The History of Scheme

Guy Steele
Sun Microsystems Laboratories

October 2006

It Started with Lisp

- Atoms

```
HELLO FIRST COND 0 27 -13
```

- Lists

```
(RED GREEN BLUE)
(THIS IS A LIST OF 7 ATOMS)
(COLORS (RED GREEN BLUE)
 SOUNDS (BUZZ CLANG)
 TASTES (SWEET SOUR SALTY BITTER))
()
(PLUS 3 4)
```

Lists Are a Great Way to Write Code

`(PLUS 3 (TIMES 4 5))` evaluates to `23`

```
(DEFINE LENGTH (X)
  (COND ((NULL X) 0)
        (T (PLUS 1 (LENGTH (REST X))))))
```

`(LENGTH (RED GREEN BLUE))`
is an error: no function named `RED`

`(LENGTH (QUOTE (RED GREEN BLUE)))`
evaluates to `3`

NOTE: Notation has been modernized for modern audiences!

Constructing Lists

```
(CONS (QUOTE RED) (QUOTE (GREEN BLUE)))
```
evaluates to `(RED GREEN BLUE)`

```
(CONS (QUOTE RED)
      (CONS (QUOTE GREEN)
            (CONS (QUOTE BLUE)
                  (QUOTE ()))))
```
evaluates to `(RED GREEN BLUE)`

```
(LIST (QUOTE RED)
      (QUOTE GREEN)
      (QUOTE BLUE))
```
evaluates to `(RED GREEN BLUE)`

Lambda Expressions as Functions

```
((LAMBDA (X Y) (LIST X Y X))
 (QUOTE RUN)
 (QUOTE LOLA))
```

evaluates to `(RUN LOLA RUN)`

because when the lambda expression is used as a function, its **body**, the expression (LIST X Y X), is evaluated in an **environment** in which the value of the **parameter** X is RUN and the value of the parameter Y is LOLA

Functional Arguments

```
(DEFINE MAP (FN X)
  (COND ((NULL X) (QUOTE ()))
        (T (CONS (FN (FIRST X))
                 (MAP FN (REST X))))))
```

applies the function FN to each element of the list X
and returns a list of the results

```
(MAP (QUOTE (LAMBDA (X) (LIST X X)))
     (QUOTE (CHITTY BANG)))
```

evaluates to `((CHITTY CHITTY) (BANG BANG))`

The Evaluator Is Easy to Write in Lisp

```
(DEFINE EVAL (X ENV)
  (COND ((NUMBERP X) X)
        ((ATOM X) (LOOKUP X ENV))
        ((EQ (FIRST X) (QUOTE QUOTE))
         (SECOND X))
        ((EQ (FIRST X) (QUOTE COND))
         (EVCOND (REST X) ENV))
        (T (APPLY (FIRST FORM)
                  (EVLIS (REST FORM)
                         ENV)
                  ENV))))
```

Representation of Environments

An “environment” is just a list of name-value pairs:

```
((N 3) (X (RED GREEN BLUE)) (Y HELLO))
```

We say that the name N is **bound** to the value 3,
the name X is bound to the value (RED GREEN BLUE),
the name Y is bound to the value HELLO, and so on.

If a name appears more than once, the leftmost pair is used.

```
(DEFINE LOOKUP (X ENV)
  (COND ((NULL ENV) (ERROR))
        ((EQ (FIRST (FIRST ENV)) X)
         (SECOND (FIRST ENV)))
        (T (LOOKUP X (REST ENV)))))
```

Conditionals and Argument Lists

```
(DEFINE EVCOND (C ENV)
  (COND ((NULL C) (QUOTE ()))
        ((ATOM (FIRST C)) (ERROR))
        ((EVAL (FIRST (FIRST C)) ENV)
         (EVAL (SECOND (FIRST C)) ENV))
        (T (EVCOND (REST C) ENV))))

(DEFINE EVLIS (X ENV)
  (COND ((NULL X) (QUOTE ()))
        (T (CONS (EVAL (FIRST X) ENV)
                 (EVLIS (REST X)
                        ENV)))))
```

Applying Functions

```
(DEFINE APPLY (FN ARGS ENV)
  (COND ((NULL FN) (ERROR))
        ((PRIMOP FN) (PRIMAPP FN ARGS))
        ((ATOM FN)
         (APPLY (LOOKUP FN ENV)
                ARGS ENV))
        ((EQ (FIRST FN) (QUOTE LAMBDA))
         (EVAL (THIRD FN)
               (BIND (SECOND FN)
                     ARGS ENV)))
        (T (APPLY (EVAL FN ENV)
                  ARGS ENV))))
```

Applying Primitive Functions

```
(DEFINE PRIMAPP (FN ARGS)
  (COND ((EQ FN (QUOTE FIRST))
          (FIRST (FIRST ARGS)))
         ((EQ FN (QUOTE SECOND))
          (SECOND (FIRST ARGS)))
         ((EQ FN (QUOTE ATOM))
          (ATOM (FIRST ARGS)))
         ((EQ FN (QUOTE CONS))
          (CONS (FIRST ARGS) (SECOND ARGS)))
         ((EQ FN (QUOTE PLUS))
          (PLUS (FIRST ARGS) (SECOND ARGS)))
         ((EQ FN (QUOTE LIST)) ARGS)
         ... ))
```

Binding Parameters to Arguments

```
(DEFINE BIND (PARAMS ARGS ENV)
  (COND ((NULL PARAMS)
                (COND ((NULL ARGS) ENV)
                      (T (ERROR))))
         ((NULL ARGS) (ERROR))
         (T (CONS (LIST (FIRST PARAMS)
                         (FIRST ARGS))
                   (BIND (REST PARAMS)
                         (REST ARGS)
                         ENV)))))
```

The “Funarg Problem”

```
(DEFINE MAP (FN X)
  (COND ((NULL X) (QUOTE ()))
        (T (CONS (FN (FIRST X))
                 (MAP FN (REST X))))))

(DEFINE CONSALL (X YS)
  (MAP (QUOTE (LAMBDA (Y) (CONS X Y))) YS))

(CONSALL (QUOTE BEAT)
         (QUOTE (HARVARD YALE)))
```

we expect to get:

```
  ((BEAT HARVARD) (BEAT YALE))
```

we actually get:

```
  (((HARVARD YALE) HARVARD) ((YALE) YALE))
```

Fixing the Evaluator

```
(DEFINE EVAL (X ENV)
  (COND ((NUMBERP X) X)
        ...
        ((EQ (FIRST X) (QUOTE FUNCTION))
         (LIST (QUOTE FUNARG)
               (SECOND X)
               ENV)))
        ...
        (T (APPLY (FIRST FORM)
                  (EVLIS (REST FORM)
                         ENV)
                  ENV))))
```

Fixing Function Application

```
(DEFINE APPLY (FN ARGS ENV)
  (COND ((NULL FN) (ERROR))
        ...
        ((EQ (FIRST FN) (QUOTE FUNARG))
         (APPLY (SECOND FN)
                ARGS
                (THIRD FN)))
        ...
        (T (APPLY (EVAL FN ENV)
                  ARGS ENV))))
```

The Funarg Solution

```
(DEFINE MAP (FN X)
  (COND ((NULL X) (QUOTE ()))
        (T (CONS (FN (FIRST X))
                 (MAP FN (REST X))))))

(DEFINE CONSALL (X YS)
  (MAP (FUNCTION (LAMBDA (Y) (CONS X Y))) YS))

(CONSALL (QUOTE BEAT)
         (QUOTE (HARVARD YALE)))
```

we **actually** get:

```
  ((BEAT HARVARD) (BEAT YALE))
```

as desired

Objects and Actors

- Inspired in part by SIMULA and Smalltalk, Carl Hewitt developed a model of computation around “actors”
  - Every agent of computation is an actor
  - Every datum or data structure is an actor
  - An actor may have “acquaintances” (other actors it knows)
  - Actors react to messages sent from other actors
  - An actor can send messages only to acquaintances and to actors received in messages
- “You don’t add 3 and 2 to get 5; instead, you send 3 a message asking it to add 2 to itself”

Factorial Function in Lisp

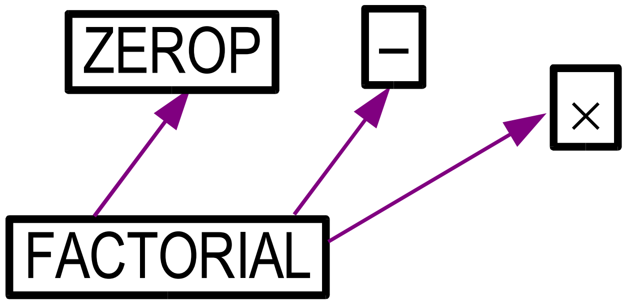
```
(DEFINE FACTORIAL (N)
  (COND ((ZEROP N) 1)
        (T (TIMES N (FACTORIAL
                      (DIFFERENCE
                         N 1))))))
```

returns the product of the integers from 1 to N

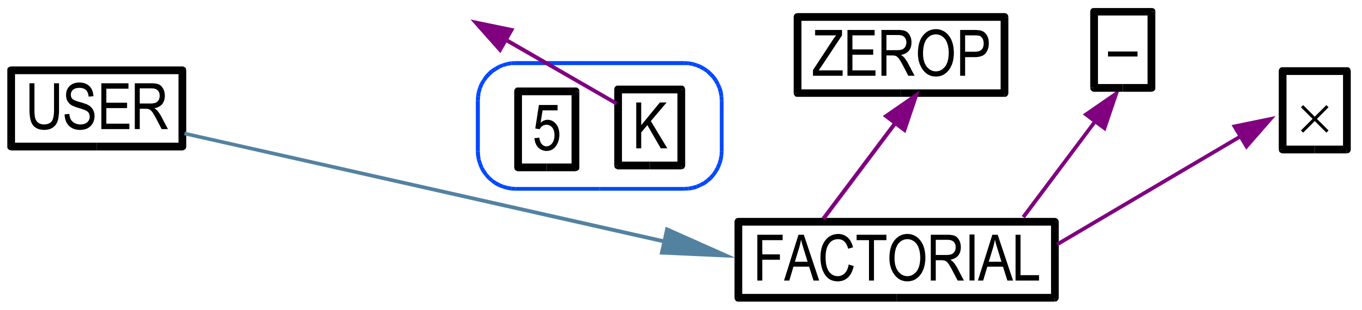
```
(FACTORIAL 5)
```

evaluates to `120`

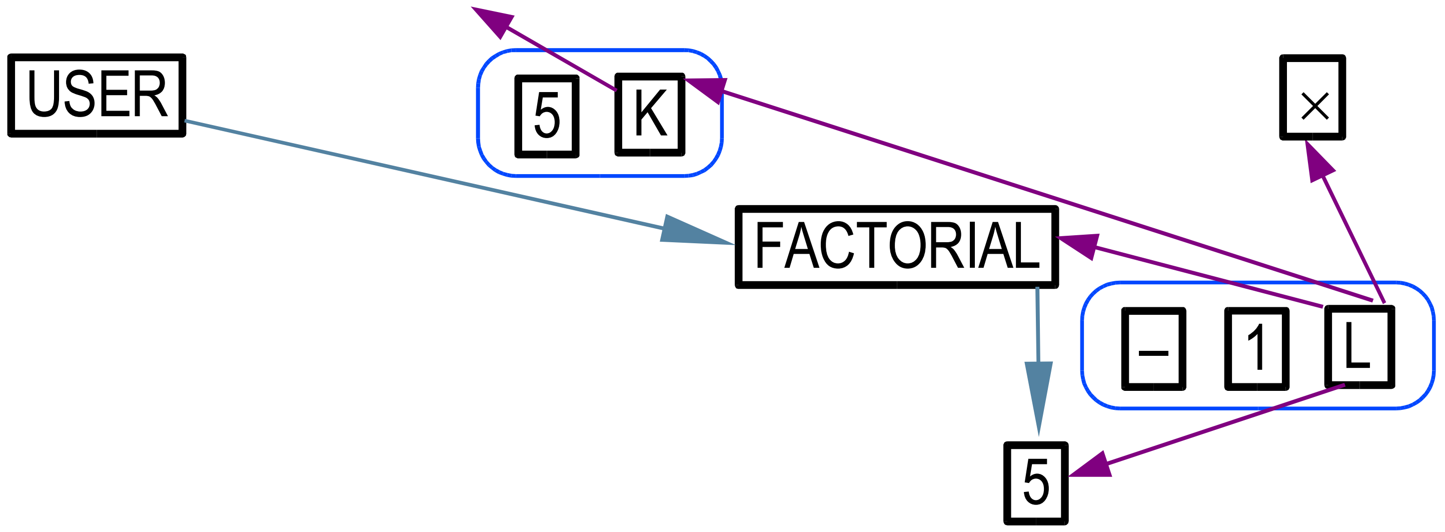
The “Factorial” Actor (1 of 8)



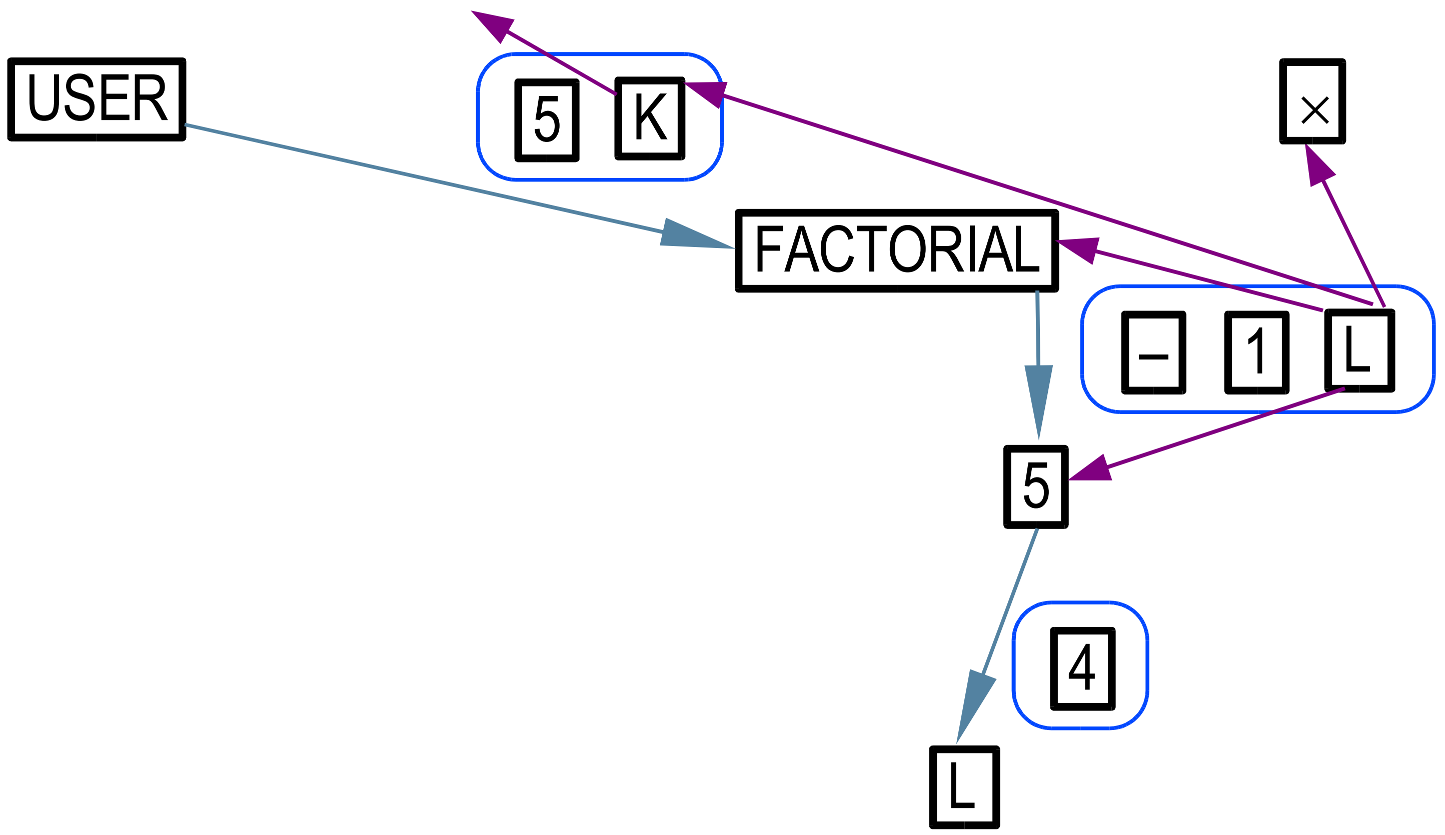
The “Factorial” Actor (2 of 8)



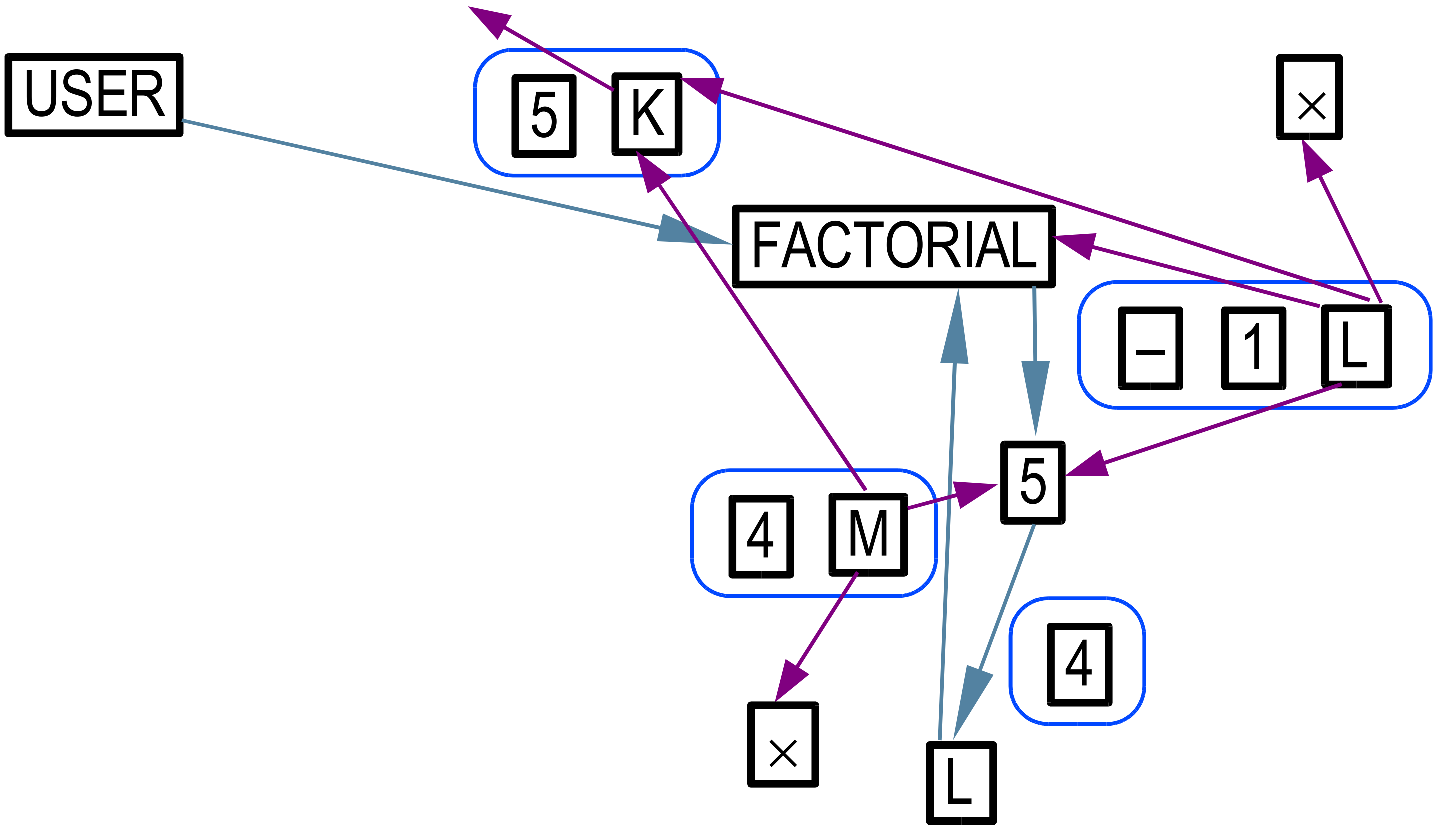
The “Factorial” Actor (3 of 8)



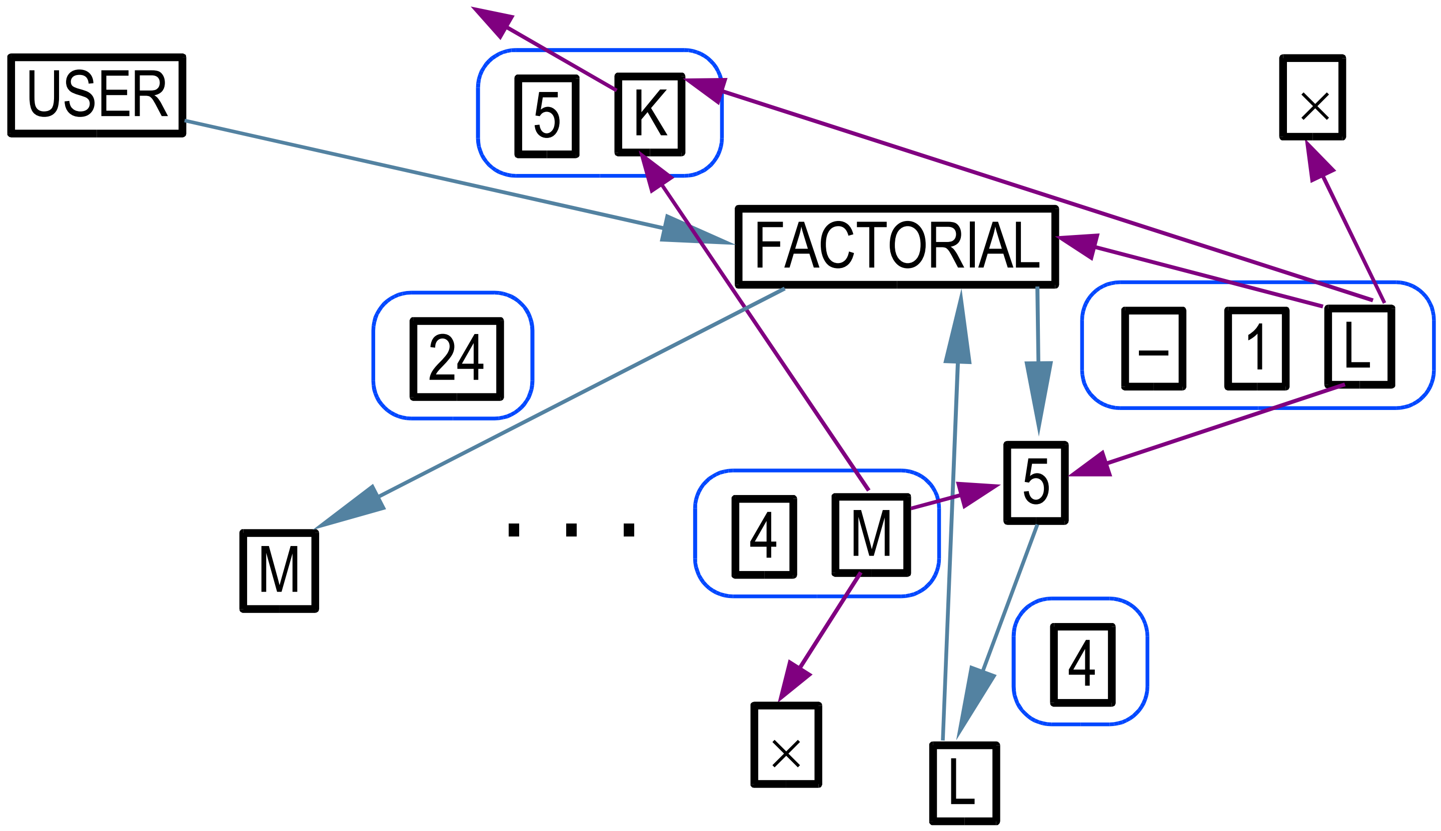
The “Factorial” Actor (4 of 8)



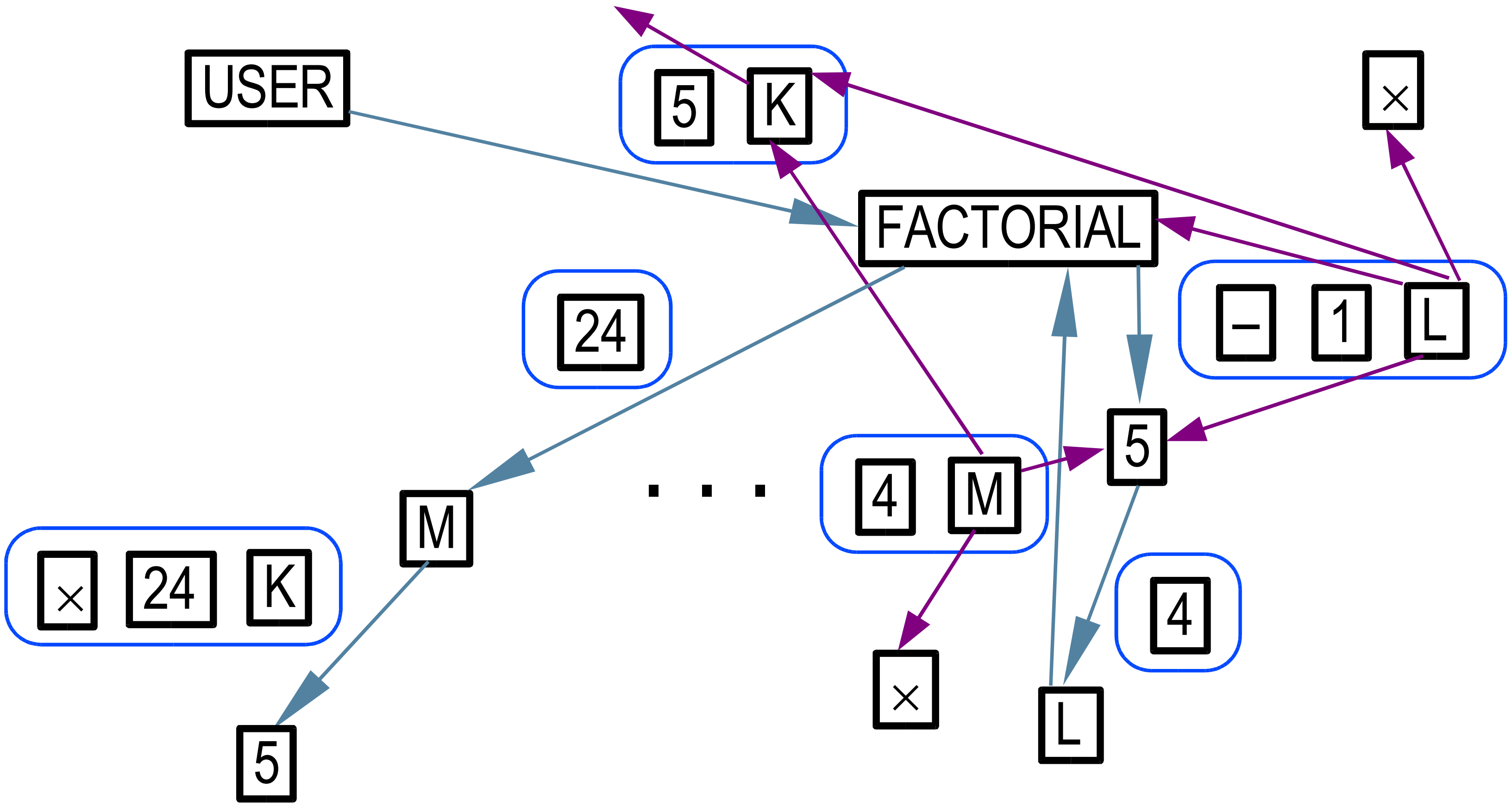
The “Factorial” Actor (5 of 8)



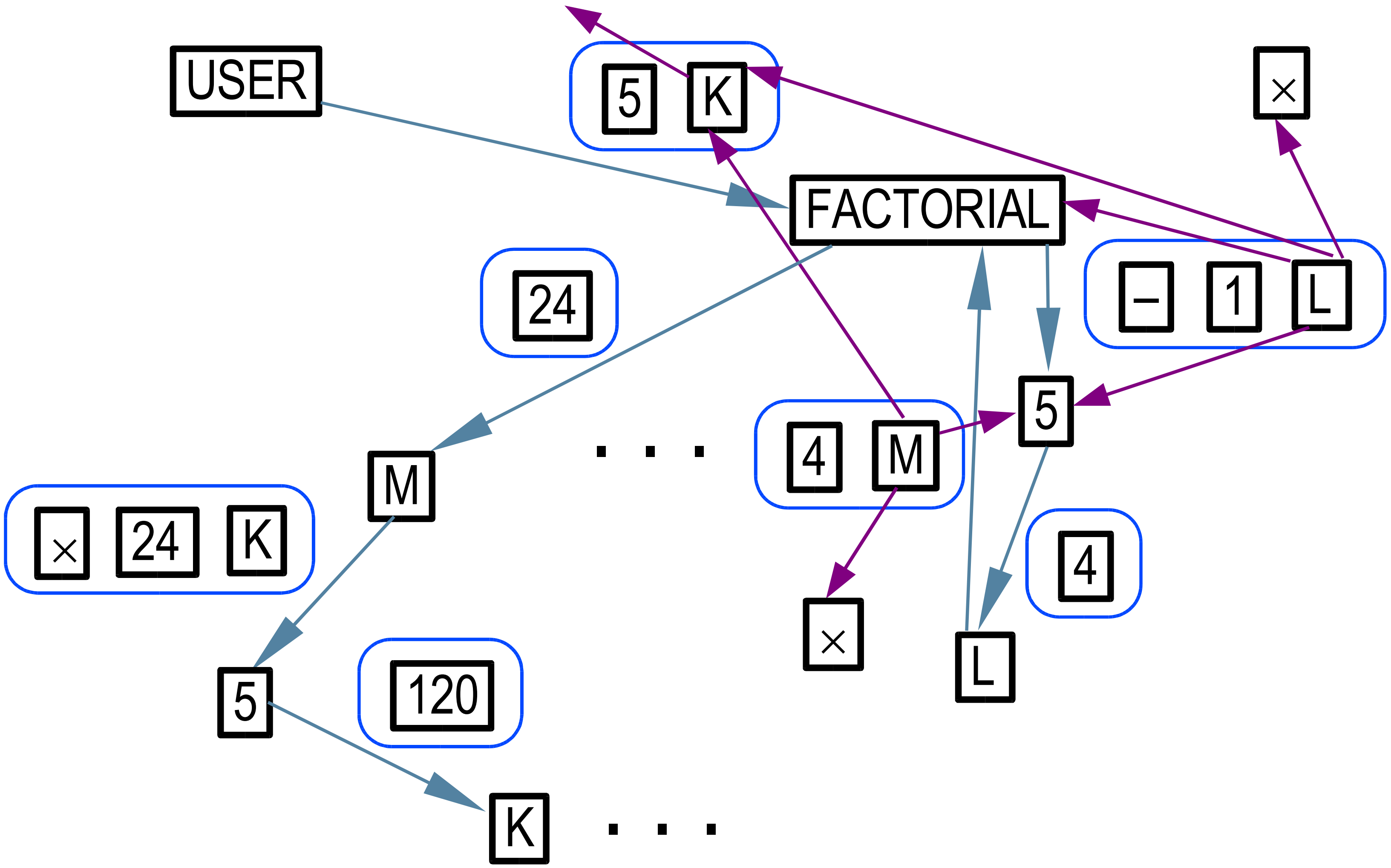
The “Factorial” Actor (6 of 8)



The “Factorial” Actor (7 of 8)



The “Factorial” Actor (8 of 8)



Factorial in the PLASMA Language

Carl had an actors-based language with a difficult syntax, described with what seemed like difficult terminology.

```
(define
  [factorial ≡
    (≡≡> (message: [=n] (reply-to: =c))
      (rules n
        (≡> 1 (c <== (message: 1)))
        (else (factorial <==
                   (message: (n - 1)
                    (reply-to:
                      (≡≡> (message: =y)
                        (c <== (message: (y * n))))))))))])
```

From an early working draft, dated December 1975, of "Viewing Control Structures as Patterns of Passing Messages" by Carl Hewitt; after multiple revisions, this became MIT AI Memo 410 in December 1976.

Gerry Sussman and I wanted to understand
Carl Hewitt’s ideas,
which seemed to have intellectual power,
but we couldn’t get past the complexity and the notation
to see “what was really going on.”

So we decided to implement a “toy” actors language.

We hoped that it could capture the essence of the ideas
while remaining simple enough to understand.

It might even turn into something useful.

A Sequence of AI Languages at MIT

LISP (McCarthy et al., 1958)

METEOR (Bobrow, 1964)

CONVERT (Guzman, 1969)

PLANNER (Hewitt, 1969)

MUDDLE (Sussman, Hewitt, et al., 1970)

MICROPLANNER (Sussman et al., 1971)

CONNIVER (Sussman et al., 1972)

PLASMA (Hewitt et al., 1973)

SCHEMER (Sussman and Steele, 1975)

We decided to start with a small Lisp interpreter and then graft on exactly two more constructs:
a way to make actors and a way to send messages.

Gerry had been studying and teaching Algol 60,
so we decided to use the full funarg solution
so that our toy language would have lexical scope.
Our intuition was that this would also
keep track of actor’s acquaintances correctly.

(Also inspired by Algol 60, the first toy interpreter
was call-by-name rather than call-by-value!
I will gloss over that distinction here.)

For making an actor, we chose the syntax

`(alpha (`*parameters*`)` *body*`)`

It would be just like a lambda expression, but its body had to send a message rather than return a value.

For sending messages, we considered

`(send` *actor* *argument* `...` *argument*`)`

but then realized `apply` could tell actors from functions and so we could just use the same keyword-free syntax for calling functions and sending messages.

We would also need some primitive actors.
We decided on (among others):

`(* m n k)`	send product of m and n to actor k
`(- m n k)`	send difference of m and n to actor k
`(= m n k q)`	if m and n equal, send an empty message to actor k; otherwise send an empty message to actor q

Note that these actors never return a value.
They always send a message to another actor.

We wanted to try out this definition of factorial:

```
(define factorial
  (alpha (n c)
    (= n 0
       (alpha () (c 1))
       (alpha ()
         (- n 1
            (alpha (z)
              (factorial z
                (alpha (y)
                  (* n y c)))))))))
```

Note that this was not very different in structure from Carl’s, but somehow it was much less intimidating to us in the details.

IMPORTANT DISCLAIMER

The original Scheme interpreter was written in a very machine-language-like style of Lisp so as to expose every implementation detail. It was very much like Peter Landin's SECD machine and did not take advantage of recursive function calls in the implementation language.

Here I want to gloss over the implementation details so as not to obscure the main point of this talk. Therefore I am going to show you a simplified version of the interpreter, in the same style as the Lisp interpreter shown on earlier slides.

A Scheme Evaluator

```
(DEFINE EVAL (X ENV)
  (COND ((NUMBERP X) X)
        ((ATOM X) (LOOKUP X ENV))
        ((EQ (FIRST X) (QUOTE QUOTE))
         (SECOND X))
        ((EQ (FIRST X) (QUOTE COND))
         (EVCOND (REST X) ENV))
        ((EQ (FIRST X) (QUOTE LAMBDA))
         (LIST (QUOTE BETA) X ENV))
        (T ((LAMBDA (EV)
              (APPLY (FIRST EV) (REST EV)))
            (EVLIS FORM ENV)))))
```

Applying Functions

```
(DEFINE APPLY (FN ARGS ENV)
  (COND ((NULL FN) (ERROR))
        ((PRIMOP FN) (PRIMAPP FN ARGS))
        ((EQ (FIRST FN) (QUOTE BETA))
         (EVAL (THIRD (SECOND FN))
               (BIND (SECOND (SECOND FN))
                     ARGS
                     (THIRD FN))))
        (T (ERROR))))
```

Everything else (LOOKUP, EVCOND, EVLIS, PRIMOP, PRIMAPP, BIND) is the same as before.

Sending to Primitive Actors (1 of 2)

```
(DEFINE PRIMSEND (ACTOR ARGS)
  (COND ((EQ ACTOR (QUOTE *))
         (APPLY (THIRD ARGS)
                (LIST (TIMES
                        (FIRST ARGS)
                        (SECOND ARGS)))))
        ((EQ ACTOR (QUOTE -))
         (APPLY (THIRD ARGS)
                (LIST (DIFFERENCE
                        (FIRST ARGS)
                        (SECOND ARGS)))))
        ...
```

Sending to Primitive Actors (2 of 2)

```
...
((EQ ACTOR (QUOTE =))
 (APPLY (COND ((EQUAL (FIRST ARGS)
                     (SECOND ARGS))
              (THIRD ARGS))
             (T (FOURTH ARGS)))
        (QUOTE ())))
... ))
```

A Scheme Evaluator with Actors

```
(DEFINE EVAL (X ENV)
  (COND ((NUMBERP X) X)
        ((ATOM X) (LOOKUP X ENV))
        ((EQ (FIRST X) (QUOTE QUOTE))
         (SECOND X))
        ((EQ (FIRST X) (QUOTE COND))
         (EVCOND (REST X) ENV))
        ((EQ (FIRST X) (QUOTE LAMBDA))
         (LIST (QUOTE BETA) X ENV))
        ((EQ (FIRST X) (QUOTE ALPHA))
         (LIST (QUOTE GAMMA) X ENV))
        (T ((LAMBDA (EV)
                 (APPLY (FIRST EV) (REST EV)))
              (EVLIS FORM ENV)))))
```

Sending Messages to Actors

```
(DEFINE APPLY (FA ARGS ENV)
  (COND ((NULL FA) (ERROR))
        ((PRIMOP FA) (PRIMAPP FA ARGS))
        ((PRIMACTOR FA) (PRIMSEND FA ARGS))
        ...
        ((EQ (FIRST FA) (QUOTE GAMMA))
         (EVAL (THIRD (SECOND FA))
               (BIND (SECOND (SECOND FA))
                     ARGS
                     (THIRD FA))))
        (T (ERROR))))
```

Now evaluating the message send

`(factorial 5 fred)`

results in sending a message containing `120` to the actor named `fred`.

Oh, joy!

A Startling Equivalence

```
(DEFINE APPLY (FA ARGS ENV)
  (COND ...
        ((EQ (FIRST FA) (QUOTE BETA))
         (EVAL (THIRD (SECOND FA))
               (BIND (SECOND (SECOND FA))
                     ARGS
                     (THIRD FA))))
        ((EQ (FIRST FA) (QUOTE GAMMA))
         (EVAL (THIRD (SECOND FA))
               (BIND (SECOND (SECOND FA))
                     ARGS
                     (THIRD FA))))
        (T (ERROR))))
```

An Astonishing Conclusion

Actor constructors and lambda expressions
in our toy language are operationally equivalent.

Does it follow that actors are “merely” functions
in a tail-recursive, lexically scoped language?

They are the same mechanism.
Any difference is not inherent, but depends only
on what you put in their bodies.

If your primitive operators are functions,
you will tend to write programs in a functional style.
If your primitive operators are actors,
you will tend to write programs in an actor style.

A New Language Is Born

After some discussion, Carl Hewitt agreed with our conclusions (with two minor exceptions).

In a way, this ended the “language competition.”

Our great new AI language “Schemer” turned out to be a small dialect of Lisp with some nice properties.

Oh, yes: the name?

File names in that OS were limited to 6 characters.

“SCHEME”

Success

I wrote a compiler for Scheme (called Rabbit).

Soon, other people built much better implementations of Scheme.

Lots of other people found it useful.

The Scheme standard is in its sixth revision.

Very few people bother to cite our papers anymore.

We are delighted.

guy.steele@sun.com