Squeak, a Smalltalk system

Smalltalk: the first **pure** OO language

- all data structures and values are **objects**
- all operations are **methods** invoked by **message passing**
- uniform reference data model, with garbage collection
- strongly, **dynamically** typed

Includes first-class function objects (**blocks**)

Includes rich standard data structure & graphics libraries

Includes interactive graphical programming environment

“Interesting” syntax...

Squeak: a current, actively growing Smalltalk system

Smalltalk syntax

An expression is one of:

- a literal
  - an integer: `17`
  - a float: `3.5`
  - a string: `’a string’`
  - a character: `$a`
  - a symbol: `#abc`
  - an array: `#(17 $a ’hi there’ () abc)`
- a variable
  - an instance variable: `xyz`
  - a class or global variable: `Xyz`
  - a pseudo-variable: `true`, `false`, `nil`, `self`, `super`
- a variable assignment
  - `xyz := expr`
  - can type _ (which prints as ←) instead of `:=`
- a message send...
- a block...

Comments in double quotes: `"this is a comment"`

Message syntax

Smalltalk uses three message syntaxes

- a postfix **unary** message: `17 negated`
- an infix **binary** message: `17 + 18`
- a **keyword** message: `17 foo: 18 bar: 19`
  (effect is like `(foo:bar:)(17, 18, 19)`)

Parsing rules:

If one or two punctuation symbols (+, <=, &&),
interpret as a binary message

- receiver to the left, argument to the right of the msg name

Else if word does not end in a colon,
interpret as a variable reference (if no receiver)
or a unary message to the receiver expression on its left

Otherwise, interpret as (part of) a keyword message

- receiver of keyword before first keyword part
- one additional argument to message after each keyword
- keep adding keywords together until end of statement
  to form one big multi-argument message

Precedence

Unaries have highest precedence, then binaries, then keywords

Example:

```
17 foo + 18 bar frob: 19 + ’asd’ zappo flim: 6.3
```

Associativity

Unaries are left associative (they have to be):

```
17 foo baz bar + bop quib droob
```

Binaries are left associative (always, possibly violating math):

```
3 + 4 * 5 / 6 ** 7 ** 8
```

Keywords don’t matter; only one per statement if no parens:

```
18 foo: 19 bar: (20 frob: 21) biz: 22
```

Methods

Example:

```
frob: foo diz: bar
  | bloop blop |
  bloop := foo dwizzle.
  blop := bar * self blip: dwaddle.
  ^ bloop + blop
```

Blocks

Blocks are like `fn` in ML:
anonymous, lexically-scoped function objects

All control structures take blocks as arguments

Users can define their own control structures
which take blocks as arguments

Examples:

```
[ ’hi there’ ]
[ :item1 :item2 | item1 print. item2 print. ]
[ self initialize. ^ ’done’ ]
```

Control structures in Smalltalk

Conditionals

```
test ifTrue: [ true part ]
    ifFalse: [ false part ]
```

While loops

```
[ test ] whileTrue: [ body ]
[ test ] whileFalse: [ body ]
```

For loops

```
number timesRepeat: [ body ]
start to: end do: [ :i | body ]
start to: end by: step do: [ :i | body ]
```

General iteration

```
collection do: [ :elem | body ]
collection collect: [ :elem | expr ]
collection select: [ :elem | test ]
collection inject: init
             into: [ :val :elem | expr ]
```

Block semantics

Evaluating a block literal returns a new block object

Blocks are lexically-scoped:

- variable references search the enclosing method to find a binding
- `self` is bound to the receiver of the lexically-enclosing method (not the block as you might expect)

Unlike methods, blocks without ^ return the result of their last expression

Non-local returns

If a block's last statement is prefixed with a `^`, the block does a *non-local return*

The block does *not* return to its caller

Instead, it returns to the caller of the lexically-enclosing method

Example:

```
safeSqrt: x
  x <= 0 ifTrue: [ ^ 0 ].
  ^ x sqrt
```

^ acts like a `return` statement in other languages

Invoking a block

If a block takes no arguments, invoke it by sending `value` :

```
[ 'hi there' print ] value
```

If a block takes one argument, invoke it by sending `value:` :

```
[ :msg | msg print ] value: 'hi there'
```

If a block takes two arguments, invoke it by sending `value:with:` :

```
[ :msg1 :msg2 | msg1 print. msg2 print ]
   value: 'hi' with: ' there'
```

If a block takes *N* arguments, invoke it by sending `value:{with:}`$^{N-1}$:

```
[ :msg1 :msg2 :msg3 :msg4 |
    msg1 print. msg2 print.
    msg3 print. msg4 print. ]
  value: 'hi' with: ' ' with: 'the' with: 're'
```