tests can be run at any time from the listener by typing "<*vocabulary-name*> `test`". Failing tests are able to be debugged using the built-in error list tool.[17]

3.4 Object Orientedness

Factor is an object oriented programming language whose object system is unique from other common languages. In Factor, every value is technically an an object which can be pushed onto the data stack. An object, like other languages, represents an instance of a class. [4] Each object holds a type value and multiple fields, which are stored within words. Here we will consider the definition, instantiation, and modification of objects in Factor.

3.4.1 Object Class Definitions: Tuples & Slots

Classes in Factor are called Tuples which hold fields called 'slots' belonging to the class. Like other OO languages, classes can inherit from specified superclasses and the "final" designation is used for classes which should not be further subclassed. A typical class definition in Factor is written in the format:

```
TUPLE: <class-name> <slot1> <slot2> ... ;
```

As an example:

```
TUPLE: student sid name gpa ;
```

Defines a new class named 'student' which has fields, termed "slots", for the values of sid, name, and gpa. Unlike other object systems, methods cannot belong to classes in Factor. [2]

3.4.2 Object modification - Methods

Similar to the Common Lisp Object System (CLOS), methods in Factor cannot belong to classes. Rather, a structure termed a "generic word" defines zero or more methods all with the intention of performing the same functionality, normally using a different implementation for each specified input class type. When a generic word is referenced in the program, it will execute the method whose signature is most appropriate for the input to the call.[19]

4. DISCUSSION

Factor has many interesting features that make it a very unique language, although many of its finer points are irrelevant for our game development. The operand stack combined with postfix syntax allows for the code to be compiled very quickly. This is not as essential now as it was 50 years ago, since hardware is much cheaper and faster today. Factor's unit testing, user interface, control flow structures, and object orientedness are the most essential part of our game, yet these aspects are not what makes Factor different from other languages.

However, the stack-based paradigm, postfix syntax, and point free style all make Factor's code extremely difficult to read – simple functions seem unintuitive, especially when combined with dataflow combinators. Whitespace delimiters are mandatory between all symbols, leading to many simple yet annoying compiler errrors. Perhaps given more time, we would begin to appreciate the elegance of Factor's backwards way of aligning variables and remembering what is on the data stack; but for now, we see Factor as a "write only" language – one that is an enigmatic challenge to write code for, but an ugly mess to read someone else's code.

5. SUMMARY

While the unique syntax of Factor might serve initially as a deterrent to adopting the language for more ambitious software applications, if one persists its many benefits quickly become evident. Lexical scope gives Factor flexibility that can be lacking in other stack-based languages. Its point-free style allows for succinct code which highlights function composition. It is also worth noting that its syntax, while challenging at first, is easy for the computer to compile.

Through our investigations it has become evident that Factor is a language suited to not only creating a game, but to any software application. It possesses all the capabilities of any other functional language with a full complement of libraries to extend its already rich feature set.[14] As soon as we becomes acclimatized to Factor's more idiosyncratic qualities, we expect to receive a euphoric joy while programming with it. One might describe it as quirky but powerful. It is with confidence that the authors of this paper agree that Factor is suited to the creation of any substantial program. We are puzzled by the decision of Factor's designers to use whitespace as the ubiquitous delimiter.

References

[1] The big mud puddle: Why concatenative programming matters. `http://evincarofautumn.blogspot.ca/2012/02/why-concatenative-programming-matters.html`.

[2] Control Flow Cookbook - Factor Documentation. `http://docs.factorcode.org/content/article-cookbook-combinators.html`.

[3] Dataflow combinators - factor documentation. `http://docs.factorcode.org/content/article-dataflow-combinators.html`.

[4] Factor - Progopedia. `http://progopedia.com/language/factor`.

[5] Factor: a practical stack language: How factor implements closures. `http://factor-language.blogspot.ca/2007/03/adding-named-parameters-to-factor.html`.

[6] Factor: a practical stack language: Named local variables and lexical closures in factor. `http://factor-language.blogspot.ca/2007/08/named-local-variables-and-lexical.html`.

[7] Factor: a practical stack language: Prevalence of shuffle words and dataflow combinators.

http://factor-language.blogspot.ca/2008/12/prevalence-of-shuffle-words-and.html.

[8] Factor philosophy. http://docs.factorcode.org/content/article-cookbook-philosophy.html.

[9] Factor/faq. http://concatenative.org/wiki/view/Factor/FAQ.

[10] Forth inc.'s website. http://www.forth.com/.

[11] Game programming patterns / design patterns revisited. http://gameprogrammingpatterns.com.

[12] Gentleman, r., & ihaka, r. lexical scope and statistical computing. http://cran.r-project.org/doc/misc/lexical.tex.

[13] Joy mirror. http://www.kevinalbrecht.com/code/joy-mirror/joy.html.

[14] Libraries - Factor Documentation. http://docs.factorcode.org/content/article-handbook-library-reference.html.

[15] Tacit programming (wikipedia). http://en.wikipedia.org/wiki/Tacit_programming.

[16] Ui framework. http://docs.factorcode.org/content/article-ui.html.

[17] Unit testing - factor documentation. http://docs.factorcode.org/content/article-tools.test.html.

[18] What is lexical scoping? (fabulous adventures in coding). http://ericlippert.com/2013/05/20/what-is-lexical-scoping/.

[19] C Double. Factor articles - work in progress. http://bluishcoder.co.nz/factor-articles.pdf, 2013.

[20] Adobe Systems Incorporated. *PostScript language reference manual, 3rd ed.* Addison-Wesley Publishing Company, 1999.

[21] Charles Moore. Forth - the early years. http://www.colorforth.com/HOPL.html, 1999.

[22] S. Pestov, D. Ehrenberg, and J. Groff. Factor: a dynamic stack-based programming language. *DLS 2010 Proceedings of the 6th Symposium on Dynamic Languages*, 2010.

[23] Manfred von Thun. Joy faq. http://www.kevinalbrecht.com/code/joy-mirror/faq.html.

[24] Manfred von Thun. Recursion theory and joy. http://www.kevinalbrecht.com/code/joy-mirror/j05cmp.html.

[25] Manfred von Thun. Joy: Forth's functional cousin. http://www.kevinalbrecht.com/code/joy-mirror/forth-joy.html, 2001.