Factor

An Introduction to Concatenative Stack Languages

Alex Vondrak

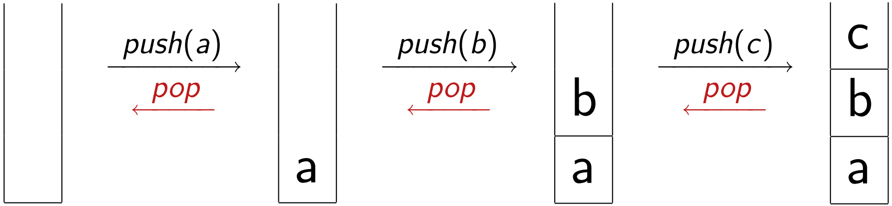
ajvondrak@csupomona.edu

October 14, 2009

From the Corner of Cool Languages™

- Assumption: you are not familiar with stack-based programming.
- Factor
  - Started development in 2003 – a baby among languages
  - Open source (BSD license)
  - Stack-based
  - Concatenative
- Priorities:
  1. Explain stack languages (bias towards Factor)
  2. What makes Factor cool?
  3. Learning all the stuff I have to skip

Review: Stacks



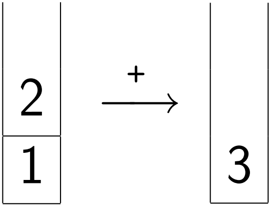
Stacks as an Evaluation Model

Example (Code)

```
1 2 +
```

Example (Execution)

```
push(1);
push(2);
y = pop();    // y = 2;
x = pop();    // x = 1;
push(x + y); // push(3);
```



Factor

A Practical Stack Language

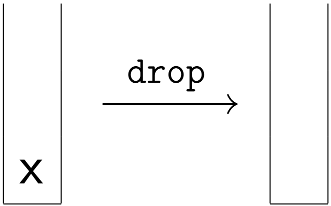
- There are several stack-based languages: Forth, PostScript, Joy, Cat, etc.
- Factor is...
  - ...high-level, typed, and garbage-collected (vs Forth)
  - ...dynamically typed (vs Cat)
  - ...more "practical" than "academic" (vs Joy)
- Instead of using variables, Factor programs manipulate global stacks.
  - Data Stack ("the" stack)
  - Retain Stack
  - Call Stack
  - Catch Stack
  - Name Stack

Stack Shufflers and Their Effects

Removing Stack Items

- drop

Stack Effect




```
drop ( x --  )
```

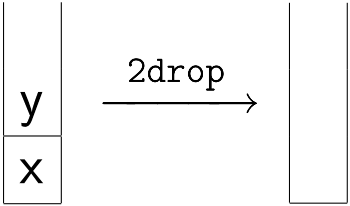
- 2drop
- nip
- Others

Stack Shufflers and Their Effects

Removing Stack Items

- `drop`
- `2drop`

Stack Effect



`2drop ( x y -- )`

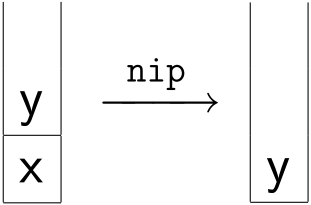
- `nip`
- Others

Stack Shufflers and Their Effects

Removing Stack Items

- drop
- 2drop
- nip

Stack Effect



`nip ( x y -- y )`

- Others

Stack Shufflers and Their Effects

Removing Stack Items

- `drop`
- `2drop`
- `nip`
- Others

Stack Effects

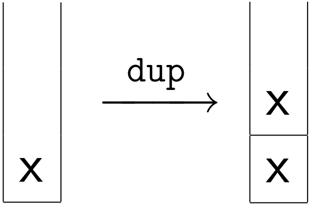
- `3drop ( x y z -- )`
- `2nip ( x y z -- z )`

Stack Shufflers and Their Effects

Duplicating Stack Items

- dup

Stack Effect




```
dup ( x -- x x )
```

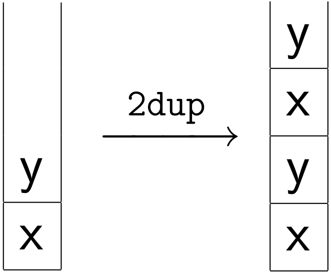
- 2dup
- Others

Stack Shufflers and Their Effects

Duplicating Stack Items

- dup
- 2dup

Stack Effect



`2dup ( x y -- x y x y )`

- Others

Stack Shufflers and Their Effects

Duplicating Stack Items

- `dup`
- `2dup`
- Others

Stack Effects

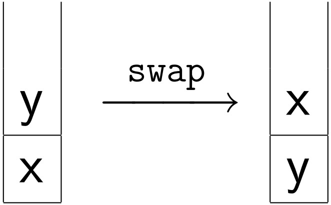
- `3dup ( x y z -- x y z x y z )`
- `dupd ( x y -- x x y )`
- `over ( x y -- x y x )`
- `2over ( x y z -- x y z x y )`
- `pick ( x y z -- x y z x )`
- `tuck ( x y -- y x y )`

Stack Shufflers and Their Effects

Permuting Stack Items

- `swap`

Stack Effect



`swap ( x y -- y x )`

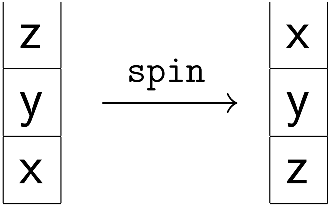
- `spin`
- Others

Stack Shufflers and Their Effects

Permuting Stack Items

- `swap`
- `spin`

Stack Effect




```
spin ( x y z -- z y x )
```

- Others

Stack Shufflers and Their Effects

Permuting Stack Items

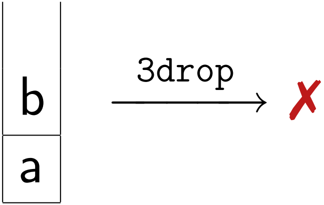
- `swap`
- `spin`
- Others

Stack Effects

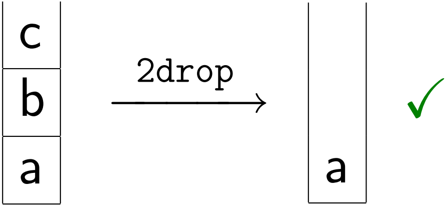
- `swapd ( x y z -- y x z )`
- `rot ( x y z -- y z x )`
- `-rot ( x y z -- z x y )`
- `roll ( x y z t -- y z t x )`
- `-roll ( x y z t -- t x y z )`

Not Enough Data? Too Much Data?

Underflow



No Underflow

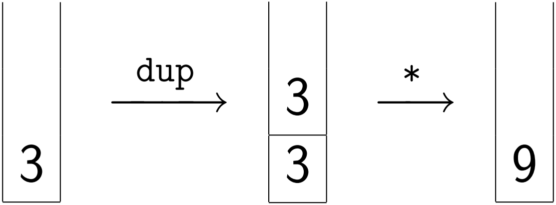


Composition

Intuitively

- By manipulating the stack, words can be executed one by one.

Example (Squaring A Number)



Composition

In Code

- To do several things to the stack, just write them out one by one.

Example ($x^2 + y^2$)

```
dup * swap dup * +
```

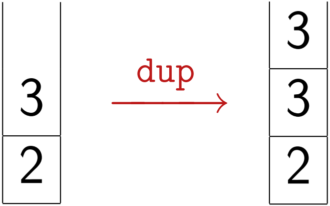
Composition

In Code

- To do several things to the stack, just write them out one by one.

Example ($x^2 + y^2$)

```
dup * swap dup * +
```



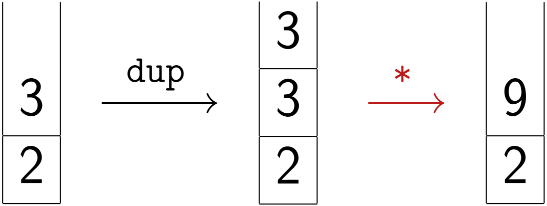
Composition

In Code

- To do several things to the stack, just write them out one by one.

Example $(x^2 + y^2)$

```
dup * swap dup * +
```



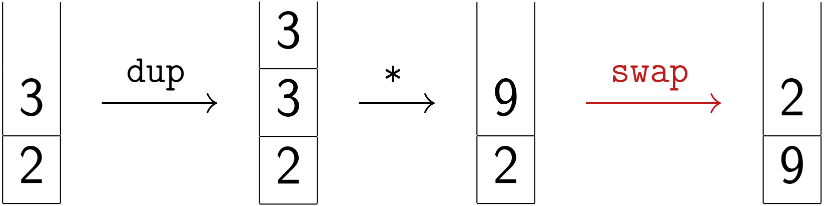
Composition

In Code

- To do several things to the stack, just write them out one by one.

Example $(x^2 + y^2)$

```
dup * swap dup * +
```

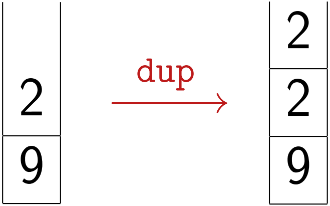


Composition
In Code

- To do several things to the stack, just write them out one by one.

Example $(x^2 + y^2)$

```
dup * swap dup * +
```



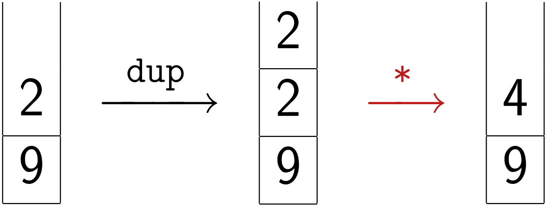
Composition

In Code

- To do several things to the stack, just write them out one by one.

Example $(x^2 + y^2)$

```
dup * swap dup * +
```



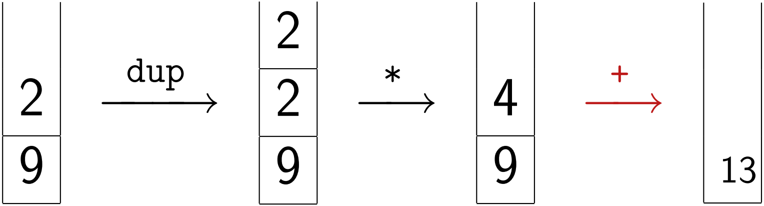
Composition

In Code

- To do several things to the stack, just write them out one by one.

Example $(x^2 + y^2)$

```
dup * swap dup * +
```



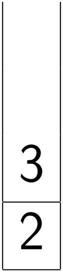
Concatenation

- Then, function composition is just word concatenation.

Example (Polar Coordinates)

$$r = \sqrt{x^2 + y^2} \quad \text{and} \quad \theta = \arctan\left(\frac{y}{x}\right)$$

```
2dup [dup * swap dup * +] sqrt spin / atan
```



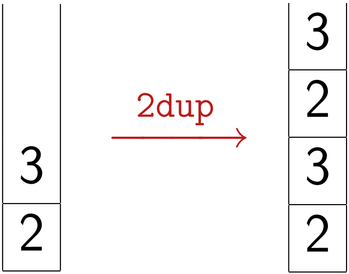
Concatenation

- Then, function composition is just word concatenation.

Example (Polar Coordinates)

$$r = \sqrt{x^2 + y^2} \quad \text{and} \quad \theta = \arctan\left(\frac{y}{x}\right)$$

```
2dup [dup * swap dup * +] sqrt spin / atan
```



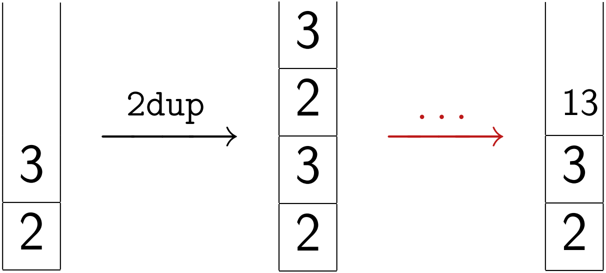
Concatenation

- Then, function composition is just word concatenation.

Example (Polar Coordinates)

$$r = \sqrt{x^2 + y^2} \quad \text{and} \quad \theta = \arctan\left(\frac{y}{x}\right)$$

```
2dup [dup * swap dup * +] sqrt spin / atan
```



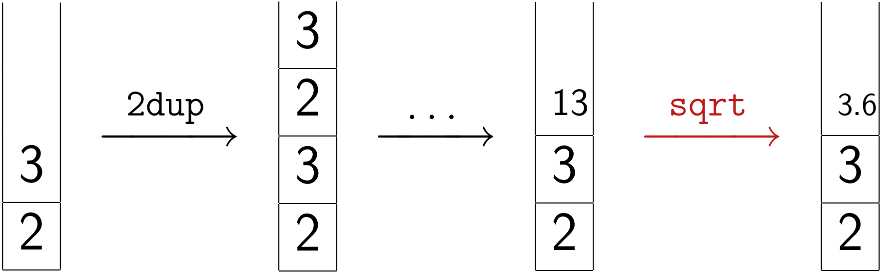
Concatenation

- Then, function composition is just word concatenation.

Example (Polar Coordinates)

$$r = \sqrt{x^2 + y^2} \quad \text{and} \quad \theta = \arctan\left(\frac{y}{x}\right)$$

```
2dup [dup * swap dup * +] sqrt spin / atan
```



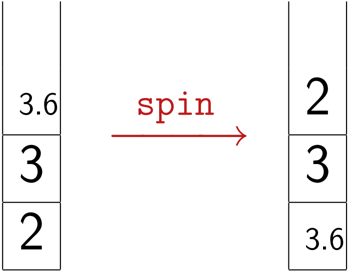
Concatenation

- Then, function composition is just word concatenation.

Example (Polar Coordinates)

$$r = \sqrt{x^2 + y^2} \quad \text{and} \quad \theta = \arctan\left(\frac{y}{x}\right)$$

```
2dup [dup * swap dup * +] sqrt spin / atan
```



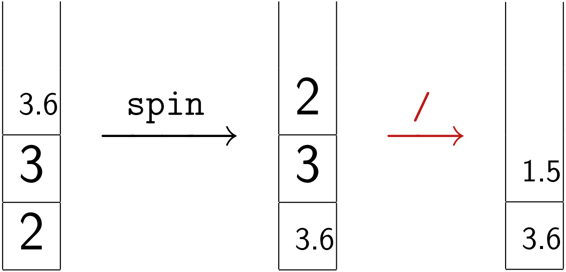
Concatenation

- Then, function composition is just word concatenation.

Example (Polar Coordinates)

$$r = \sqrt{x^2 + y^2} \quad \text{and} \quad \theta = \arctan\left(\frac{y}{x}\right)$$

```
2dup [dup * swap dup * +] sqrt spin / atan
```



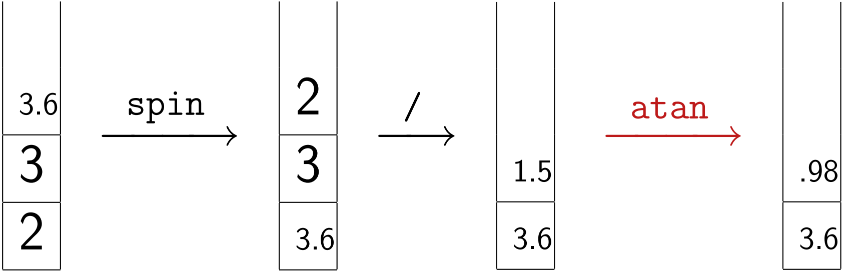
Concatenation

- Then, function composition is just word concatenation.

Example (Polar Coordinates)

$$r = \sqrt{x^2 + y^2} \quad \text{and} \quad \theta = \arctan\left(\frac{y}{x}\right)$$

`2dup` `dup * swap dup * +` `sqrt spin / atan`



Factoring

Before

```
2dup dup * swap dup * + sqrt spin / atan
```

After

```
: r ( x y -- magnitude ) dup * swap dup * + sqrt ;
: theta ( y x -- angle ) / atan ;

2dup r spin theta
```

- How else could we factor this?

Parsing

- Parsing is very simple in Factor: words are separated by whitespace.
- Data literals (numbers) are parsed and pushed onto the stack.
- Normal words execute code, but parsing words are a little special.

Example (How the Parser Sees It)

```
: theta ( y x -- angle ) / atan ;
```

- Tokenized as `:` `theta` `(` `y` `x` `--` `angle` `)` `/` `atan` `;`
- `:` is a parsing word that scans ahead for `;` and creates a word.
- `(` is a parsing word that scans ahead for `)` and gives a stack-effect.

Quotations

- Parsing words are defined in Factor.

Definition

```
USING: parser ;
IN: syntax
SYNTAX: [ parse-quotation parsed ;
```

Definition

```
IN: syntax
DEFER: ] ( -- * ) delimiter
```

- Code between the [and] is a quotation.
- The code in a quotation isn't executed until invoked.

Combinators

- Words that use quotations on the stack are called combinators.

Example (Control Flow)

```
2 3 < [ "true" print ] [ "false" print ] if ! prints "true"

[ t ] [ "hello" print "world" print ] while ! infinite loop
```

Example (Iteration)

```
{ "a" "b" "c" } [ print ] each
```

is the same as

```
"a" print "b" print "c" print
```

But It's Backwards!

Compare:

- Dot notation (Java, C++, et al.)

```
BigInteger.probablePrime(numBits/2, rnd);
```

- Unix pipes

```
$ find {basis,core,extra} -name *.factor |
> xargs wc -l |
> tail -1
263486 total
```

Example

```
USING: calendar calendar.format ;
11 days ago timestamp>ymd ! as of writing, "2009-09-11"
```

Can't I Just Use Variables?

- Variables can be a mental burden. Without them. . .
  - . . . what the program *does* becomes clearer.
  - . . . you worry less about bad variable names.
  - . . . the underlying structure is revealed – makes factoring easier.
- The stack allows for interesting abstractions.
  - Re-imagine old ones (e.g., continuations)
  - Multiple return values
  - Point-free style by default
- With enough use, of course it won't seem weird!

But Seriously, Can't I Just Use Variables?

Example (Lexical Variables)

```
USE: locals

:: discriminant ( a b c -- d )
    b sq
    4 a c * *
    - ;
```

Less than 1% of Factor's source uses locals:

```
$ find -name *.factor | xargs grep -l "^::" | wc -l
254
$ find -name *.factor | wc -l
3346
```

But It's Still Backwards!

Before

```
USE: locals

:: discriminant ( a b c -- d )
    b sq
    4 a c * *
    - ;
```

After

```
USING: locals infix ;

:: discriminant ( a b c -- d ) [infix b*b - 4*a*c infix] ;
```

1. Stack Languages
   - In the Abstract
   - In Code
   - Common Talking Points

2. Factor
   - Features, Libraries, Etc.

Implementation

- VM: about 15,000 lines of C++
- Core: about 10,000 lines of Factor (sans tests, docs)
- Basis: over 100,000 lines of Factor (sans tests, docs)
- Two machine-code compilers
  - Non-optimizing quotation compiler: quick, naive, part of the VM
  - Optimizing word compiler: slower, smarter, written in Factor
- Generational garbage collector
- Continuous integration build-farm (74,000 lines of tests in basis, core)
  - Architecture: x86, x86-64, PowerPC
  - OS: Windows, OS X, Linux, FreeBSD, NetBSD, OpenBSD

Interactive Development

```
( scratchpad ) 1

--- Data stack:
1
( scratchpad ) 2

--- Data stack:
1
2
( scratchpad ) +

--- Data stack:
3
```

Sequence Protocol

```
( scratchpad ) { "a" "b" "c" } [ . ] each
"a"
"b"
"c"
( scratchpad ) "abc" [ . ] each
97
98
99
( scratchpad ) 3 [ . ] each
0
1
2
```

Flexible Naming

Example (Ranges)

```
( scratchpad ) USE: math.ranges
( scratchpad ) 1 3 (a,b) [ . ] each
2
( scratchpad ) 1 3 (a,b] [ . ] each
2
3
( scratchpad ) 1 3 [a,b) [ . ] each
1
2
( scratchpad ) 1 3 [a,b] [ . ] each
1
2
3
```

Libraries

Sending an Email

```
USING: accessors smtp ;

<email>
    "css@csupomona.edu" >>from
    { "ajvondrak@csupomona.edu" } >>to
    "That was awful" >>subject
    "Get out." >>body
send-email
```

Libraries

Parser Expression Grammars

```
USING: peg.ebnf ;
...
EBNF: parse-url

protocol = [a-z]+                    => [[ url-decode ]]
username = [^/:@#?]+                 => [[ url-decode ]]
password = [^/:@#?]+                 => [[ url-decode ]]
pathname = [^#?]+                    => [[ url-decode ]]
query    = [^#]+                     => [[ query>assoc ]]
anchor   = .+                        => [[ url-decode ]]
...
;EBNF
```

Libraries

More

- GUI tools
- Macros
- Farkup (custom HTML markup language)
- Furnace (web framework)
- C Foreign Function Interface
- Regular expressions
- UI and command-line “listeners”
- Text editor integration (Vim, Emacs, TextMate)
- Deploy tool
- Various data structures
- . . .

Summary

- Concatenative programming lets you compose programs by joining them together with whitespace.
- Stack languages facilitate concatenative programming by passing data around on the stack(s).
- Factor is a particularly good stack programming language:
  - High level
  - Practical – has a lot of libraries
  - Cross platform
  - Focuses on performance, which is always getting better
  - And of course. . .

Did You *See* That Fucking Raptor?!

Figure: Velociraptor Mongoliensis

Who's going to mess with you if your mascot is a dinosaur? Nobody, that's who!

More

For the stuff I missed, check out:

- Factor's website: http://factorcode.org/
  - Searchable documentation (http://docs.factorcode.org/)
  - Wiki
  - Downloads
  - etc.
- Creator Slava Pestov's Google Tech Talk (on YouTube)
  - First Google result for *Factor tech talk*
  - A little old, but explains Factor's compiler and object system
  - Much more about Factor itself
- Development blog: http://factor-language.blogspot.com/
  - Slava Pestov discusses new features
  - Other blogs aggregated at http://planet.factorcode.org/