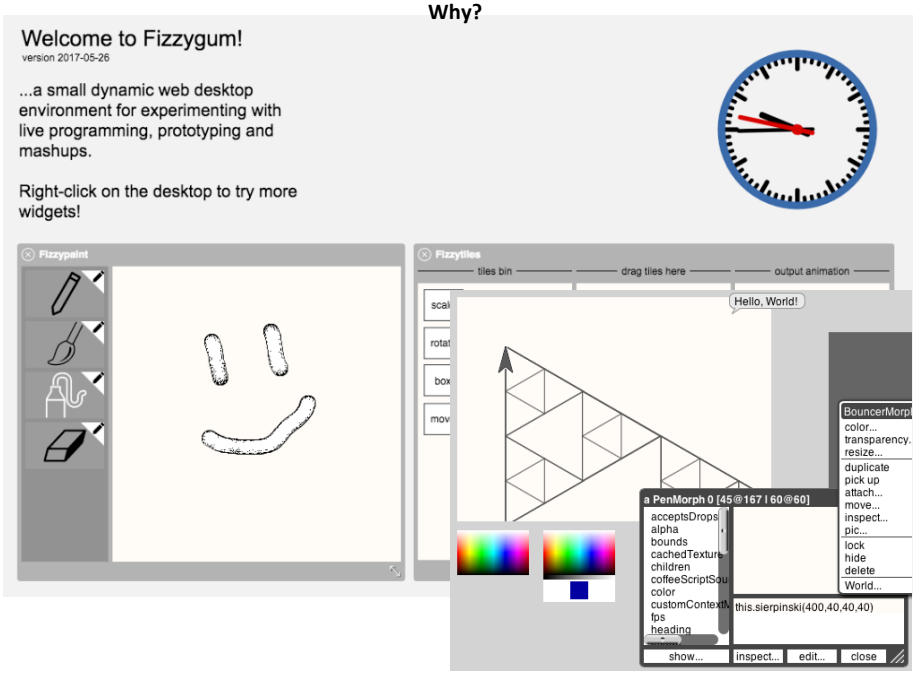
- Why?
- ST-72 examples
- ST-72 sources
- Desperation stage
- Hope again: Rosetta Smalltalk
- Rosetta Smalltalk examples
- Rosetta Smalltalk sources
- Fizzylogo: Rosetta Smalltalk (with some twists)
- Examples of Fizzylogo

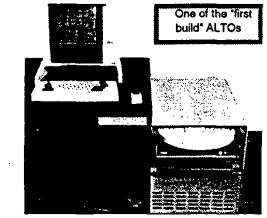
ST 72
- flexible grammar: parsed dynamically
- takes eval/apply loop of LISP, roots it in OO, bends grammar to look like LOGO
- non-recursive interpreter can be used.

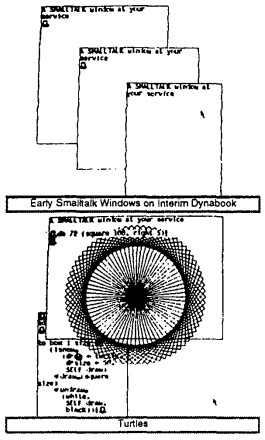
> *"To me, the most interesting thing about this exercise was how much expressive power could be obtained from a tiny amount of code"* - AK

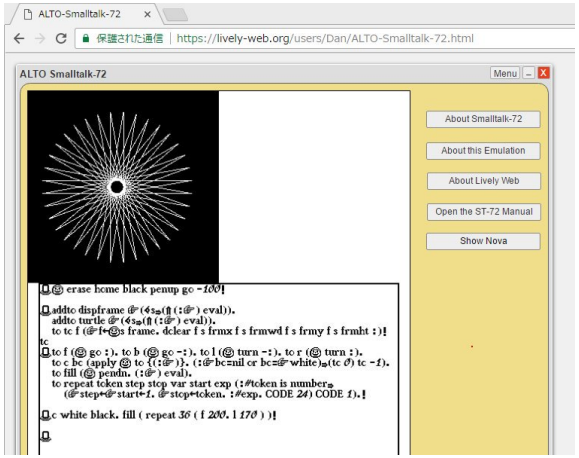
ST 76
- fixed grammar
- inheritance.

Why?







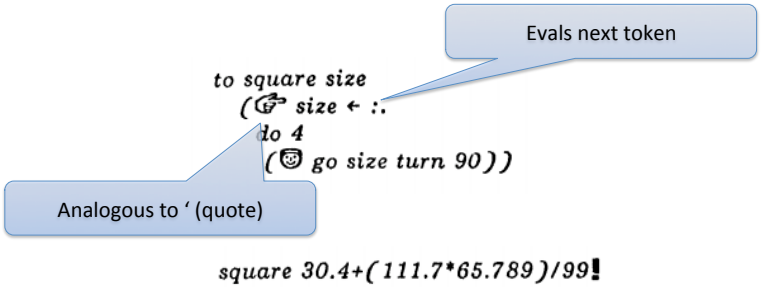


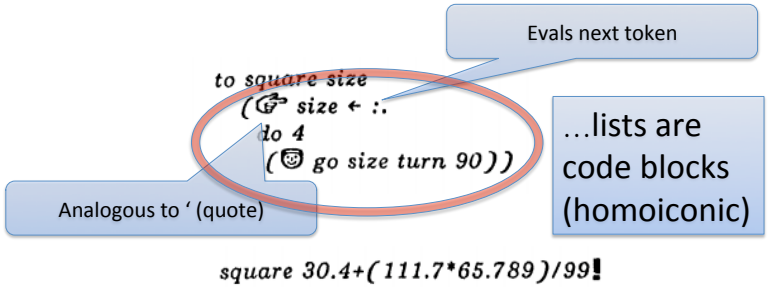
Examples of sending messages:

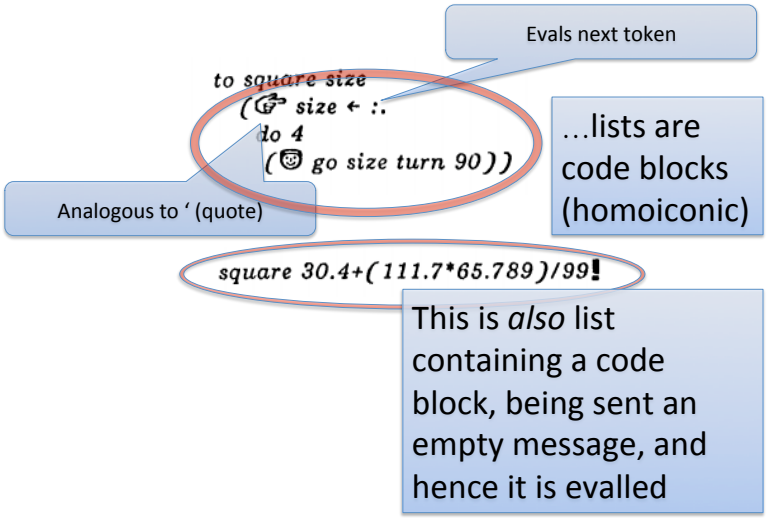
	Communication	Object	Message	Reply	Graphics Action
1.	3+4+5	3	+4+5	12	none
2.	5 mod 3	5	mod 3	2	none
3.	'abc'+'def'	'abc'	+'def'	'abcdef'	none
4.	🐵 go 100	🐵	go 100	🐵	draws a line 100 units long
5.	do 4 (🐵 go 50 turn 90.)	do	4 (🐵 go 50 turn 90.)	none	draws a square with side 50 units long
6.	joe grow 50	joe	grow 50	none	joe, the box, grows his sides by 50 units
7.	joe turn 25. jill grow 30.	joe jill	turn 25 grow 30	none none	joe turns 25 degrees jill grows her sides 30 units

```
to square size
  (👉 size ← :.
   do 4
    (🐵 go size turn 90))

square 30.4+(111.7*65.789)/99!
```








```
to square size
  (☞ size ← :.
   do 4
    (🦁 go size turn 90))

square 30.4+(111.7*65.789)/99!
```

“square” looks like a LOGO function, but actually creates a class.

(one could instantiate it)


```
to square size
  (☞ size ← :.
   do 4
    (🦁 go size turn 90))

square 30.4+(111.7*65.789)/99!
```

“square” looks like a LOGO function, but actually creates a class.

(one could instantiate it)

…shorthand for:

☞ square ← class <temp vars> …
… messages and responses


```
to square size
  (☞ size ← :.
   do 4
    (🦁 go size turn 90))
```

This class only answers to one message.

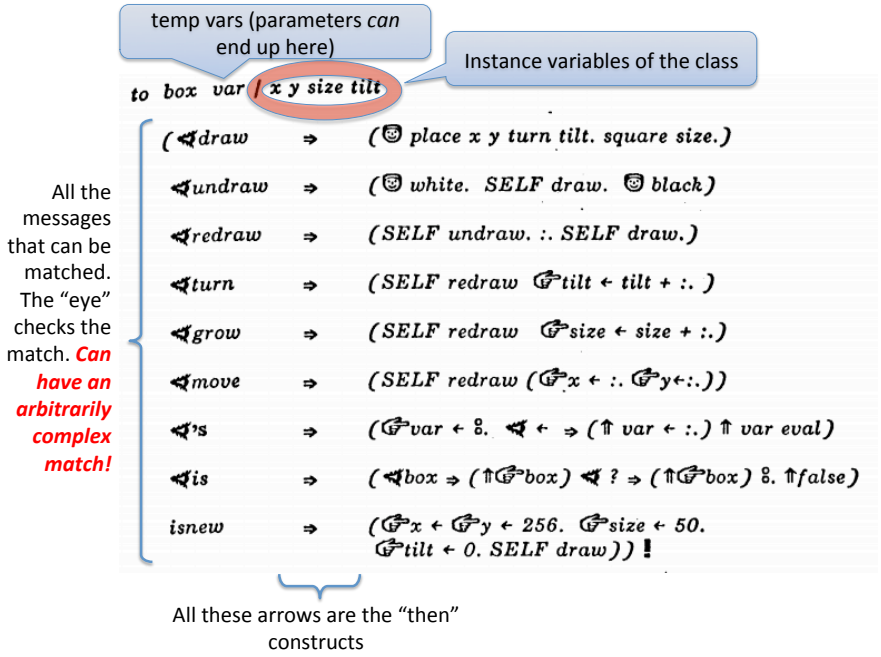
```
square 30.4+(111.7*65.789)/99!
```

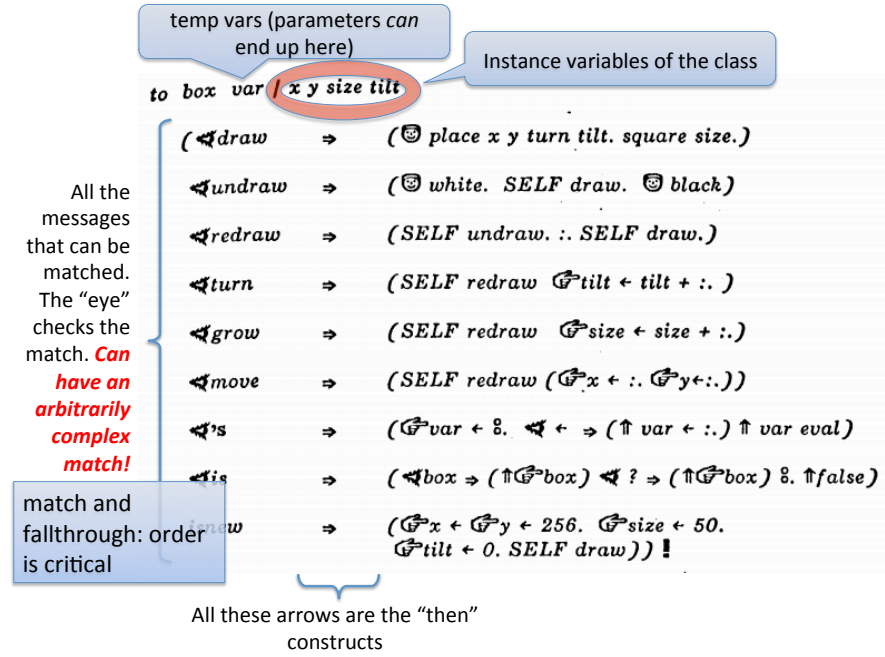
“square” looks like a LOGO function, but actually creates a class.

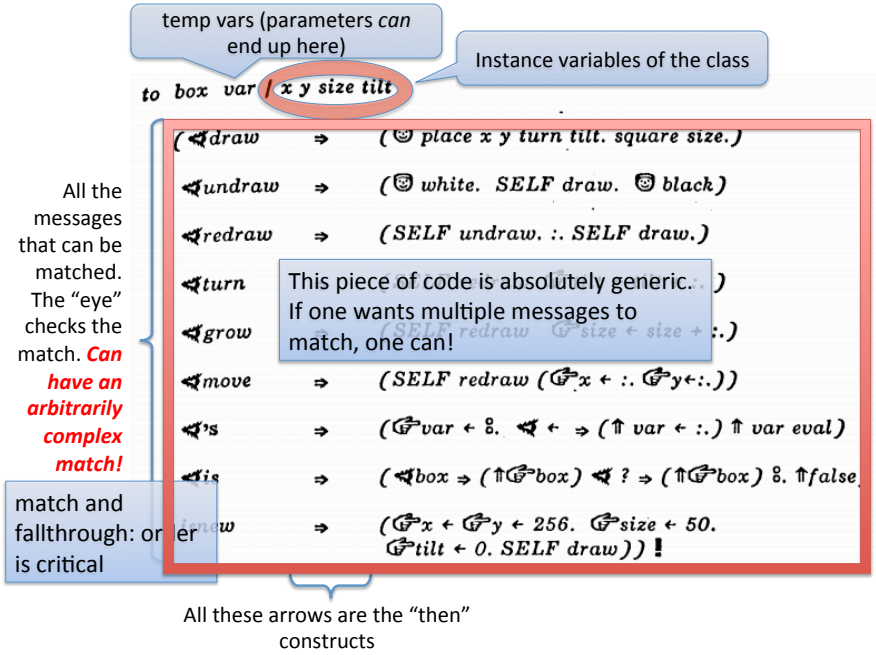
(one could instantiate it)

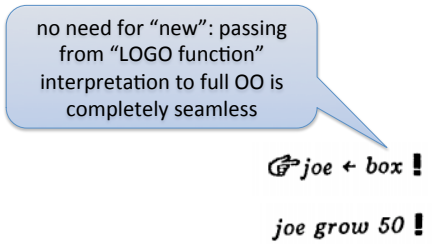
…shorthand for:

☞ square ← class <temp vars> …
… messages and responses

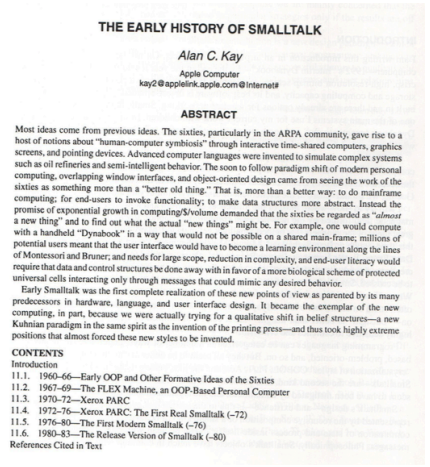

temp vars (parameters *can* end up here)
Instance variables of the class
to box var | x y size tilt
All the messages that can be matched. The "eye" checks the match. *Can have an arbitrarily complex match!*
(◁draw ⇒ (☺ place x y turn tilt. square size.)
◁undraw ⇒ (☺ white. SELF draw. ☹ black)
◁redraw ⇒ (SELF undraw. :. SELF draw.)
◁turn ⇒ (SELF redraw ☞tilt ← tilt + :.)
◁grow ⇒ (SELF redraw ☞size ← size + :.)
◁move ⇒ (SELF redraw (☞x ← :. ☞y←:.))
◁'s ⇒ (☞var ← ☟. ◁ ← ⇒ (⇑ var ← :.) ⇑ var eval)
◁is ⇒ (◁box ⇒ (⇑☞box) ◁ ? ⇒ (⇑☞box) ☟. ⇑false)
isnew ⇒ (☞x ← ☞y ← 256. ☞size ← 50. ☞tilt ← 0. SELF draw)) !
All these arrows are the "then" constructs


temp vars (parameters *can* end up here)
Instance variables of the class
to box var | x y size tilt
All the messages that can be matched. The "eye" checks the match. *Can have an arbitrarily complex match!*
match and fallthrough: order is critical
(◁draw ⇒ (☺ place x y turn tilt. square size.)
◁undraw ⇒ (☺ white. SELF draw. ☹ black)
◁redraw ⇒ (SELF undraw. :. SELF draw.)
◁turn ⇒ (SELF redraw ☞tilt ← tilt + :.)
◁grow ⇒ (SELF redraw ☞size ← size + :.)
◁move ⇒ (SELF redraw (☞x ← :. ☞y←:.))
◁'s ⇒ (☞var ← ☟. ◁ ← ⇒ (⇑ var ← :.) ⇑ var eval)
◁is ⇒ (◁box ⇒ (⇑☞box) ◁ ? ⇒ (⇑☞box) ☟. ⇑false)
isnew ⇒ (☞x ← ☞y ← 256. ☞size ← 50. ☞tilt ← 0. SELF draw)) !
All these arrows are the "then" constructs


temp vars (parameters *can* end up here)
Instance variables of the class
to box var | x y size tilt
All the messages that can be matched. The "eye" checks the match. *Can have an arbitrarily complex match!*
match and fallthrough: order is critical
This piece of code is absolutely generic. If one wants multiple messages to match, one can!
(◁draw ⇒ (☺ place x y turn tilt. square size.)
◁undraw ⇒ (☺ white. SELF draw. ☹ black)
◁redraw ⇒ (SELF undraw. :. SELF draw.)
◁turn
◁grow
◁move ⇒ (SELF redraw (☞x ← :. ☞y←:.))
◁'s ⇒ (☞var ← ☟. ◁ ← ⇒ (⇑ var ← :.) ⇑ var eval)
◁is ⇒ (◁box ⇒ (⇑☞box) ◁ ? ⇒ (⇑☞box) ☟. ⇑false
isnew ⇒ (☞x ← ☞y ← 256. ☞size ← 50. ☞tilt ← 0. SELF draw)) !
All these arrows are the "then" constructs


no need for "new": passing from "LOGO function" interpretation to full OO is completely seamless
☞joe ← box !
joe grow 50 !

(not exactly as simple as Logo though)



THE EARLY HISTORY OF SMALLTALK

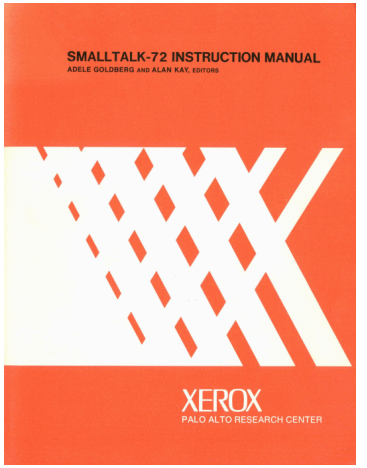
Alan C. Kay
Apple Computer
kay2@applelink.apple.com@Internet#

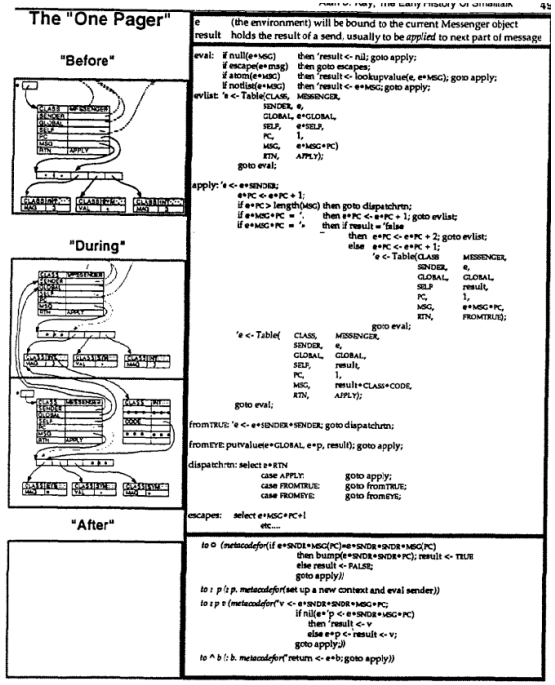
ABSTRACT

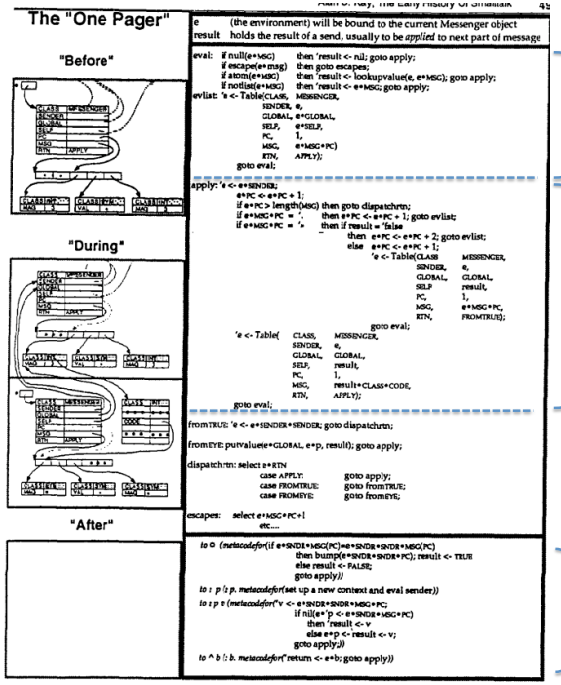
Most ideas come from previous ideas. The sixties, particularly in the ARPA community, gave rise to a host of notions about "human-computer symbiosis" through interactive time-shared computers, graphics screens, and pointing devices. Advanced computer languages were invented to simulate complex systems such as oil refineries and semi-intelligent behavior. The soon to follow paradigm shift of modern personal computing, overlapping window interfaces, and object-oriented design came from seeing the work of the sixties as something more than a "better old thing." That is, more than a better way: to do mainframe computing; for end-users to invoke functionality; to make data structures more abstract. Instead the promise of exponential growth in computing/$/volume demanded that the sixties be regarded as "*almost* a new thing" and to find out what the actual "new things" might be. For example, one would compute with a handheld "Dynabook" in a way that would not be possible on a shared main-frame; millions of potential users meant that the user interface would have to become a learning environment along the lines of Montessori and Bruner; and needs for large scope, reduction in complexity, and end-user literacy would require that data and control structures be done away with in favor of a more biological scheme of protected universal cells interacting only through messages that could mimic any desired behavior.

Early Smalltalk was the first complete realization of these new points of view as parented by its many predecessors in hardware, language, and user interface design. It became the exemplar of the new computing, in part, because we were actually trying for a qualitative shift in belief structures—a new Kuhnian paradigm in the same spirit as the invention of the printing press—and thus took highly extreme positions that almost forced these new styles to be invented.

CONTENTS



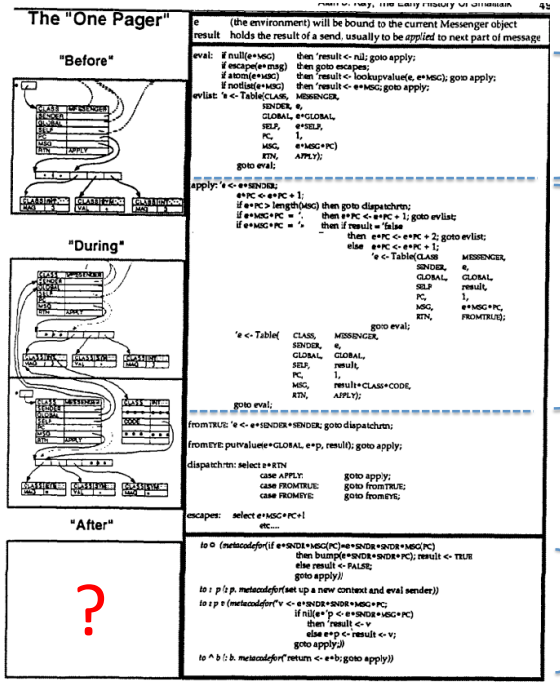


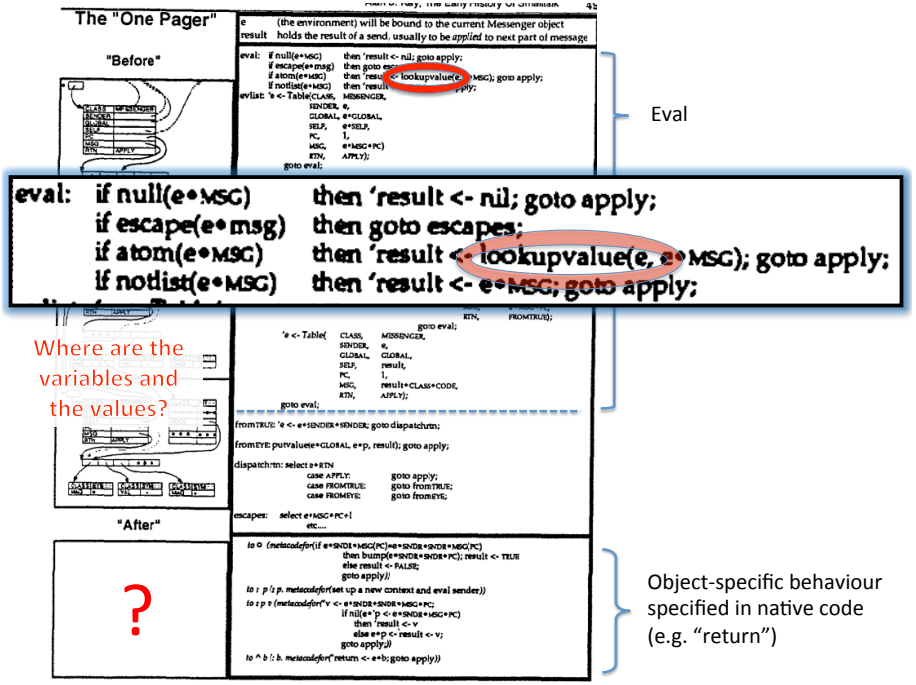


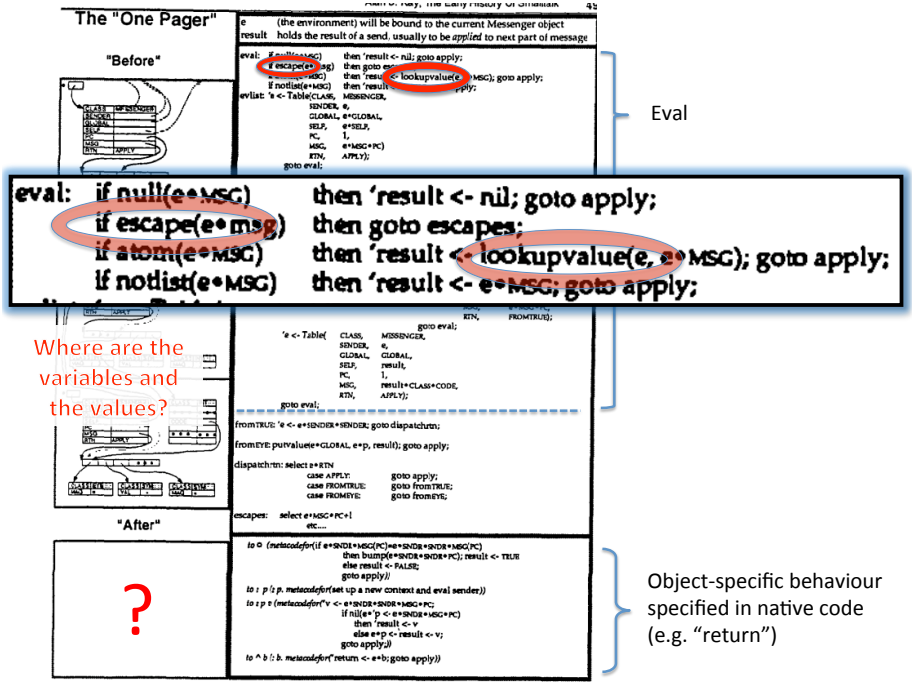
Eval

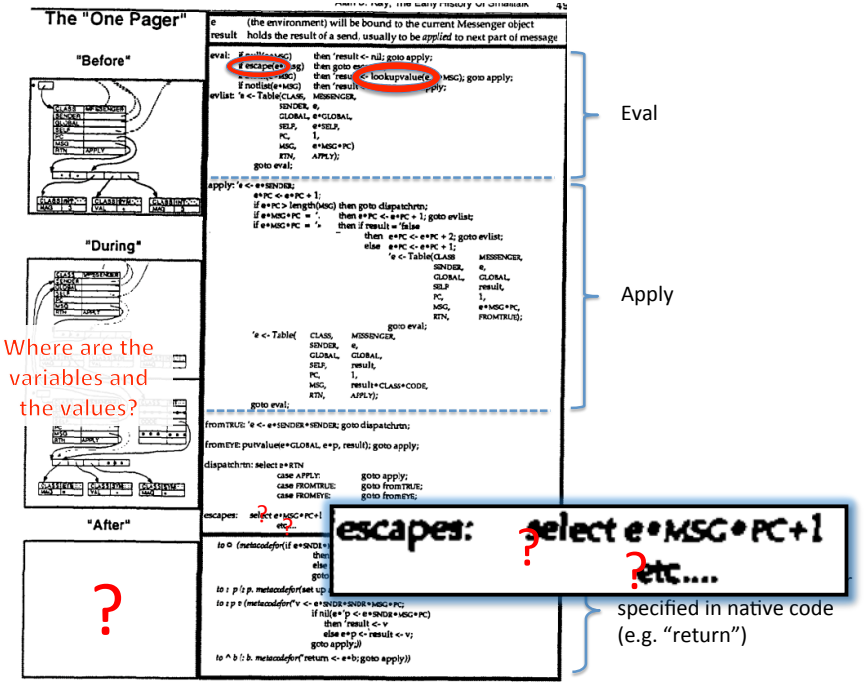
Apply

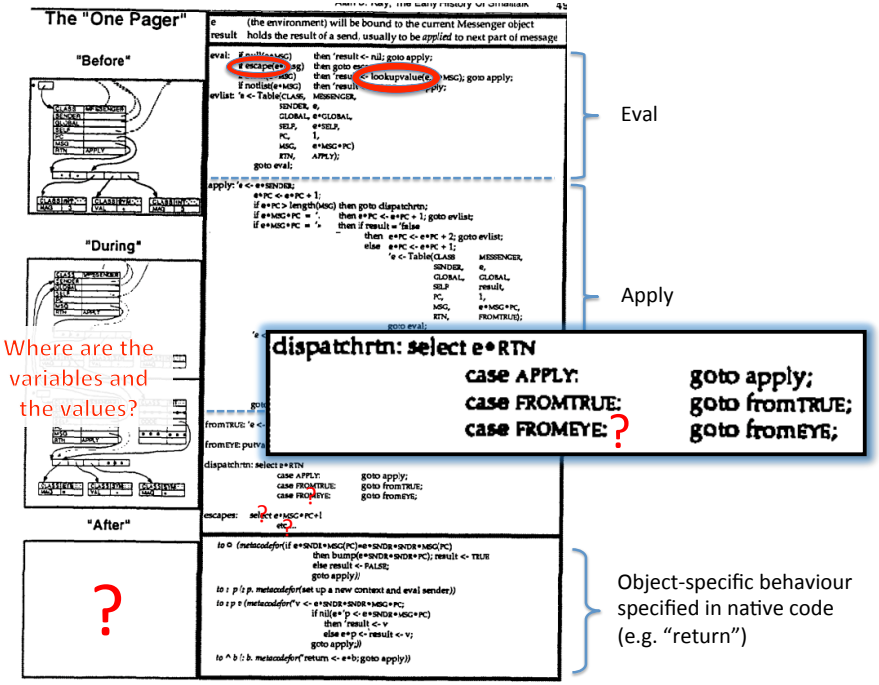
Object-specific behaviour specified in native code (e.g. “return”)

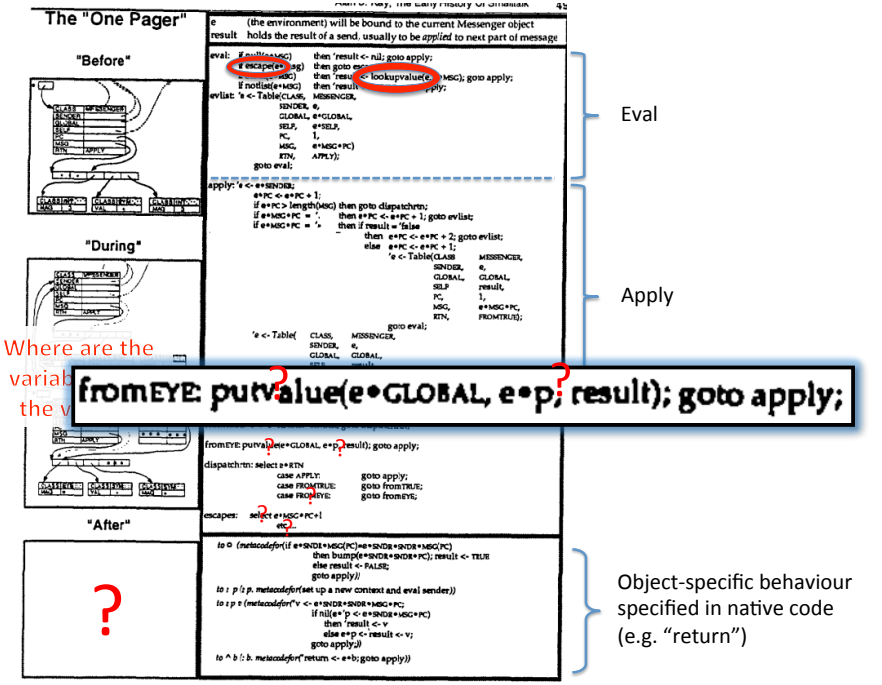

The "One Pager"
"Before"
"During"
"After"
?
Eval
Apply
Object-specific behaviour specified in native code (e.g. "return")

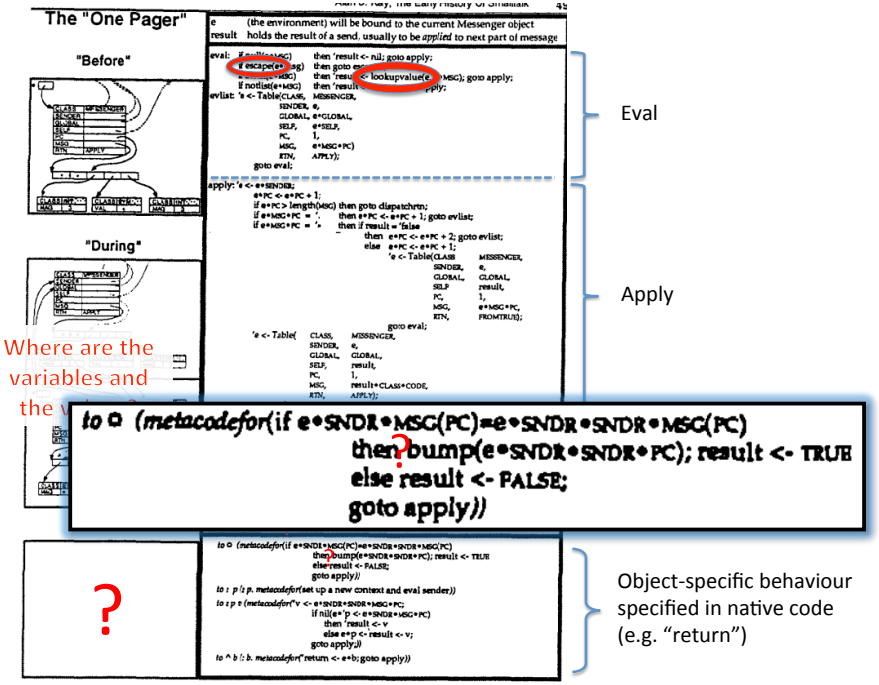

The "One Pager"
"Before"
"After"
?
eval: if null(e•MSG) then 'result <- nil; goto apply;
if escape(e•msg) then goto escapes;
if atom(e•MSG) then 'result <- lookupvalue(e, e•MSG); goto apply;
if notlist(e•MSG) then 'result <- e•MSG; goto apply;
Where are the variables and the values?
Eval
Object-specific behaviour specified in native code (e.g. "return")

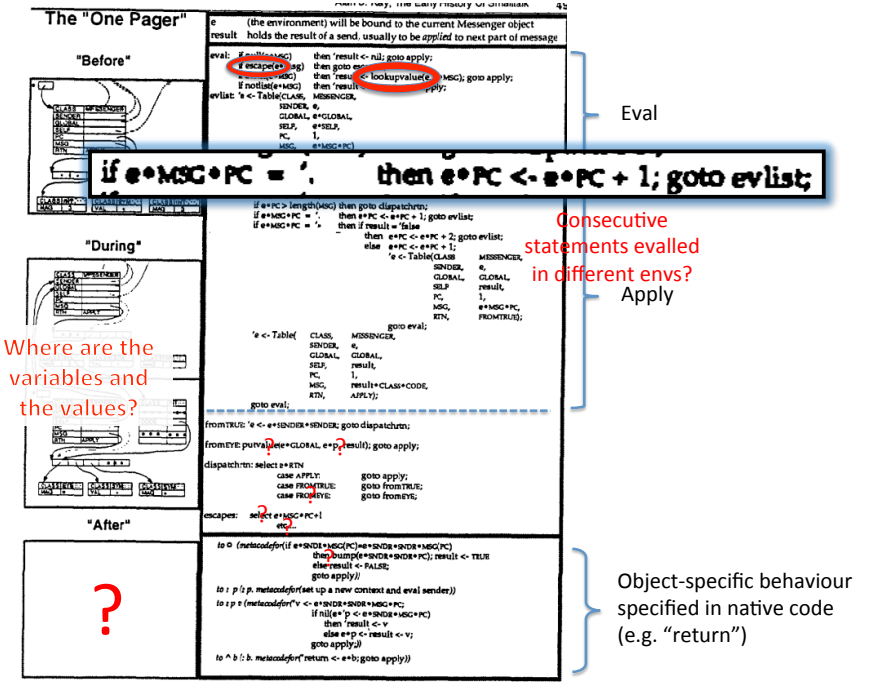

The "One Pager"
"Before"
"During"
"After"
?
eval: if null(e•MSG) then 'result <- nil; goto apply;
if escape(e•msg) then goto escapes;
if atom(e•MSG) then 'result <- lookupvalue(e, e•MSG); goto apply;
if notlist(e•MSG) then 'result <- e•MSG; goto apply;
Where are the variables and the values?
Eval
Apply
Object-specific behaviour specified in native code (e.g. "return")

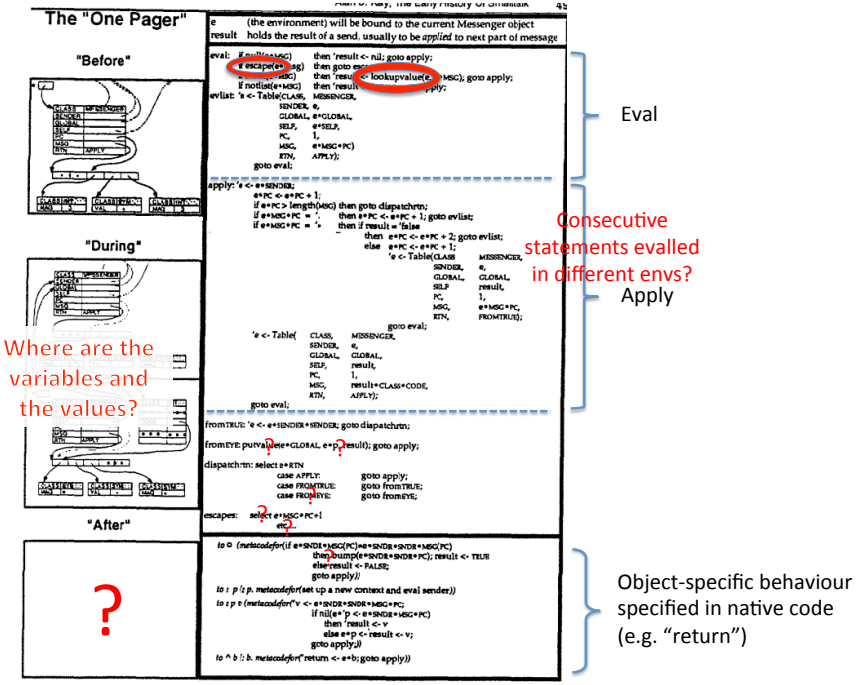

The "One Pager"
"Before"
"During"
"After"
?
escapes: select e•MSG•PC+1
etc....
Where are the variables and the values?
Eval
Apply
specified in native code (e.g. "return")

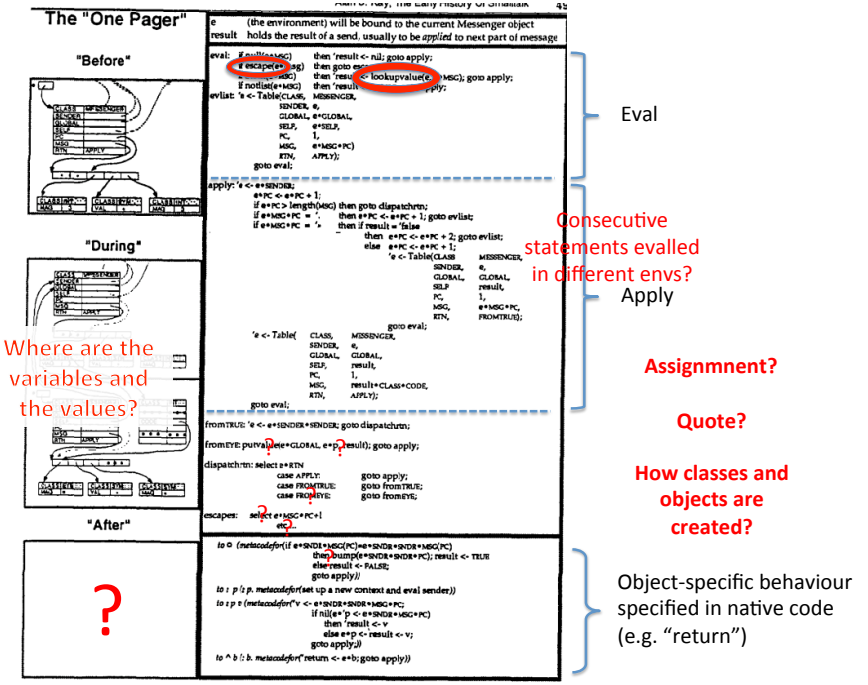

The "One Pager"
"Before"
"During"
"After"
Where are the variables and the values?
Eval
Apply
Object-specific behaviour specified in native code (e.g. "return")
dispatchrtn: select e•RTN
case APPLY: goto apply;
case FROMTRUE: goto fromTRUE;
case FROMEYE: goto fromEYE;

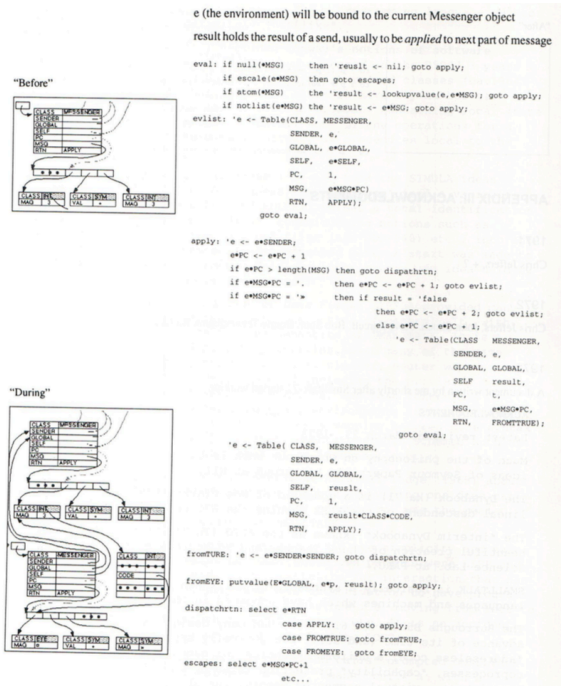

The "One Pager"
"Before"
"During"
"After"
Where are the variables and the values?
Eval
Apply
Object-specific behaviour specified in native code (e.g. "return")
fromEYE: putvalue(e•GLOBAL, e•p, result); goto apply;

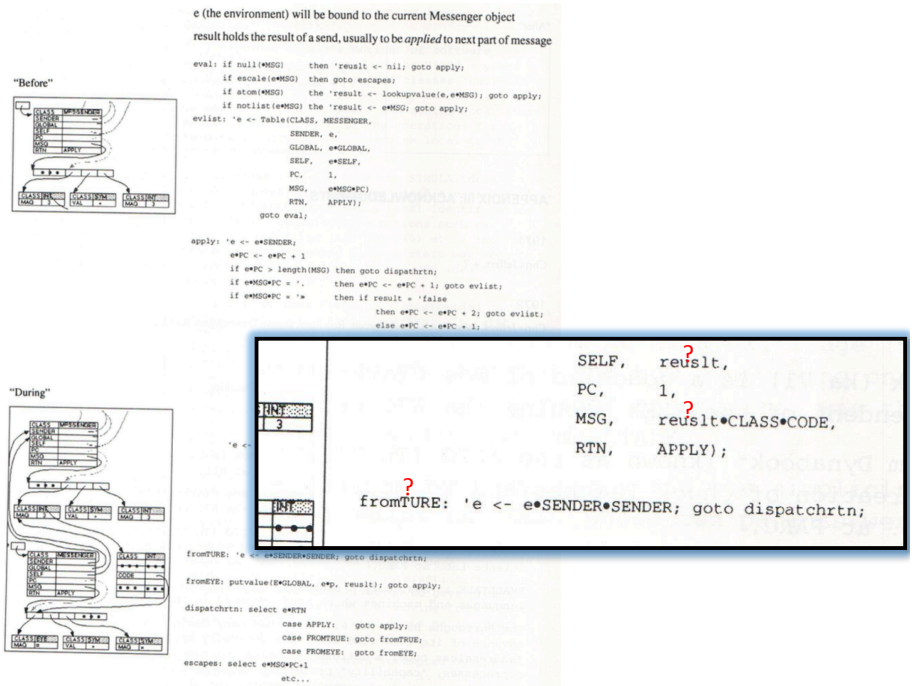

The "One Pager"
"Before"
"During"
"After"
Where are the variables and the values?
Eval
Apply
Object-specific behaviour specified in native code (e.g. "return")
to ◘ (metacodefor(if e•SNDR•MSG(PC)=e•SNDR•SNDR•MSG(PC)
then bump(e•SNDR•SNDR•PC); result <- TRUE
else result <- FALSE;
goto apply))


The "One Pager"
"Before"
"During"
"After"
Where are the variables and the values?
Eval
Apply
Consecutive statements evalled in different envs?
if e•MSG•PC = ', then e•PC <- e•PC + 1; goto evlist;
Object-specific behaviour specified in native code (e.g. "return")


The "One Pager"
"Before"
"During"
"After"
Where are the variables and the values?
?
e (the environment) will be bound to the current Messenger object
result holds the result of a send, usually to be applied to next part of message
Eval
Consecutive statements evalled in different envs?
Apply
Object-specific behaviour specified in native code (e.g. "return")


The "One Pager"
"Before"
"During"
"After"
Where are the variables and the values?
?
e (the environment) will be bound to the current Messenger object
result holds the result of a send, usually to be applied to next part of message
Eval
Consecutive statements evalled in different envs?
Apply
Assignmnent?
Quote?
How classes and objects are created?
Object-specific behaviour specified in native code (e.g. "return")


"Before"
"During"
e (the environment) will be bound to the current Messenger object
result holds the result of a send, usually to be applied to next part of message


"Before"
"During"
e (the environment) will be bound to the current Messenger object
result holds the result of a send, usually to be applied to next part of message
SELF, reuslt,
PC, 1,
MSG, reuslt•CLASS•CODE,
RTN, APPLY);
fromTURE: 'e <- e•SENDER•SENDER; goto dispatchrtn;

Smalltalk 72

Summary: This is Smalltalk 72 for Squeak
Author: Dan Ingalls
Owner: Marcus Denker (md)
Co-maintainers: *<None>*
Categories:

- Stable - Useable by all. Bugs are rare.
- SqueakL - The original license of Squeak from Apple. Obsolete since Squeak 4.0 (which only accepts MIT contributions). SqueakL is not recommended to be used for new projects.
- Squeak3.4 -
- Squeak - The official distribution of Squeak.

Homepage: http://wiki.squeak.org/squeak/989
PackageInfo name: *<Not entered>*
RSS feed:

Description:

Dan Ingalls did an emulation of the Smalltalk-72 emulator for Alan Kay's birthday.

The original Smalltalk 72 User Manual is available:
http://www.spies.com/~aek/pdf/xerox/alto/Smalltalk72_Manual.pdf

Releases

- 1-1 -

1600 LOC (*probably non recursive?*)

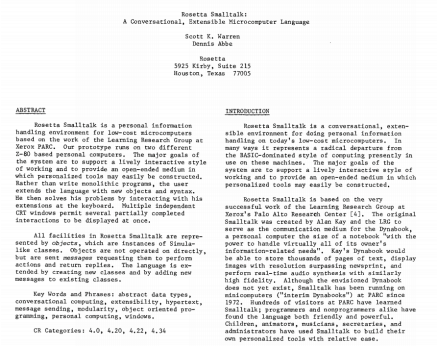
However:

1) would copy exactly an existing implementation with warts and all

"Repeats an acknowleged crock in the original ST72 intepreter: Test if an inlined vector should be evalled if it is the same as the last message token and that is not preceded by a quote(!!)"

2) could become an exercise in matching the ST runtime to JS runtime:

to: title temps: tempNames ivars: ivars cvars: cvars code: codeVector "Create a new class, or identify an existing one or Squeak equivalent"



Rosetta Smalltalk:
A Conversational, Extensible Microcomputer Language

Scott K. Warren
Dennis Abbe

Rosetta
5923 Kirby, Suite 215
Houston, Texas 77005

ABSTRACT

Rosetta Smalltalk is a personal information handling environment for low-cost microcomputers based on the work of the Learning Research Group at Xerox PARC. Our prototype runs on two different Z-80 based personal computers. The major goals of the system are to support a lively interactive style of working and to provide an open-ended medium in which personalized tools may easily be constructed. Rather than write monolithic programs, the user extends the language with new objects and syntax. He then solves his problems by interacting with his extensions at the keyboard. Multiple independent CRT windows permit several partially completed interactions to be displayed at once.

All facilities in Rosetta Smalltalk are represented by objects, which are instances of Simula-like classes. Objects are not operated on directly, but are sent messages requesting them to perform actions and return replies. The language is extended by creating new classes and by adding new messages to existing classes.

Key Words and Phrases: abstract data types, conversational computing, extensibility, hypertext, message sending, modularity, object oriented programming, personal computing, windows.

CR Categories: 4.0, 4.20, 4.22, 4.34

INTRODUCTION

Rosetta Smalltalk is a conversational, extensible environment for doing personal information handling on today's low-cost microcomputers. In many ways it represents a radical departure from the BASIC-dominated style of computing presently in use on these machines. The major goals of the system are to support a lively interactive style of working and to provide an open-ended medium in which personalized tools may easily be constructed.

Rosetta Smalltalk is based on the very successful work of the Learning Research Group at Xerox's Palo Alto Research Center [4]. The original Smalltalk was created by Alan Kay and the LRG to serve as the communication medium for the Dynabook, a personal computer the size of a notebook "with the power to handle virtually all of its owner's information-related needs". Kay's Dynabook would be able to store thousands of pages of text, display images with resolution surpassing newsprint, and perform real-time audio synthesis with similarly high fidelity. Although the envisioned Dynabook does not yet exist, Smalltalk has been running on minicomputers ("Interim Dynabooks") at PARC since 1972. Hundreds of visitors at PARC have learned Smalltalk; programmers and nonprogrammers alike have found the language both friendly and powerful. Children, animators, musicians, secretaries, and administrators have used Smalltalk to build their own personalized tools with relative ease.

Don't cointain pseudocode for interpreters, but contain really thourough and clear description of language, with examples

"My intent was that we should eventually make the recognizers look more like BNF (there are examples of what was in mind in the history), but we never did.

However, Scott Warren (who lurks on this list) a few years later did "Rosetta Smalltalk" in which he and his friends correctly figured out where this style should go, and implemented a version in a very nice and very pretty system on an 8-bit micro"

- AK

"Scott Warren" and "his friends" are: Scott Warren, Dennis Abe, and Intel Corporation

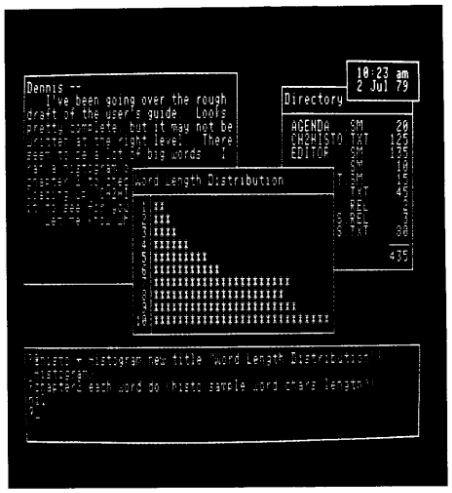


Figure 1. Multiple independent CRT windows.

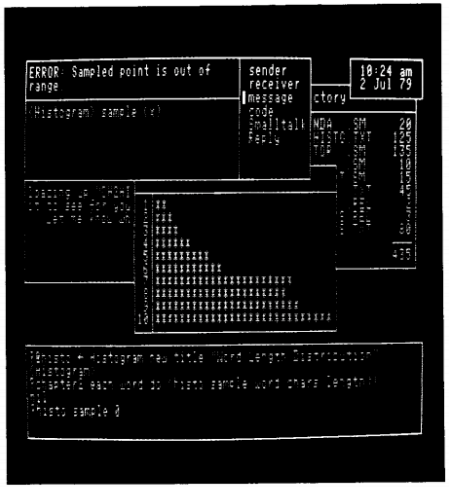
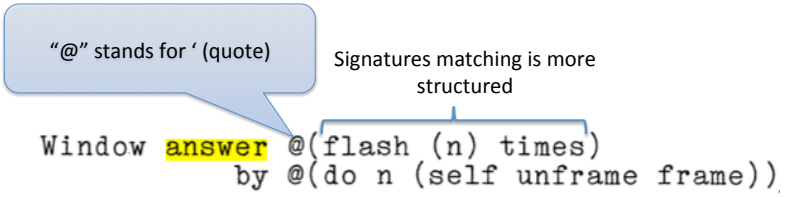
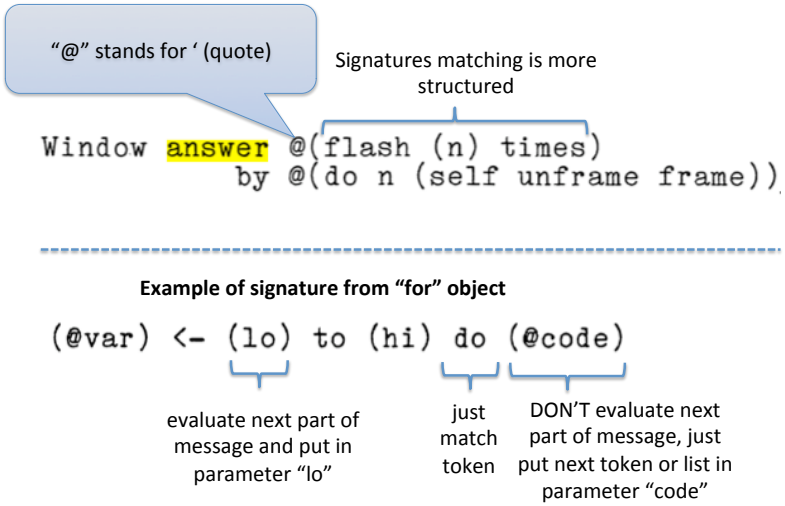


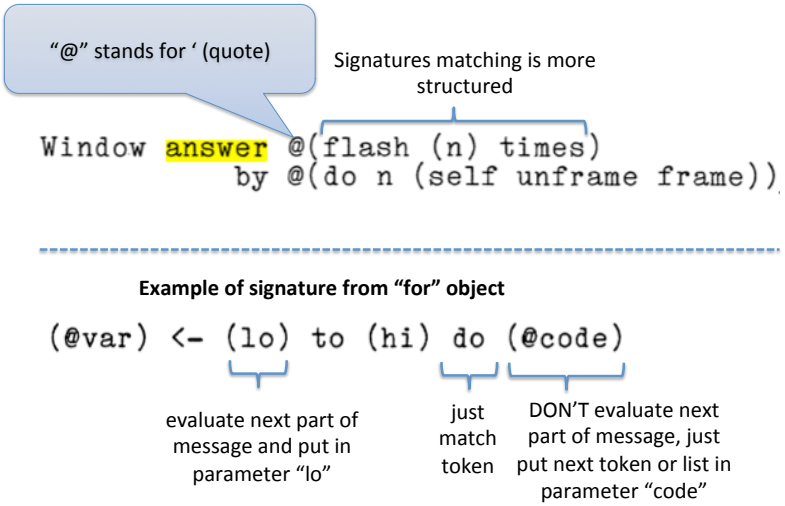
Figure 2. A diagnostic window.

two main differences:

1) how matching of messages is specified and performed
2) "to" creates objects (not classes) and use of "new" is explicit







The order of the matches is automatic and it goes from most specific to most generic. *Interpreter must take care of extra "matching" logic, not as "plain" as ST72 interpreter.*

```
to (flash (win)) (do 10 (win unframe frame))
```

"to" transparently creates a class
and returns the flash OBJECT


```
to (flash (win)) (do 10 (win unframe frame))
```

"to" transparently creates a class
and returns the flash OBJECT

To create a class, give it a method and create an instance of the class:

```
@Stack <- Class new!
Stack answer @(empty) by @(top = 0)!

@s <- Stack new 10!
```


```
to (flash (win)) (do 10 (win unframe frame))
```

"to" transparently creates a class
and returns the flash OBJECT

To create a class, give it a method and create an instance of the class:

still "cruel assignment"

```
@Stack <- Class new!
Stack answer @(empty) by @(top = 0)!

@s <- Stack new 10!
```

LISP-itis symptoms


```
to (flash (win)) (do 10 (win unframe frame))
```

"to" transparently creates a class
and returns the flash OBJECT

To create a class, give it a method and create an instance of the class:

```
@Stack <- Class new!
Stack answer @(empty) by @(top = 0)!
Stack idict <- @(array top) tdict <- @(a 1 x)!
@s <- Stack new 10!
```

Instance variables and temps are explicitly added (instead of <temps> | <instance> notation of ST72)

What we have:

```
@Stack <- Class new!
Stack answer @(empty) by @(top = 0)!
Stack idict <- @(array top) tdict <- @(a 1 x)!
@s <- Stack new 10!
```

1) Less cruel assignment operator
2) No need to declare temps
3) Use dot notation instead of ‘s
4) Cure for LISP-itis

What we have:

```
@Stack <- Class new!
Stack answer @(empty) by @(top = 0)!
Stack idict <- @(array top) tdict <- @(a 1 x)!
@s <- Stack new 10!
```

What we’d like:

```
Stack <= Class new!
Stack answer empty by top = 0!
Stack idict <- @(array top) tdict <- @(a 1 x)!
s <= Stack new 10!
```

1) Less cruel assignment operator
2) No need to declare temps
3) Use dot notation instead of ‘s
4) Cure for LISP-itis

Assume temp unless parameter or global. Temp only live in this scope.

Use indentation and “:” as quote (sometimes!)

```
Number answer:
  __factorial
  by:
  __if self == 0:
  _____return 1
  __else:
  _____(self - 1) factorial * self
console print 3 factorial
```

```
Number answer @(factorial) by @(if self == 0
@(return 1) else @((self - 1) factorial * self))

console print 3 factorial
```

Fizz Buzz:

```
for each i in: (1...100) do:
   if 0 == i % 15:
      console print "FizzBuzz "
   else if 0 == i % 3:
      console print "Fizz "
   else if 0 == i % 5:
      console print "Buzz "
   else:
      console print i + " "
```

Closures (read only):

```
op1 = 2
codeToBeRun = '(console print (op1+op2))
op2 = 3
codeToBeRun eval
op2 = 6
codeToBeRun eval
op1 = 1000
codeToBeRun eval

> 5 8 8
```

Homoiconicity:

```
codeToBeRun = ' (console print 1+2)
codeToBeRun[3] = '-
codeToBeRun eval

> -1
```

Any token can contain any value:

```
temp=true
true=false
false=temp
If true:
___console print 1
else:
___console print 2
 > 2
```

The tokens "true" and "false" anywhere in the program will now contain the opposite boolean object. 1==0 will still be the object false

```
"world" = "Dave"
 console print "Hello "
 console print "world"

> "Hello Dave"
```

and more…

- unicode support means that objects and messages can be emojis
- macros…
- recursive interpreter, ***but yielding***

Still tricky…

- ‘ (quote) still coming up on occasions
- standard plain arithmetic precedence goes out of the window
- toy language (error detection, error reporting, more examples…)