# Fast and Dynamic

Maxime Chevalier-Boisvert

Strange Loop 2013

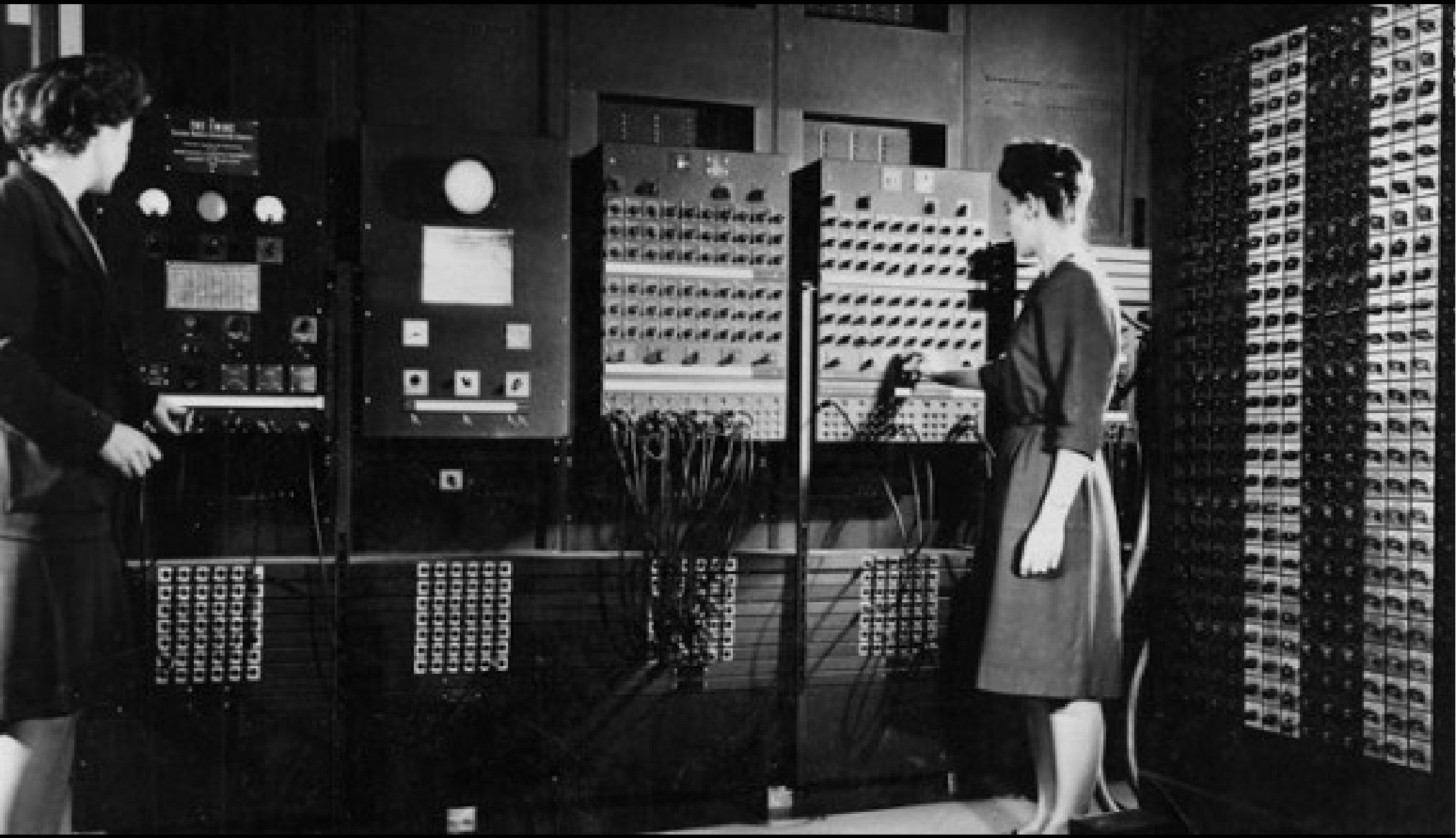
Université de Montréal

"I think of a programming language as a tool to convert a programmer's mental images into precise operations that a machine can perform. The main idea is to match the user's intuition as well as possible. […]"

– Donald Knuth

"I think of a programming language as a tool to convert a programmer's mental images into precise operations that a machine can perform. The main idea is to **match the user's intuition as well as possible**. […]"

– Donald Knuth



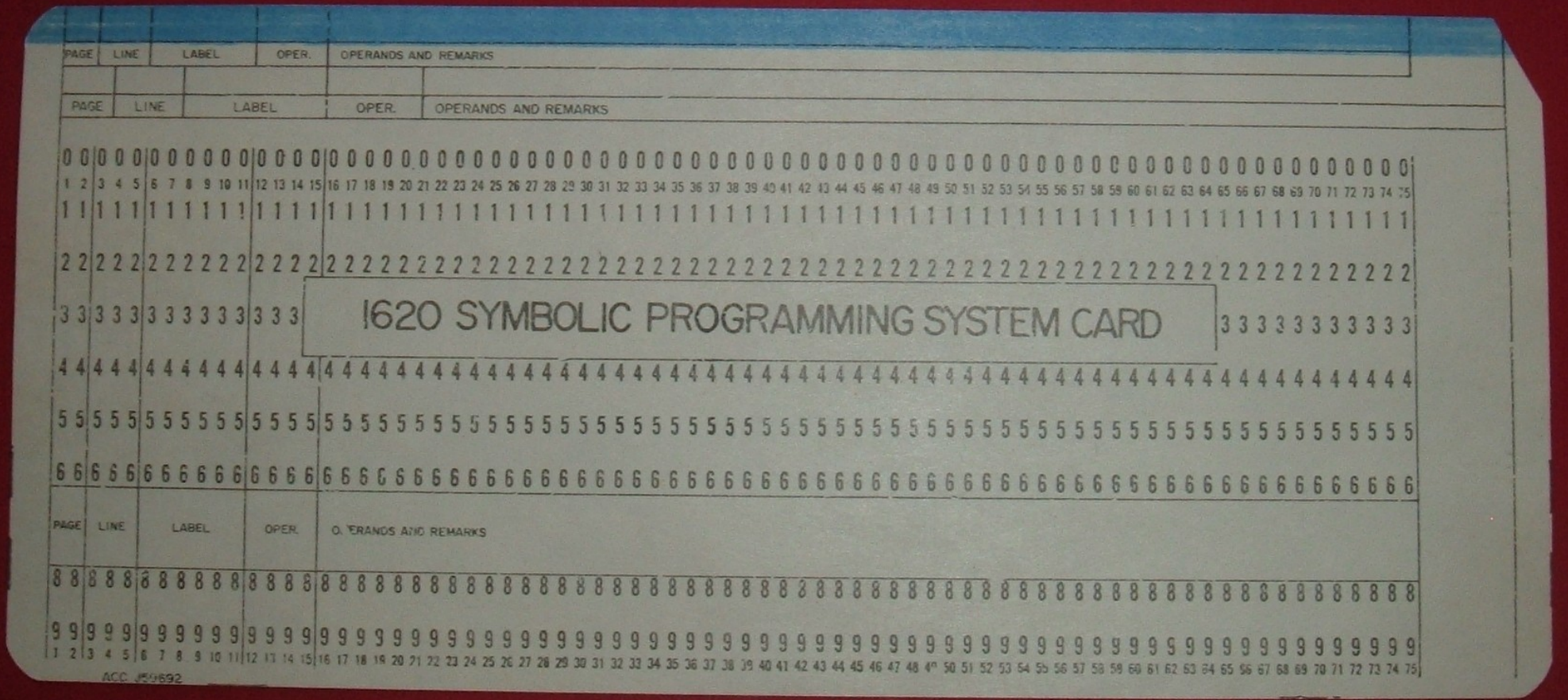



Photo by Arnold Reinhold

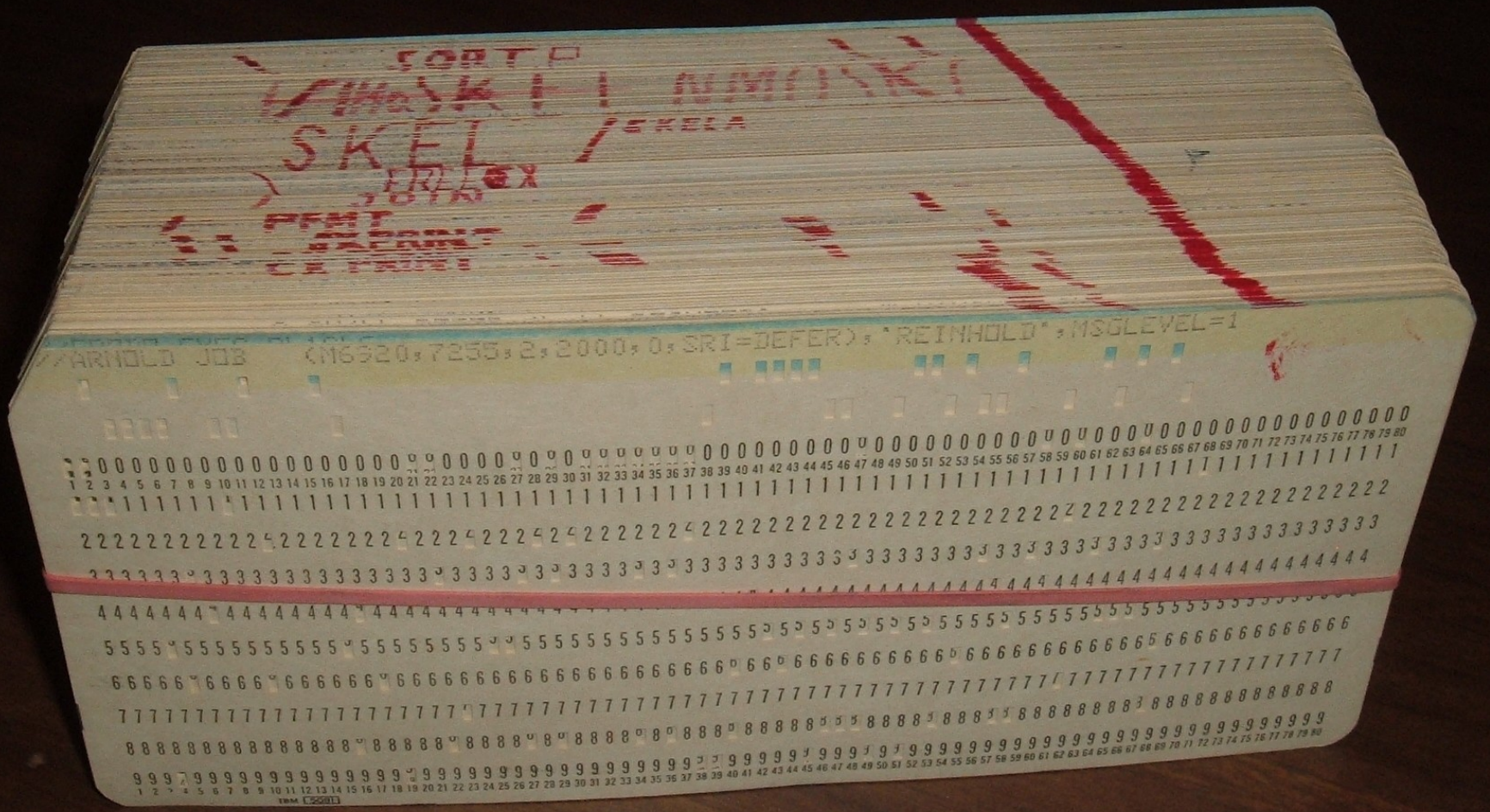



Photo by Arnold Reinhold

Photo by Arnold Reinhold

"Much of my work has come from being lazy. I didn't like writing programs, and so, when I was working on the IBM 701, writing programs for computing missile trajectories, I started work on a programming system to make it easier to write programs."

– John Backus

# Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I

JOHN MCCARTHY, *Massachusetts Institute of Technology, Cambridge, Mass.*

## 1. Introduction

A programming system called LISP (for LISt Processor) has been developed for the IBM 704 computer by the Artificial Intelligence group at M.I.T. The system was designed to facilitate experiments with a proposed system called the Advice Taker, whereby a machine could be instructed to handle declarative as well as imperative sentences and could exhibit "common sense" in carrying out its instructions. The original proposal [1] for the Advice Taker was made in November 1958. The main requirement was a programming system for manipulating expressions representing formalized declarative and imperative sentences so that the Advice Taker system could make deductions.

In the course of its development the LISP system went through several stages of simplification and eventually came to be based on a scheme for representing the partial recursive functions of a certain class of symbolic expressions. This representation is independent of the IBM 704 computer, or of any other electronic computer, and it now seems expedient to expound the system by starting with the class of expressions called S-expressions and the functions called S-functions.

In this article, we first describe a formalism for defining functions recursively. We believe this formalism has advantages both as a programming language and as vehicle for developing a theory of computation. Next, we describe S-expressions and S-functions, give some examples, and

## 2. Functions and Function Definitions

We shall need a number of mathematical ideas and notations concerning functions in general. Most of the ideas are well known, but the notion of *conditional expression* is believed to be new, and the use of conditional expressions permits functions to be defined recursively in a new and convenient way.

a. *Partial Functions.* A partial function is a function that is defined only on part of its domain. Partial functions necessarily arise when functions are defined by computations because for some values of the arguments the computation defining the value of the function may not terminate. However, some of our elementary functions will be defined as partial functions.

b. *Propositional Expressions and Predicates.* A propositional expression is an expression whose possible values are T (for truth) and F (for falsity). We shall assume that the reader is familiar with the propositional connectives $\wedge$ ("and"), $\vee$ ("or"), and $\sim$ ("not"). Typical propositional expressions are:

$$x < y$$

$$(x < y) \wedge (b = c)$$

$$x \text{ is prime}$$

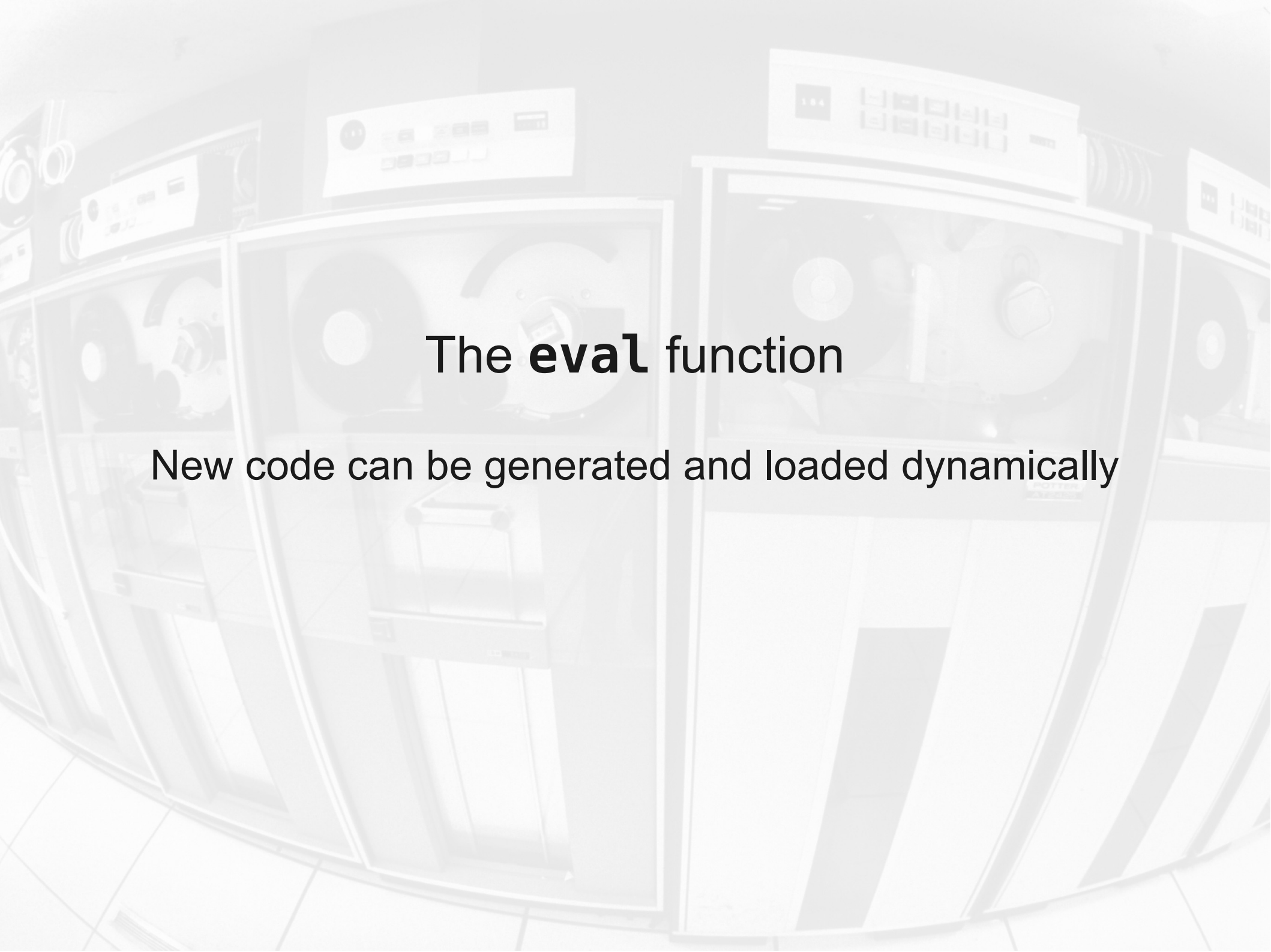
A predicate is a function whose range consists of the truth values T and F.

# Dynamic typing

Types are associated with values, not variables

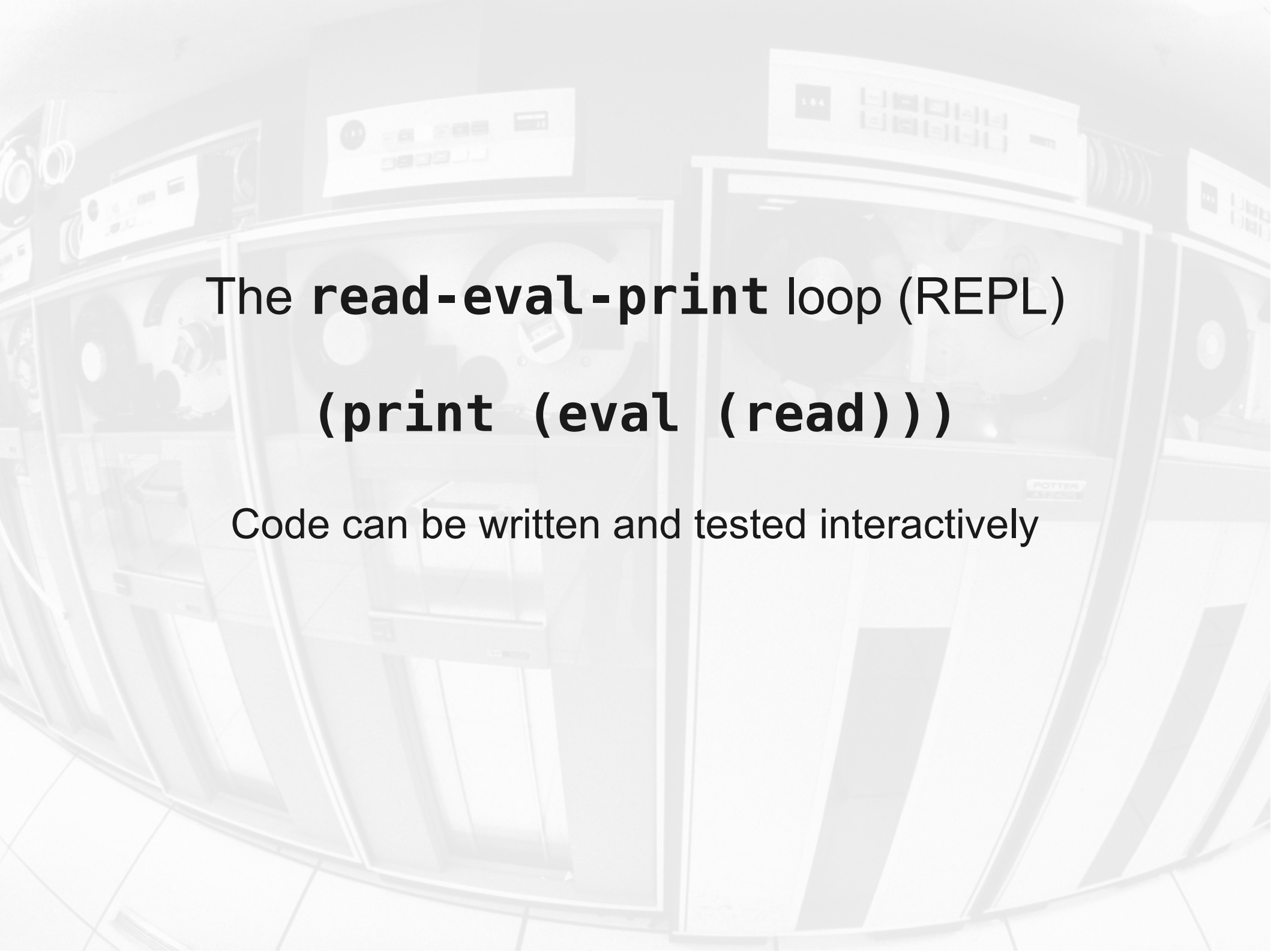
# Late binding

Names are resolved dynamically



# The `eval` function

New code can be generated and loaded dynamically



The **`read-eval-print`** loop (REPL)

**`(print (eval (read)))`**

Code can be written and tested interactively

# Garbage collection

Without it, LISP would have been highly impractical


Why so slow?

# Garbage Collection

- When LISP was invented, GC was in its infancy
- Mark/sweep is inefficient: traverses all objects
- Stop-the-world GCs cause noticeable pauses
- Semi-space GCs can waste memory
  - Half the heap is empty space
  - If the heap is near-full, collect all the time
  - For better performance, need 4x-7x overhead

# Dynamic Type Tests

- Operators change meaning based on types
- Different operation based on input types
- For example, the **+** operator could mean:
  - Integer addition
  - Floating-point addition
  - String concatenation
  - Array appending
- Inside each operator hide many type tests

```
function add1(n)
{
    return n + 1;
}
```

---

```
add1(2)           ⟹ 3

add1('hello')     ⟹ 'hello1'

add1(true)        ⟹ 2
add1(null)        ⟹ 1
add1(undefined)   ⟹ NaN

add1({ toString: function() { return '3'; } }) ⟹ '31'

add1({ toString: function() { return 3; } })   ⟹ 4
```

# Property Access

- Objects can grow dynamically
- Properties can be dynamically deleted
- Property enumeration, introspection possible
- Objects behave like hash table
  - Map property names (string keys) to values
- Hash table lookups are relatively inefficient
  - O(1), but the constant is large
  - Requires you store (key, value) pairs

# `eval`

- `c = a + b`
  - int + int →int
  - string + string →string
  - string + int → string
- Would like a *sufficiently smart* compiler
  - i.e.: a compiler that does type inference
  - Logical reasoning to eliminate unnecessary type tests
- **`eval`** is your compiler's worst nightmare
  - eval(read());
  - Can redefine any function or global variable, add new code
  - The **`eval`** function destroys all type information
- In JavaScript, **`load`** poses a similar problem

# LISP: Type Tagging

- In dynamic languages, all values carry a type
- LISP's strategy: steal low bits for type tags
  - Heuristic: prioritize, integer and list performance
- For example:
  - 32-bit words, lowest 2 bits are type tag
  - 00 means integer, 01 means pair, 11 means pointer
- Better performance for common operations
  - Faster type tests, less allocations, less memory accesses
- Adds overhead to box/unbox values

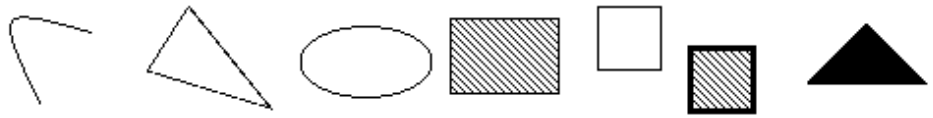
# Digital introduces PDP-11/70. The system all other 11's have been leading up to.

Sample document

? Close Paginate

# XEROX 8010 Star Information System

Star provides integrated text and graphics. A variety of type sizes and styles may be used.



| Description | Price |
|---|---|
| Peas | $0,39 |
| Beans | $0,50 |

Public

Sample document

Letters

Report

Calc

Program

Sample

Close

This is some text in a text frame.

Form field

Button

Printer

In

Out

Blank Document

Basic Graphics

Blank Mail Note

Blank Folder

Blank Record File

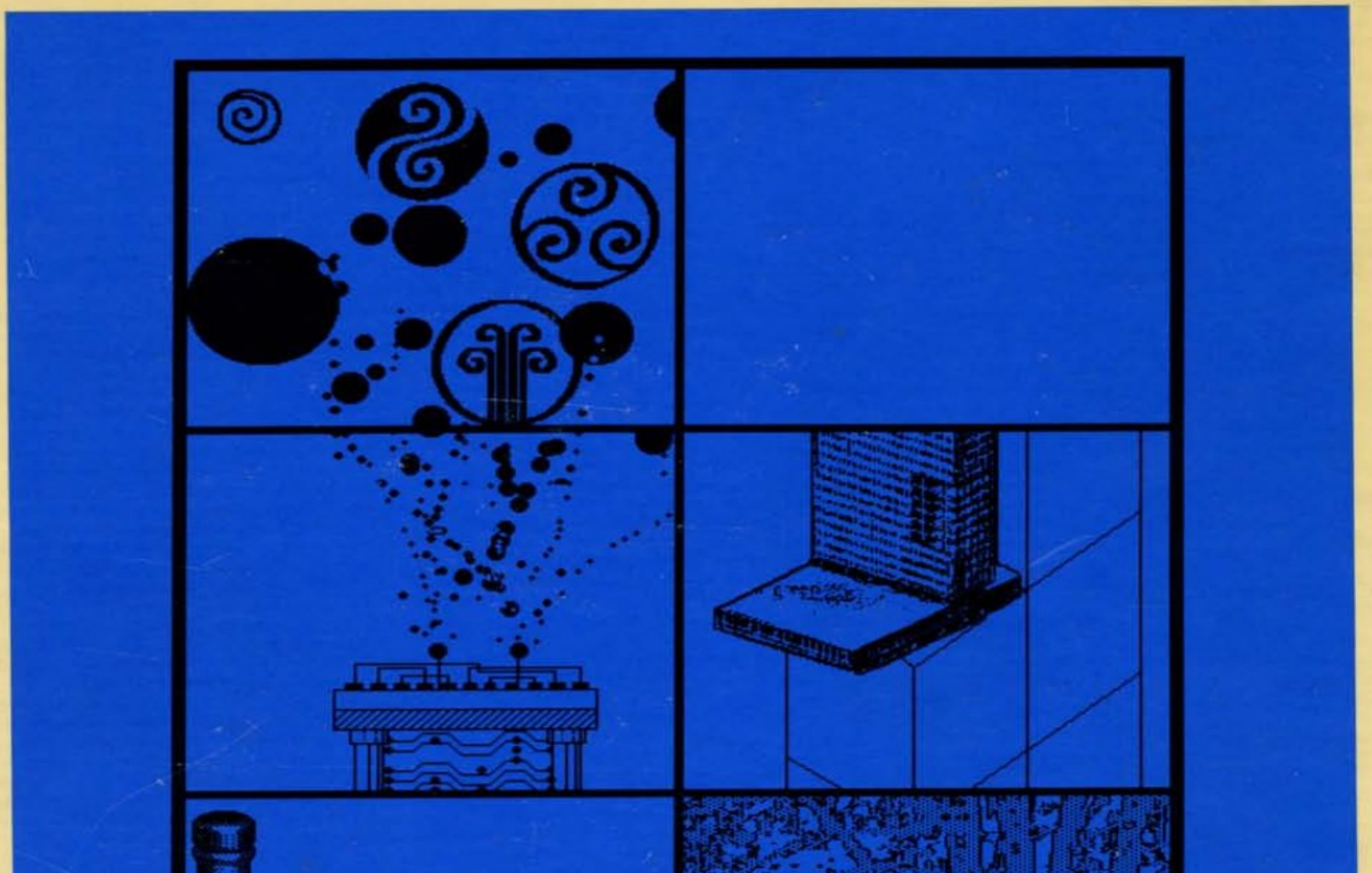
Blank Reference

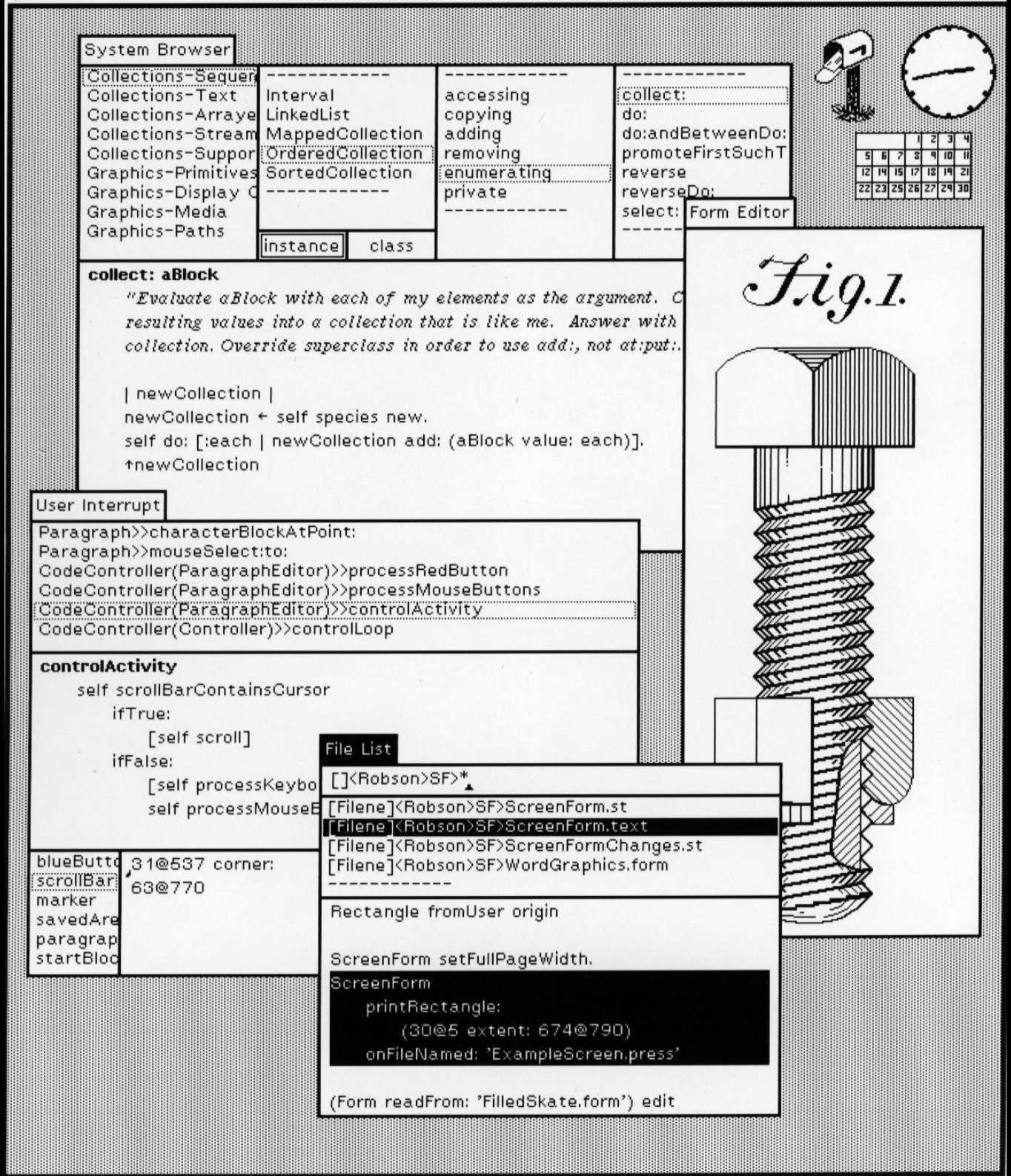
Blank Spread Sheet

Directory

# SMALLTALK-80

## THE LANGUAGE AND ITS IMPLEMENTATION




System Browser
Collections-Sequer
Collections-Text
Collections-Arraye
Collections-Stream
Collections-Suppor
Graphics-Primitives
Graphics-Display C
Graphics-Media
Graphics-Paths
Interval
LinkedList
MappedCollection
OrderedCollection
SortedCollection
instance
class
accessing
copying
adding
removing
enumerating
private
collect:
do:
do:andBetweenDo:
promoteFirstSuchT
reverse
reverseDo:
select:
collect: aBlock
"Evaluate aBlock with each of my elements as the argument. C
resulting values into a collection that is like me. Answer with
collection. Override superclass in order to use add:, not at:put:.
| newCollection |
newCollection ← self species new.
self do: [:each | newCollection add: (aBlock value: each)].
↑newCollection
User Interrupt
Paragraph>>characterBlockAtPoint:
Paragraph>>mouseSelect:to:
CodeController(ParagraphEditor)>>processRedButton
CodeController(ParagraphEditor)>>processMouseButtons
CodeController(ParagraphEditor)>>controlActivity
CodeController(Controller)>>controlLoop
controlActivity
self scrollBarContainsCursor
ifTrue:
[self scroll]
ifFalse:
[self processKeybo
self processMouseB
blueButto
scrollBar
marker
savedAre
paragrap
startBloc
31@537 corner:
63@770
File List
[]<Robson>SF>*
[Filene]<Robson>SF>ScreenForm.st
[Filene]<Robson>SF>ScreenForm.text
[Filene]<Robson>SF>ScreenFormChanges.st
[Filene]<Robson>SF>WordGraphics.form
Rectangle fromUser origin
ScreenForm setFullPageWidth.
ScreenForm
printRectangle:
(30@5 extent: 674@790)
onFileNamed: 'ExampleScreen.press'
(Form readFrom: 'FilledSkate.form') edit
Form Editor
Fig.1.


1978: Thomas Knight, Richard Greenblatt and their CADR LISP Machine

# The LISP Machine

- Goal: make LISP fast, tailor hardware to LISP
- Dynamic operations optimized in hardware
  - Fast path and type tests run in parallel
  - Array bound checks in parallel with array access
- Tagged architecture
  - Memory words have extra tag bits (metadata)
  - Type tags, CDR coding
- Microcoded instructions
  - Call overhead as little as 20 cycles
  - Some primitives directly in microcode
  - Fast incremental garbage collection
- OS written in LISP dialect


1983: Symbolics 3600 Series “L-Machine”

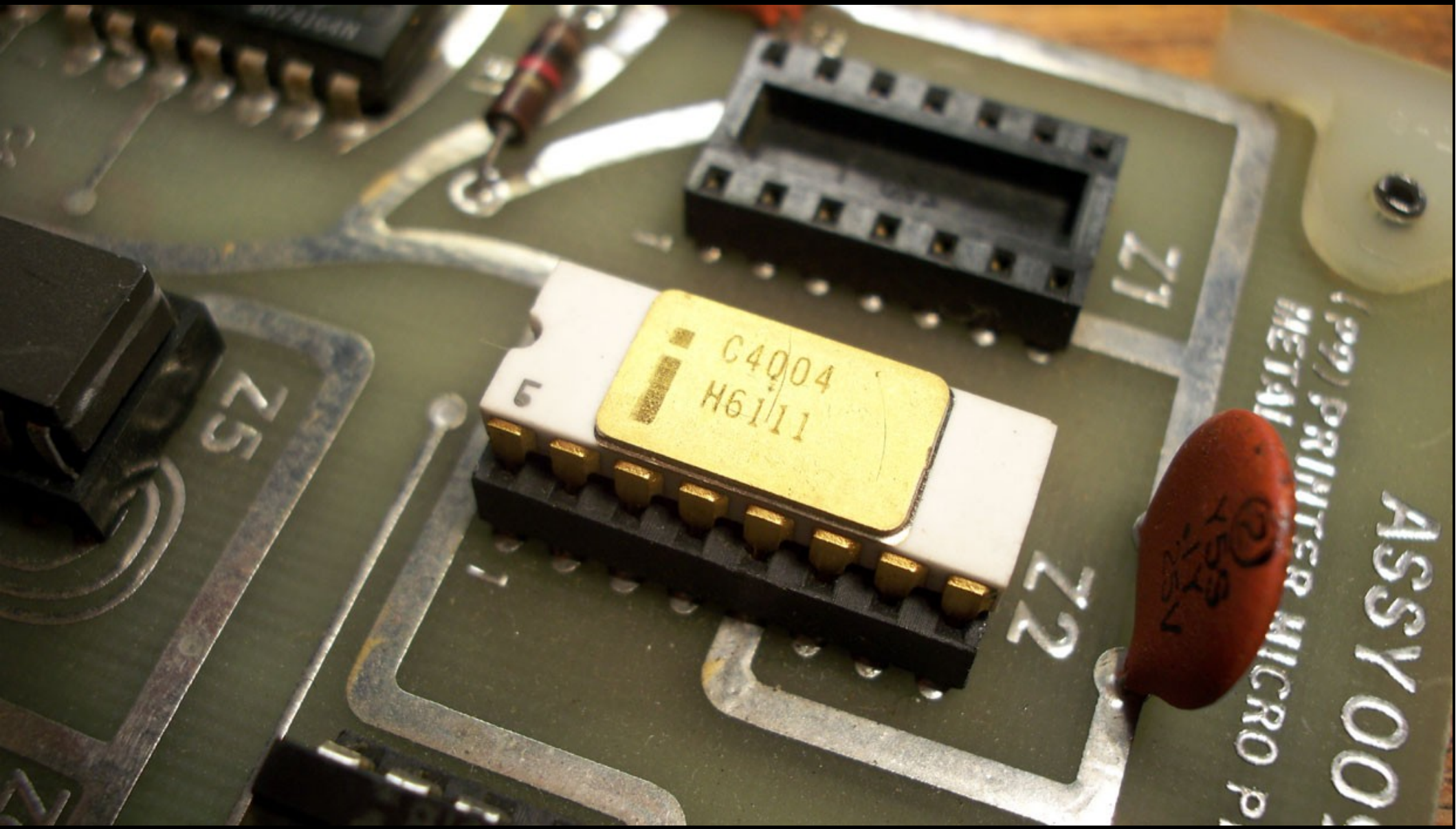
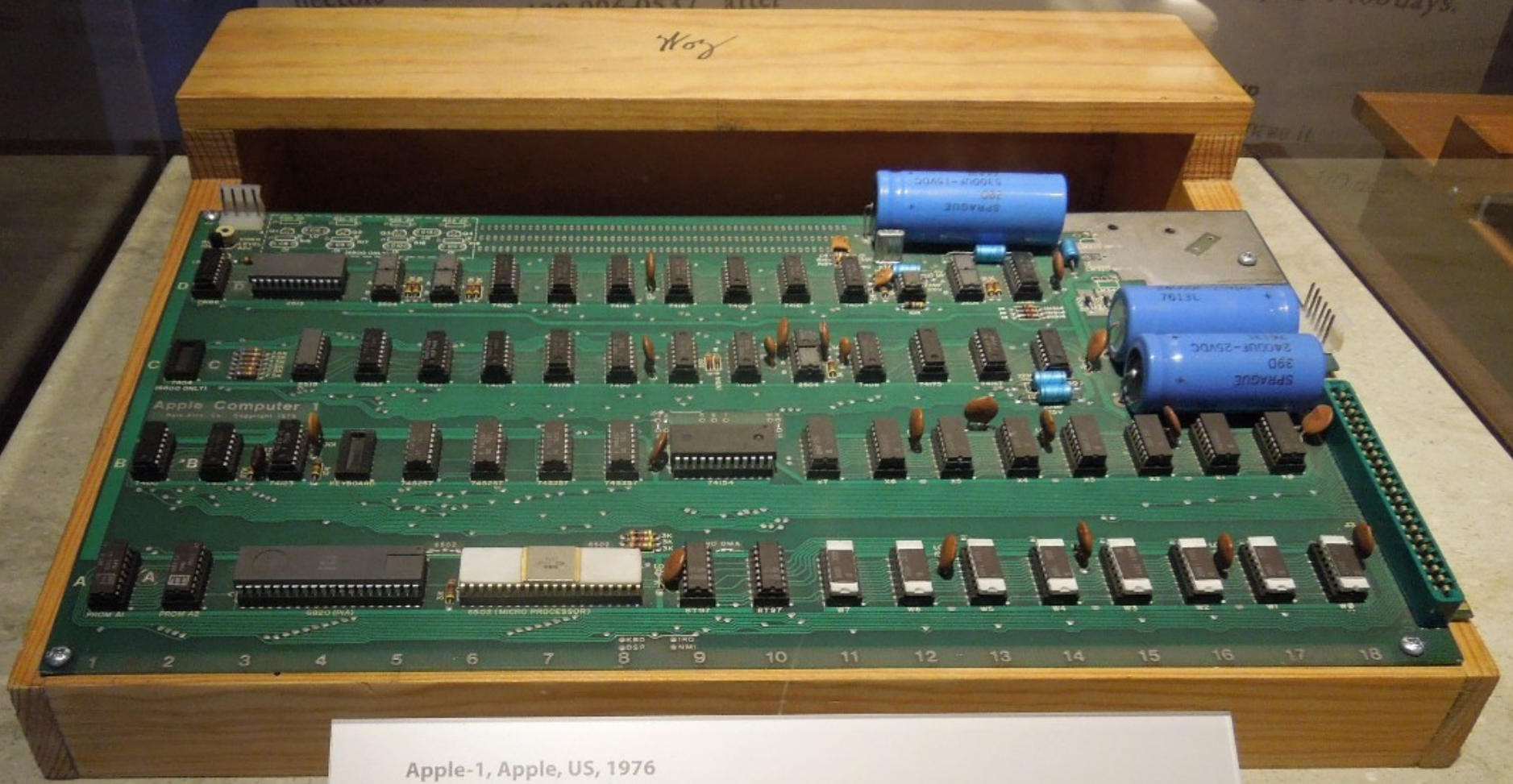



Photo by Paul Ash




**Apple-1, Apple, US, 1976**

Steve Wozniak debuted the prototype Apple-1 at the Homebrew Computer Club in 1976. For $666.66, buyers received a blank printed circuit board, parts kit, and 16-page assembly manual. One had to add a power supply, keyboard, storage system, and display.

**Speed: 1 MHz Memory size: 4K Memory type: Semiconductor Memory width: 8-bit Cost: $666.66**

*Gift of Dysan Corporation, X210.83A*

# Scheme: k-CFA (1981)

- Neil D. Jones, flow analysis of lambda expressions
- Logical deduction based on typing rules
- Extract types from untyped expressions
  - Eliminate type checks, dynamic lookups
- What about **`eval`**?
  - In Scheme, has more restricted semantics
  - Useful, but less applicable for other languages

# SELF (1986)

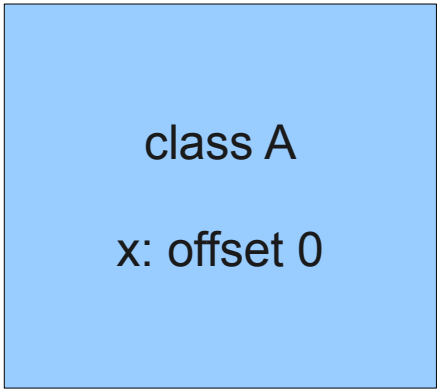
- David Ungar and Randall Smith at Xerox PARC
  - "a Smalltalk that was more Smalltalky than Smalltalk"
- Faster than previous Smalltalk implementations
  - "Up to half the speed of optimized C" *
- Several key innovations:
  - Type feedback, profile-driven optimization
  - Inline caches
  - Inlining of message sends
  - Dynamic deoptimization
  - Maps: optimized object layouts

```
a = {x: 1};
```

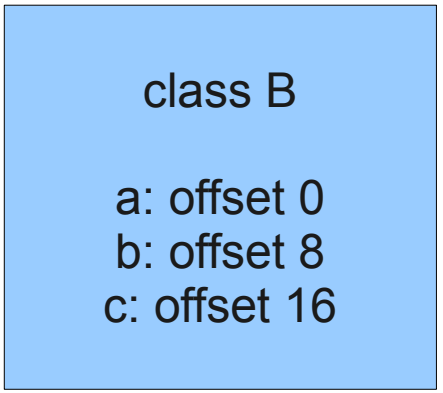
```
b = {a: 1, b: 0.5, c: null};
```

```
a = {x: 1};
// a = new HashMap();
// a.set('x', 1);



b = {a: 1, b: 0.5, c: null};
// b = new HashMap();
// b.set('a', 1);
// b.set('b', 0.5);
// b.set('c', null);
```

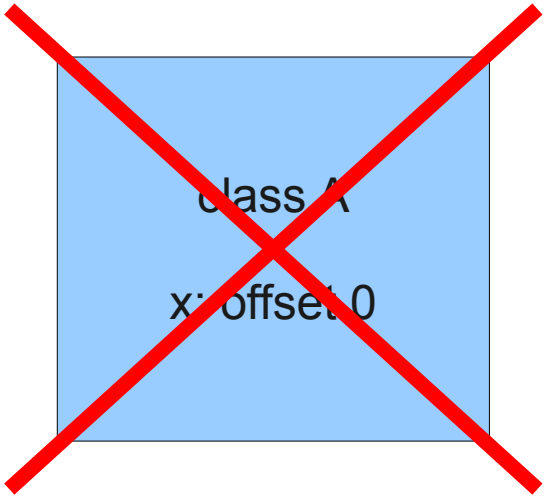
```
a = {x: 1};
```



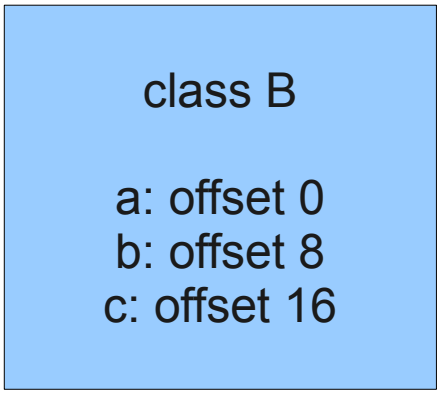

```
b = {a: 1, b: 0.5, c: null};
```



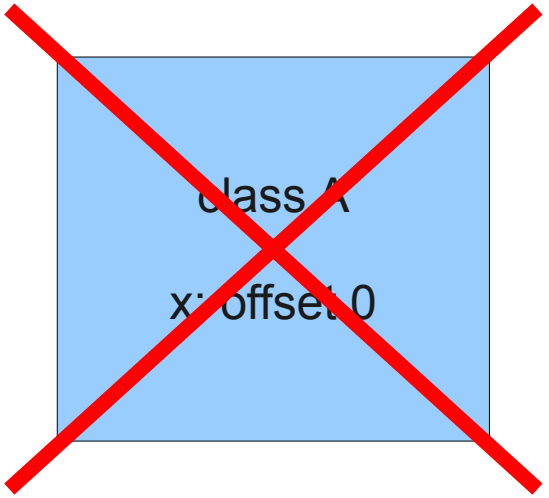
```
a = {x: 1};
a.y = “foo”;
```



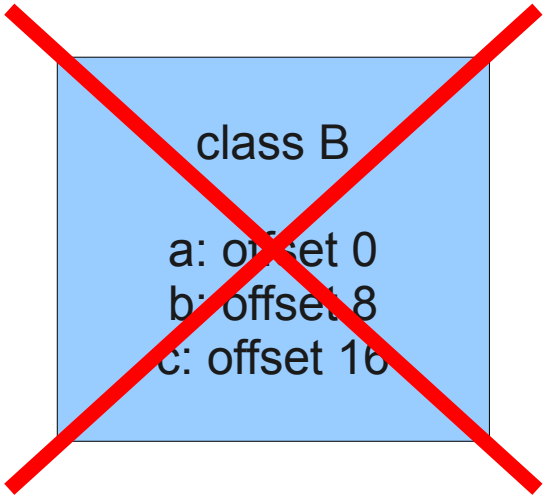

```
b = {a: 1, b: 0.5, c: null};
```



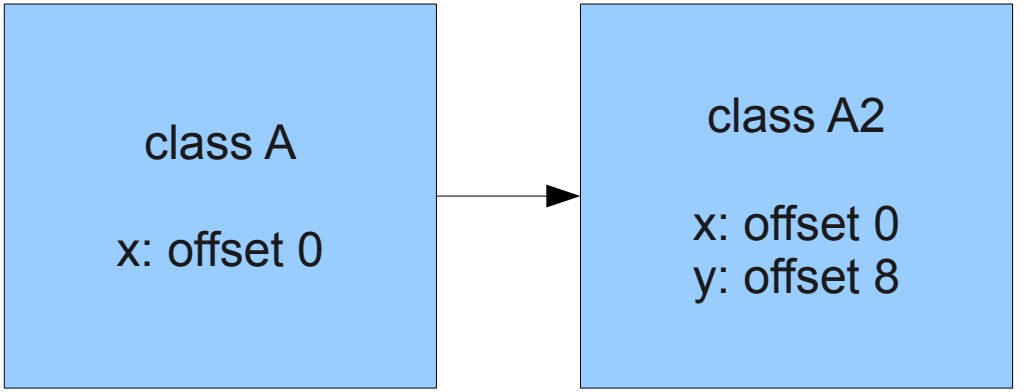
```
a = {x: 1};
a.y = “foo”;
```



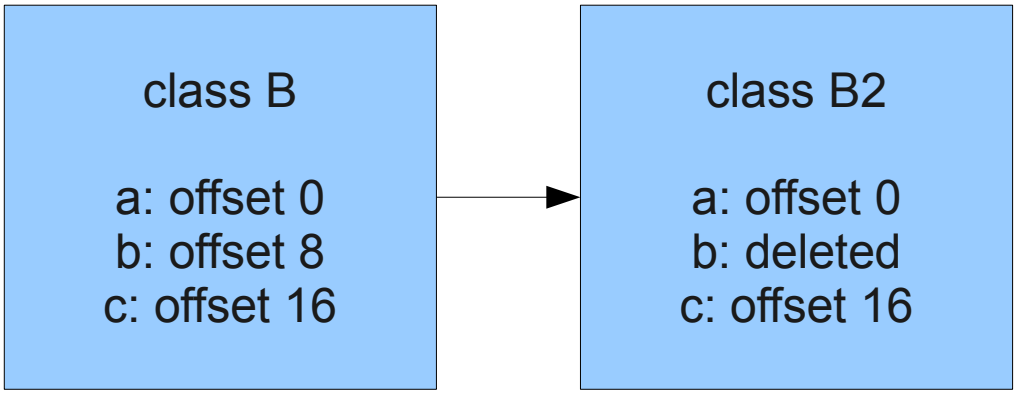

```
b = {a: 1, b: 0.5, c: null};
eval('delete b.b;');
```



```
a = {x: 1};
a.y = “foo”;
```




```
b = {a: 1, b: 0.5, c: null};
eval('delete b.b;');
```



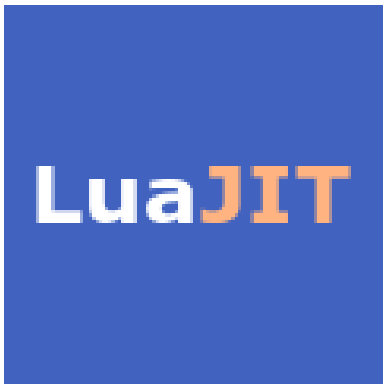

python
TM


Lua


Perl


php

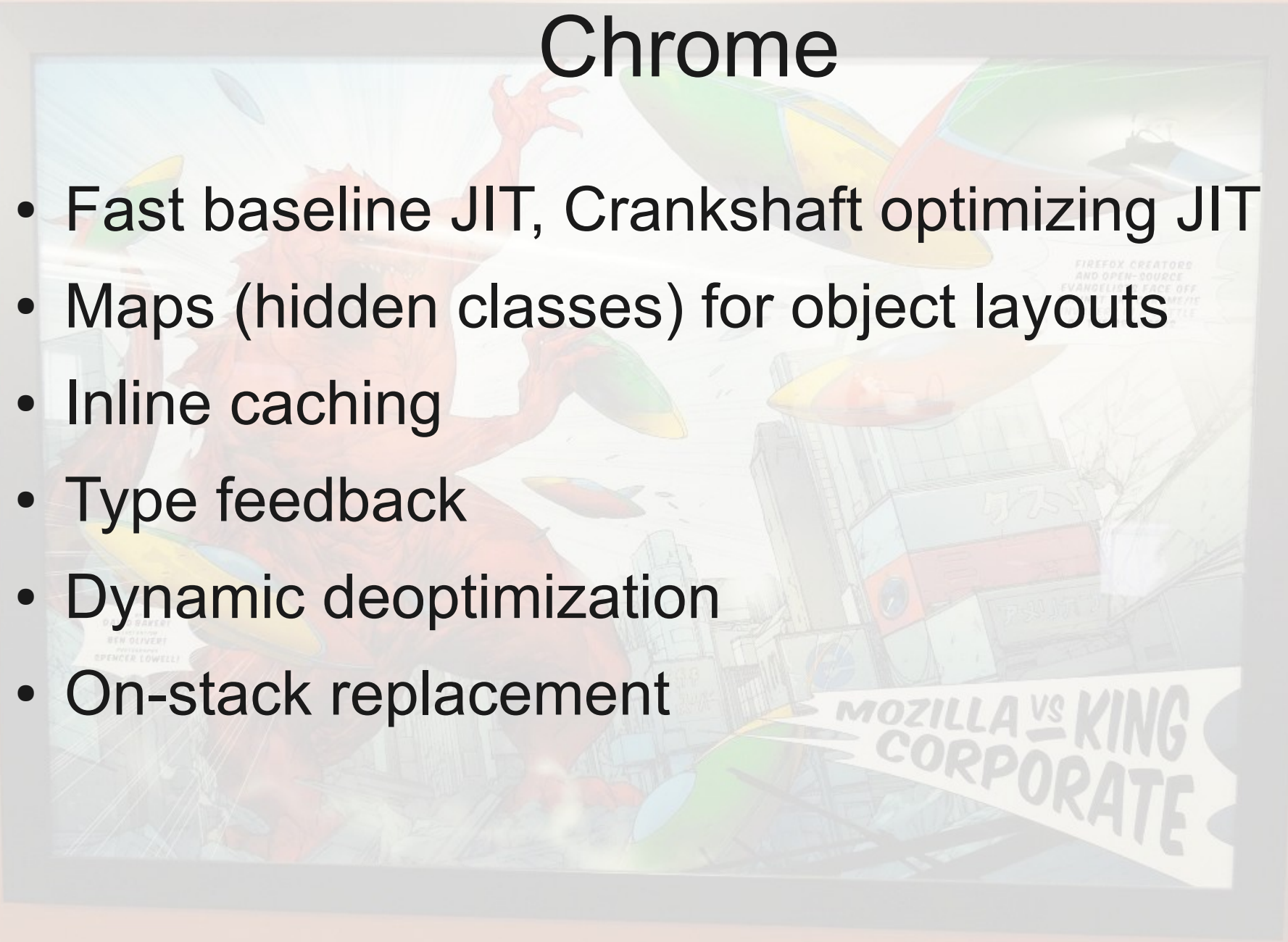

pypy


LuaJIT


FIREFOX CREATORS AND OPEN-SOURCE EVANGELISTS FACE OFF AGAINST THE CHROME/IE INVADERS IN A BATTLE OF THE BROWSERS...
BY DAVID BAKER!
ILLUSTRATION BEN OLIVER!
PHOTOGRAPHY SPENCER LOWELL!
MOZILLA VS KING CORPORATE

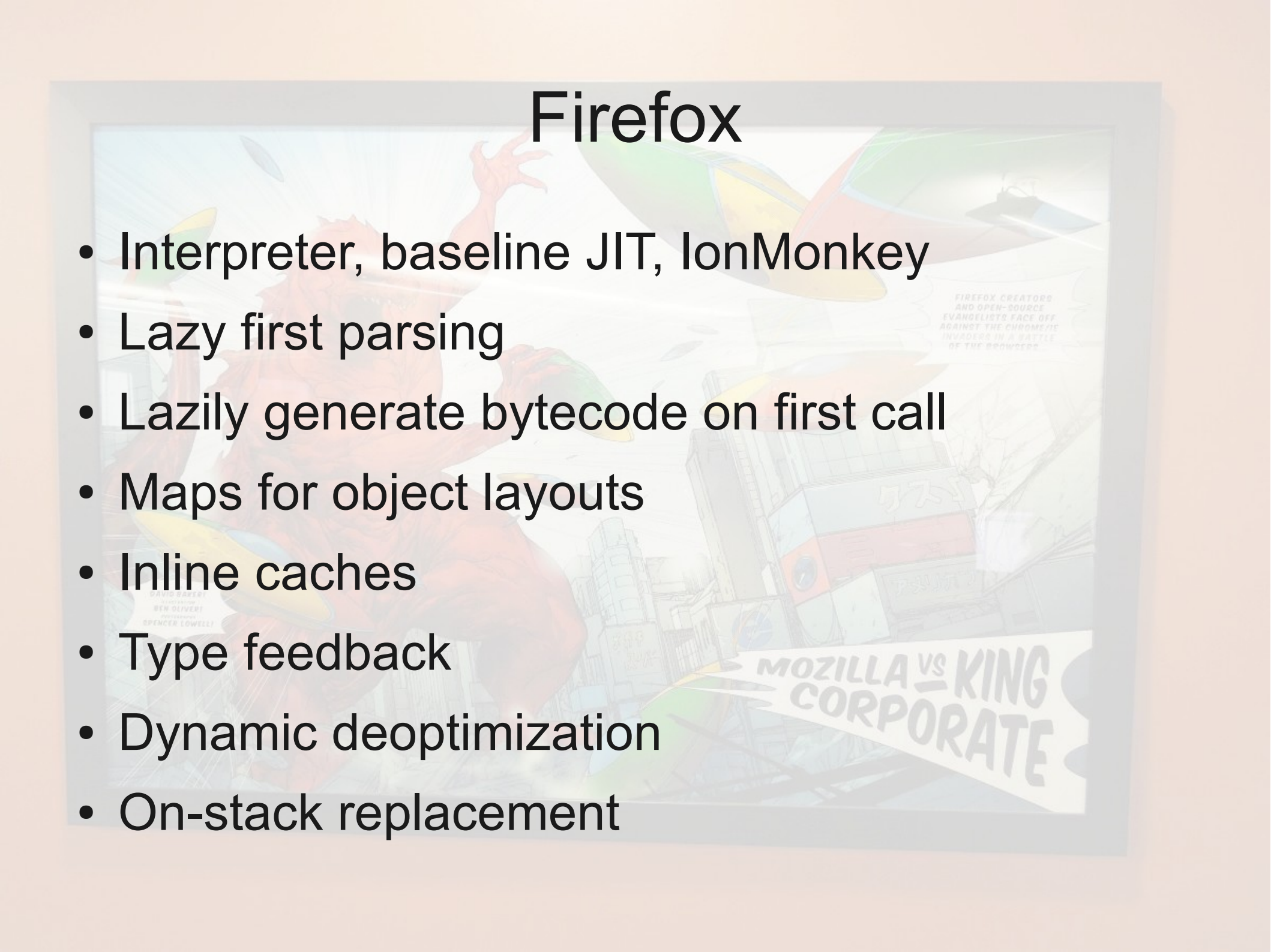
# Chrome

- Fast baseline JIT, Crankshaft optimizing JIT
- Maps (hidden classes) for object layouts
- Inline caching
- Type feedback
- Dynamic deoptimization
- On-stack replacement



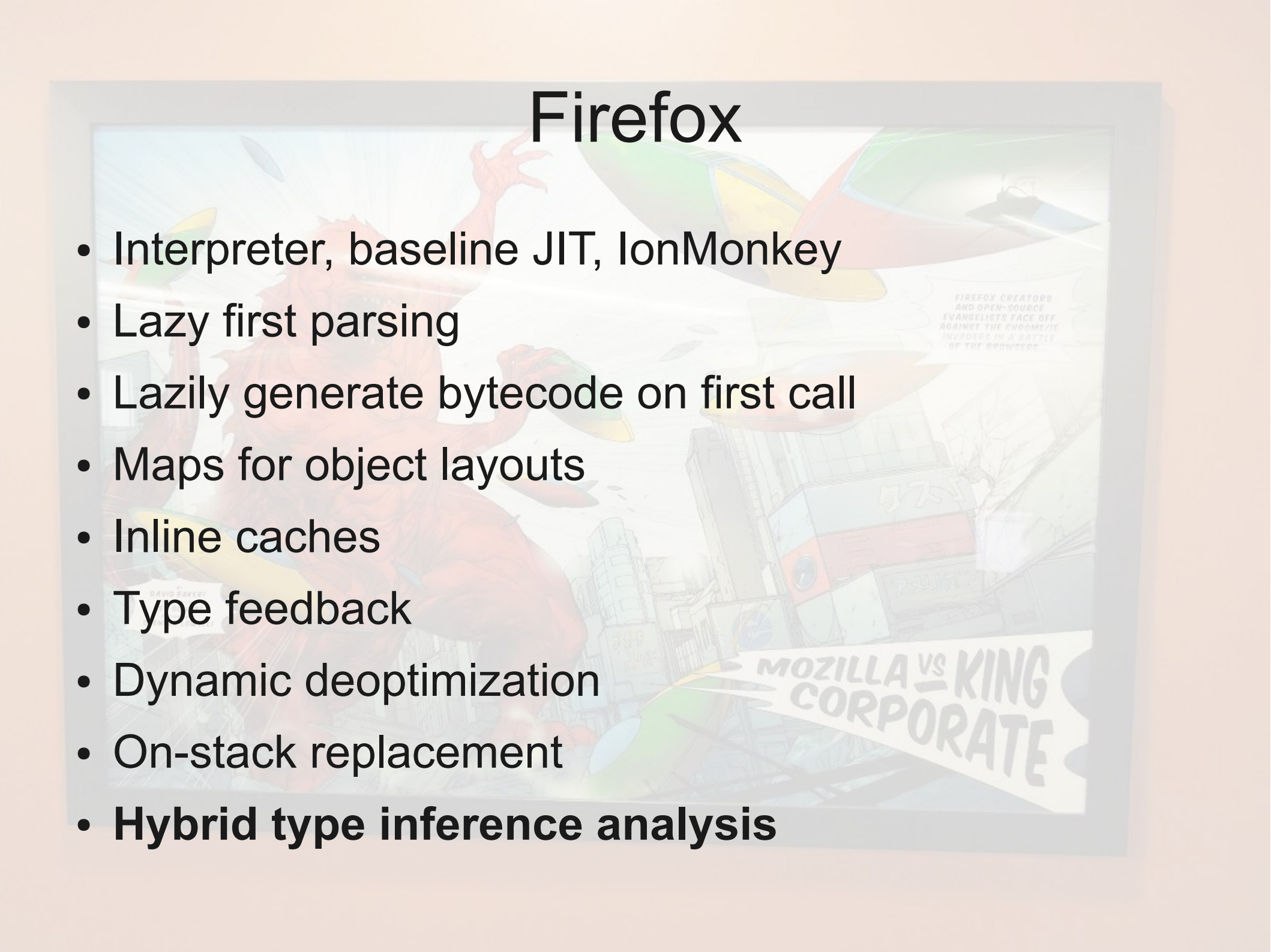
# Firefox

- Interpreter, baseline JIT, IonMonkey
- Lazy first parsing
- Lazily generate bytecode on first call
- Maps for object layouts
- Inline caches
- Type feedback
- Dynamic deoptimization
- On-stack replacement



# Firefox

- Interpreter, baseline JIT, IonMonkey
- Lazy first parsing
- Lazily generate bytecode on first call
- Maps for object layouts
- Inline caches
- Type feedback
- Dynamic deoptimization
- On-stack replacement
- **Hybrid type inference analysis**



# Fast and Precise Hybrid Type Inference for JavaScript

Brian Hackett  Shu-yu Guo *
Mozilla
{bhackett,shu}@mozilla.com

## Abstract

JavaScript performance is often bound by its dynamically typed nature. Compilers do not have access to static type information, making generation of efficient, type-specialized machine code difficult. We seek to solve this problem by inferring types. In this paper we present a hybrid type inference algorithm for JavaScript based on points-to analysis. Our algorithm is *fast*, in that it pays for itself in the optimizations it enables. Our algorithm is also *precise*, generating information that closely reflects the program's actual behavior even when analyzing polymorphic code, by augmenting static analysis with run-time type barriers.

We showcase an implementation for Mozilla Firefox's JavaScript engine, demonstrating both performance gains and viability. Through integration with the just-in-time (JIT) compiler in Firefox, we have improved performance on major benchmarks and JavaScript-heavy websites by up to 50%. Inference-enabled compilation is the default compilation mode as of Firefox 9.

**Categories and Subject Descriptors** D.3.4 [*Processors*]: Compilers, optimization

**Keywords** type inference, hybrid, just-in-time compilation

## 1. The Need for Hybrid Analysis

Consider the example JavaScript program in Figure 1. This pro-

```
1  function Box(v) {
2    this.p = v;
3  }
4
5  function use(a) {
6    var res = 0;
7    for (var i = 0; i < 1000; i++) {
8      var v = a[i].p;
9      res = res + v;
10   }
11   return res;
12 }
13
14 function main() {
15   var a = [];
16   for (var i = 0; i < 1000; i++)
17     a[i] = new Box(10);
18   use(a);
19 }
```

**Figure 1.** Motivating Example

values, using either a separate type tag for the value or a specialized marshaling format. This incurs a large runtime overhead on the

# Optimizing MATLAB through Just-In-Time Specialization *

Maxime Chevalier-Boisvert, Laurie Hendren, and Clark Verbrugge

School of Computer Science, McGill University, Montreal, QC, Canada
{mcheva,hendren,clump}@cs.mcgill.ca

**Abstract.** Scientists are increasingly using dynamic programming languages like MATLAB for prototyping and implementation. Effectively compiling MATLAB raises many challenges due to the dynamic and complex nature of MATLAB types. This paper presents a new JIT-based approach which specializes and optimizes functions on-the-fly based on the current types of function arguments.

A key component of our approach is a new type inference algorithm which uses the run-time argument types to infer further type and shape information, which in turn provides new optimization opportunities. These techniques are implemented in McVM, our open implementation of a MATLAB virtual machine. As this is the first paper reporting on McVM, a brief introduction to McVM is also given.

We have experimented with our implementation and compared it to several other MATLAB implementations, including the Mathworks proprietary system, McVM without specialization, the Octave open-source interpreter and the McFor static compiler. The results are quite encouraging and indicate that specialization is an effective optimization—McVM with specialization outperforms Octave by a large margin and also sometimes outperforms the Mathworks implementation.

## 1 Introduction

Dynamic languages require dynamic compilers

# Higgs

- JavaScript VM
  - Interpreter
  - JIT compiler for x86-64
  - Self-hosted runtime library
- Support for most of ECMAScript 5
  - No `with` statement, no getters/setters
  - Foreign Function Interface (FFI)
- Platform to try various ideas
  - Relatively small codebase (< 50 KLOC)

# Basic Block Versioning

- Similarities with tracing, procedure cloning
- As you compile code, accumulate facts
- Compile multiple versions of code
- Specialize based on accumulated facts:
  - Low-level type information (type tags)
  - Register allocation state
  - Object types, global variable types

```
var v = 4294967296;
for (var i = 0; i < 600000; i++)
    v = v & i;

// From the SunSpider bitwise-and benchmark
```

```
var v = 4294967296;
for (var i = 0; less_than(i,600000); i = add(i,1))
    v = bitwise_and(v,i);
```

```
var v = 4294967296;
for (var i = 0; less_than(i,600000); i = add(i,1))
    v = bitwise_and(v,i);

function bitwise_and(x,y)
{
   if (is_int32(x) && is_int32(y))
      return bitwise_and_int32(x,y); // Fast path

   return bitwise_and(toInt32(x), toInt32(y));
}
```

```
var v = 4294967296;
for (var i = 0; less_than(i,600000); i = add(i,1))
    v = bitwise_and(v,i);

function bitwise_and(x,y)
{
   if (is_int32(x) && is_int32(y))
      return bitwise_and_int32(x,y); // Fast path

   return bitwise_and(toInt32(x), toInt32(y));
}

function add(x,y)
{
   if (is_int32(x) && is_int32(y))
   {
      var r = add_int32(x,y); // Fast path

      if (cpu_overflow_flag)
         return add_double(toDouble(x), toDouble(y));
   }

   return add_general(x,y);
}
```

```
var v = 4294967296;
var i = 0;
for (;;)
{
   // if (i >= 600000) break;
   if (is_int32(i) && is_int32(600000))
      if (greater_eq_int32(i, 600000)) break;
   else
      if (greater_eq_general(i, 6000000) break;

   // v = v & i
   if (is_int32(v) && is_int32(i))
      v = bitwise_and_int32(v,i);
   else
      v = bitwise_and(toInt32(v), toInt32(i));

   // i = i + 1
   if (is_int32(i) && is_int32(1))
   {
      i = add_int32(i,1);
      if (cpu_overflow_flag)
         i = add_double(toDouble(i), toDouble(1));
   }
   else
      i = add_general(i,1);
}
```

```
var v = 4294967296;
var i = 0;
for (;;)
{
   // if (i >= 600000) break;
   if (is_int32(i) && is_int32(600000))
      if (greater_eq_int32(i, 600000)) break;
   else
      if (greater_eq_general(i, 6000000) break;

   // v = v & i
   if (is_int32(v) && is_int32(i))
      v = bitwise_and_int32(v,i);
   else
      v = bitwise_and(toInt32(v), toInt32(i));

   // i = i + 1
   if (is_int32(i) && is_int32(1))
   {
      i = add_int32(i,1);
      if (cpu_overflow_flag)
         i = add_double(toDouble(i), toDouble(1));
   }
   else
      i = add_general(i,1);
}
```

```
var v = 4294967296;
var i = 0;
for (;;)
{
   // if (i >= 600000) break;
   if (is_int32(i))
      if (greater_eq_int32(i, 600000)) break;
   else
      if (greater_eq_general(i, 6000000) break;

   // v = v & i
   if (is_int32(v) && is_int32(i))
      v = bitwise_and_int32(v,i);
   else
      v = bitwise_and(toInt32(v), toInt32(i));

   // i = i + 1
   if (is_int32(i))
   {
      i = add_int32(i,1);
      if (cpu_overflow_flag)
         i = add_double(toDouble(i), toDouble(1));
   }
   else
      i = add_general(i,1);
}
```

```
var v = 4294967296;
var i = 0; // when we enter the loop, i is int32
for (;;)
{
   // if (i >= 600000) break;
   if (is_int32(i))
      if (greater_eq_int32(i, 600000)) break;
   else
      if (greater_eq_general(i, 6000000) break;

   // v = v & i
   if (is_int32(v) && is_int32(i))
      v = bitwise_and_int32(v,i);
   else
      v = bitwise_and(toInt32(v), toInt32(i));

   // i = i + 1
   if (is_int32(i))
   {
      i = add_int32(i,1);
      if (cpu_overflow_flag)
         i = add_double(toDouble(i), toDouble(1));
   }
   else
      i = add_general(i,1);
}
```

```
var v = 4294967296;
var i = 0;
for (;;)
{
   // if (i >= 600000) break;
   if (greater_eq_int32(i, 600000)) break;

   // v = v & i
   if (is_int32(v) && is_int32(i))
      v = bitwise_and_int32(v,i);
   else
      v = bitwise_and(toInt32(v), toInt32(i));

   // i = i + 1
   if (is_int32(i))
   {
      i = add_int32(i,1);
      if (cpu_overflow_flag)
         i = add_double(toDouble(i), toDouble(1));
   }
   else
      i = add_general(i,1);
}
```

```
var v = 4294967296;
var i = 0;
for (;;)
{
   // if (i >= 600000) break;
   if (greater_eq_int32(i, 600000)) break;

   // v = v & i
   if (is_int32(v) && is_int32(i))
      v = bitwise_and_int32(v,i);
   else
      v = bitwise_and(toInt32(v), toInt32(i));

   // i = i + 1
   if (is_int32(i))
   {
      i = add_int32(i,1);
      if (cpu_overflow_flag)
         i = add_double(toDouble(i), toDouble(1));
   }
   else
      i = add_general(i,1);
}
```

```
var v = 4294967296;
var i = 0;
for (;;)
{
    // if (i >= 600000) break;
    if (greater_eq_int32(i, 600000)) break;

    // v = v & i
    v = bitwise_and_int32(v,i);

    // i = i + 1
    if (is_int32(i))
    {
        i = add_int32(i,1);
        if (cpu_overflow_flag)
            i = add_double(toDouble(i), toDouble(1));
    }
    else
        i = add_general(i,1);
}
```

```
var v = 4294967296;
var i = 0;
for (;;)
{
   // if (i >= 600000) break;
   if (greater_eq_int32(i, 600000)) break;

   // v = v & i
   v = bitwise_and_int32(v,i);

   // i = i + 1
   if (is_int32(i))
   {
      i = add_int32(i,1);
      if (cpu_overflow_flag)
         i = add_double(toDouble(i), toDouble(1));
   }
   else
      i = add_general(i,1);
}
```

```
var v = 4294967296;
var i = 0;
for (;;)
{
   // if (i >= 600000) break;
   if (greater_eq_int32(i, 600000)) break;

   // v = v & i
   v = bitwise_and_int32(v,i);

   // i = i + 1
   i = add_int32(i,1);
   if (cpu_overflow_flag)
      i = add_double(toDouble(i), toDouble(1));
}
```

```
var v = 4294967296;
var i = 0;
for (;;)
{
    // if (i >= 600000) break;
    if (greater_eq_int32(i, 600000)) break;

    // v = v & i
    v = bitwise_and_int32(v,i);

    // i = i + 1
    i = add_int32(i,1);
    if (cpu_overflow_flag)
       i = add_double(toDouble(i), toDouble(1));
}
```

```
var v = 4294967296;
var i = 0;
for (;;)
{
   // if (i >= 600000) break;
   if (greater_eq_int32(i, 600000)) break;

   // v = v & i
   v = bitwise_and_int32(v,i);

   // i = i + 1
   i = add_int32(i,1);
   if (cpu_overflow_flag)
   {
      i = add_double(toDouble(i), toDouble(1));
      NEW_LOOP_VERSION = gen_new_version();
      goto NEW_LOOP_VERSION;
   }

   // If we make it here, i is still int32
}
```

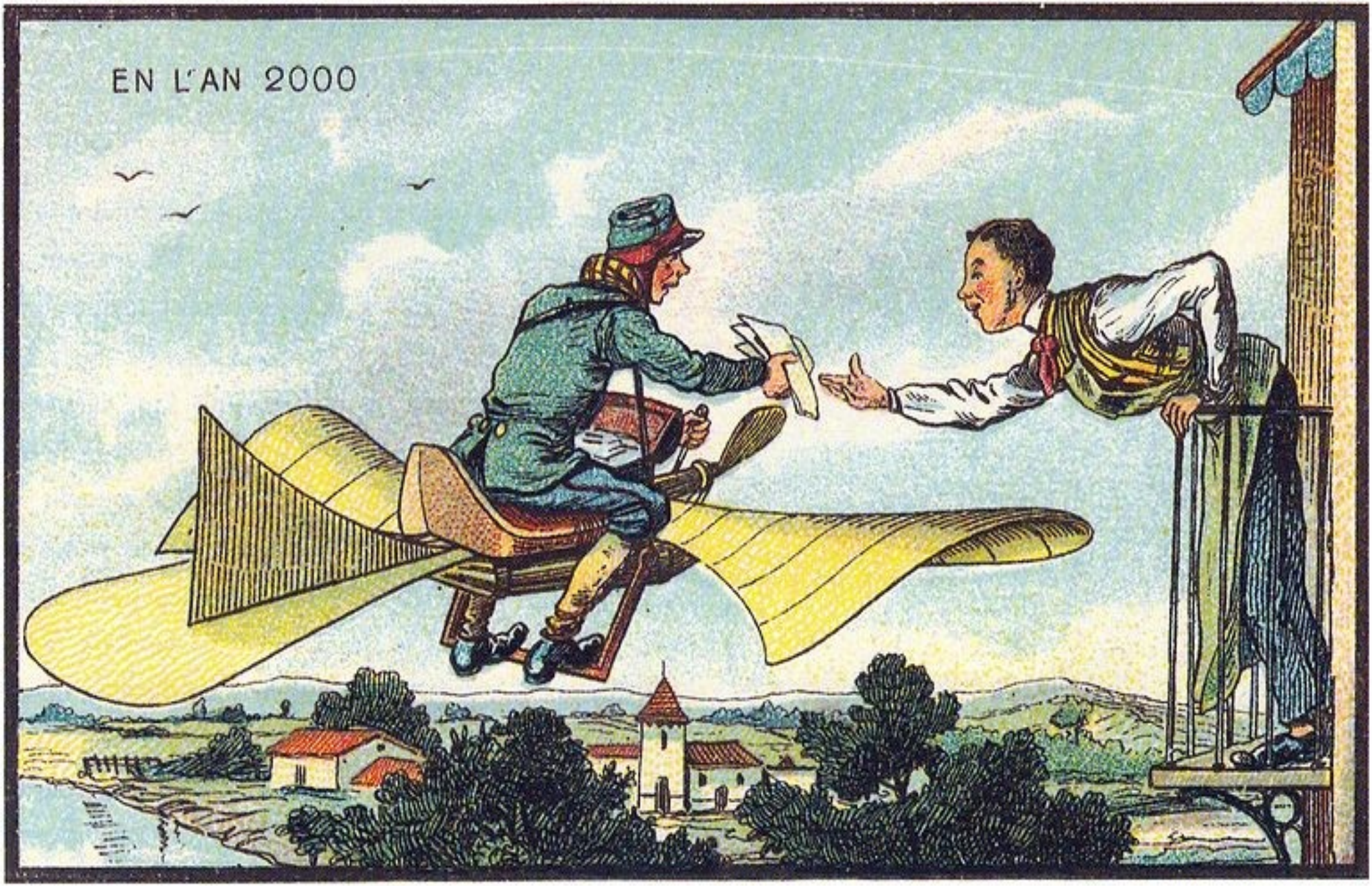
# A “Multi-World” Approach

- Traditional type analysis
  - Fixed-point on types
  - At each program point, agree with all inputs
  - Pessimistic, conservative answer
- Basic block versioning
  - Multiple solutions possible
  - Don't necessarily have to sacrifice
  - Fixed-point on versioning of blocks

# Research Questions

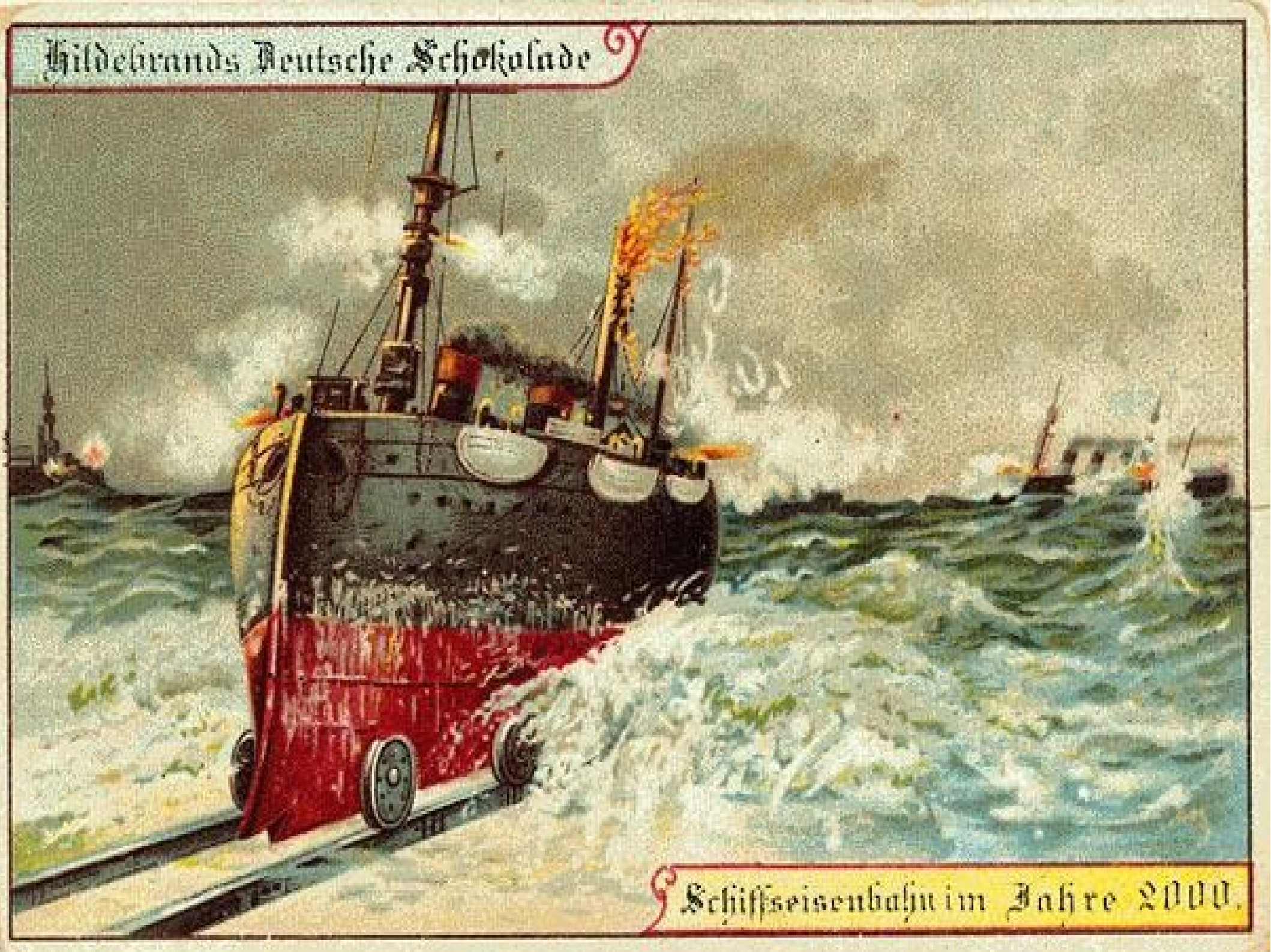
- How much code blowup can we expect?
- What can we do to reduce code blowup?
- What performance gains can we expect?
- What kind of info should we version with?
  - Constant propagation
  - Granularity of type info used
  - How much is too much?
- What's the effect on compilation time?


What does the future hold?




The Rural Postman

Hildebrands Deutsche Schokolade



Schiffseisenbahn im Jahre 2000.

# What does the future hold?

# What does the future hold?

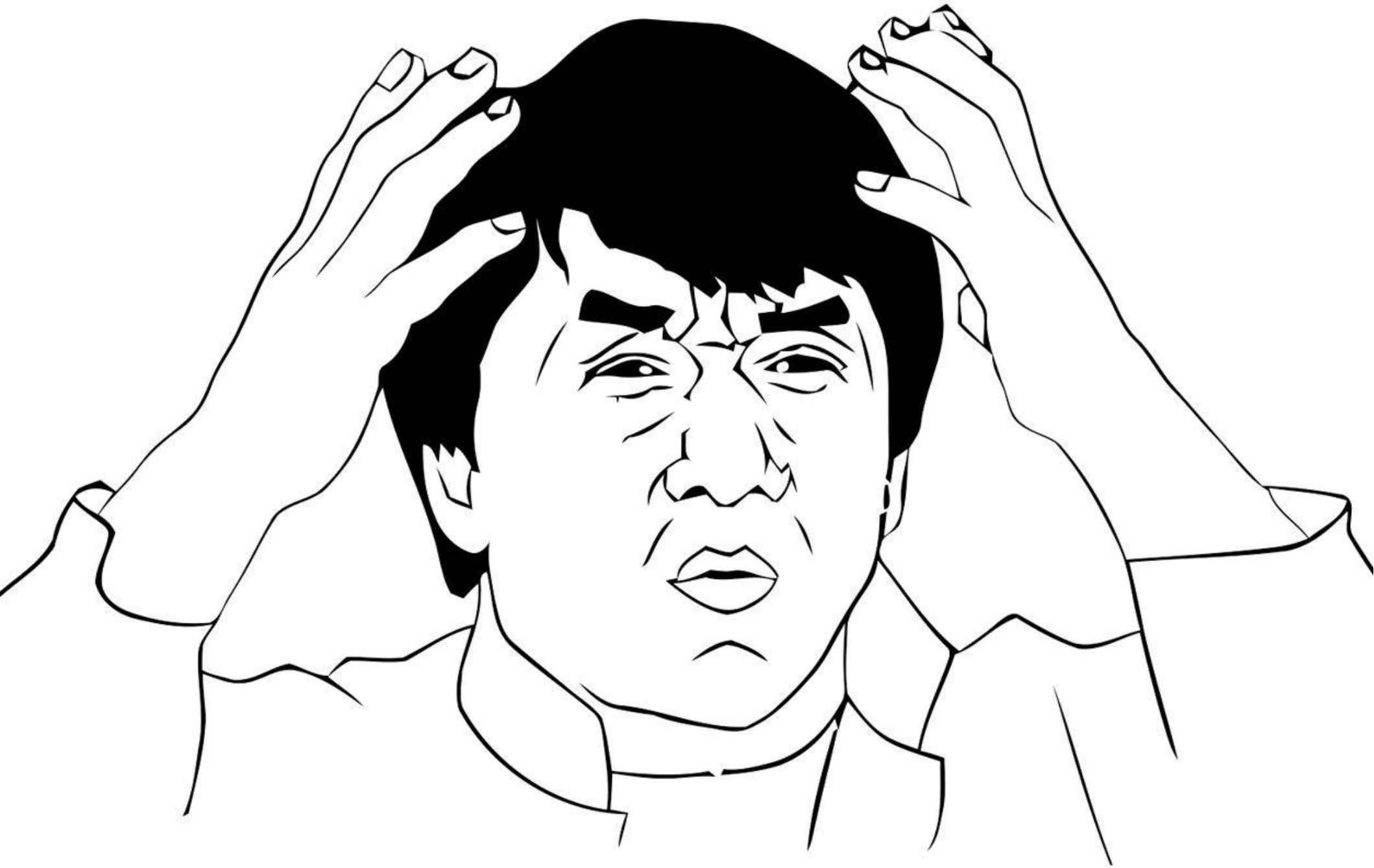
- More advanced CPUs

# What does the future hold?

- More advanced CPUs
- Smarter compilers

# What does the future hold?

- More advanced CPUs
- Smarter compilers
- Better language design

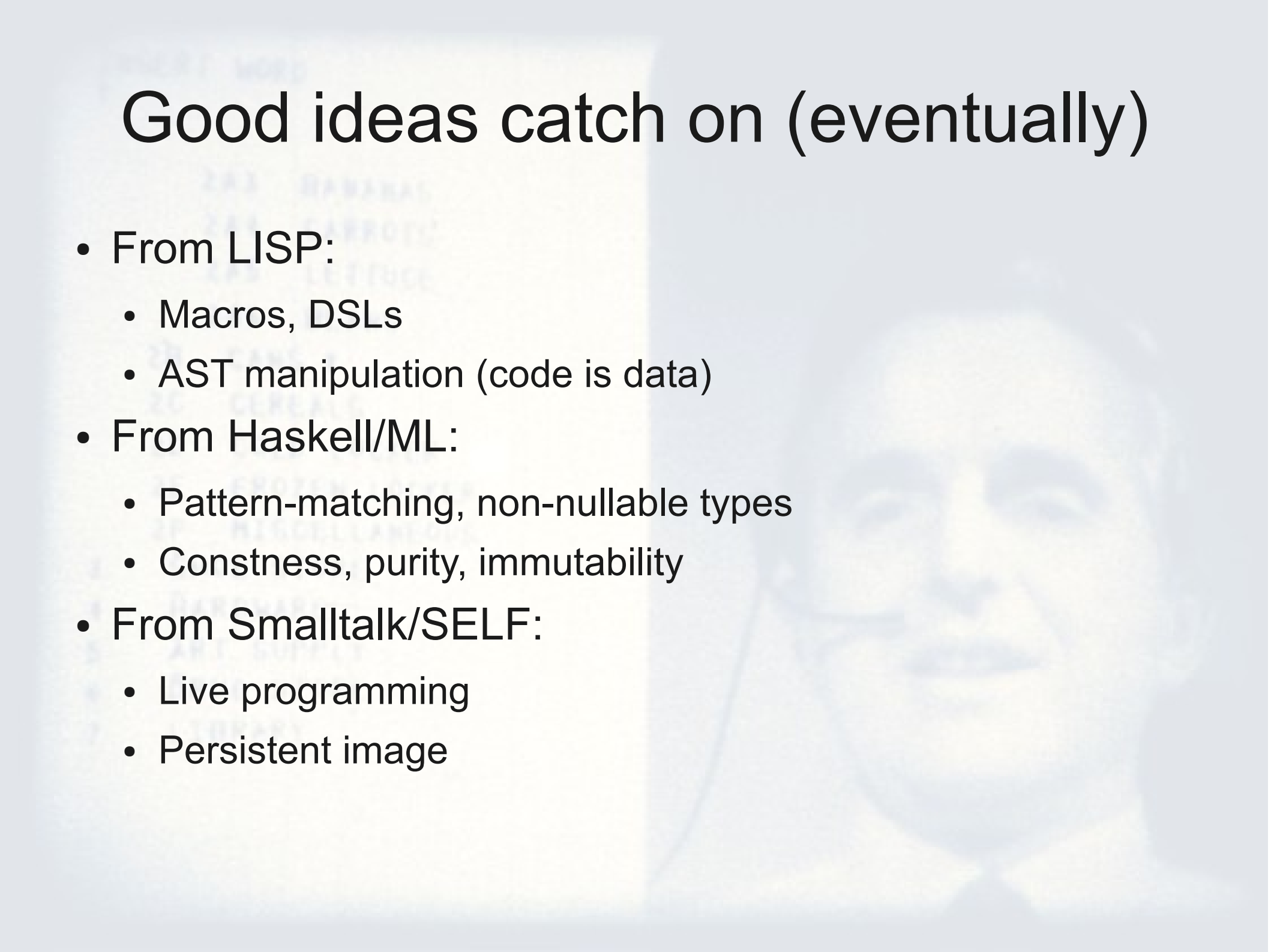


# Good ideas that caught on

- From LISP:
  - Dynamic typing
  - Closures
  - The REPL
- From Haskell/ML:
  - Type inference
- From Smalltalk/SELF:
  - Prototypal inheritance
  - Duck-typing

# Good ideas catch on (eventually)

- From LISP:
  - Macros, DSLs
  - AST manipulation (code is data)
- From Haskell/ML:
  - Pattern-matching, non-nullable types
  - Constness, purity, immutability
- From Smalltalk/SELF:
  - Live programming
  - Persistent image



github.com/maximecb/Higgs

maximechevalierb@gmail.com

pointersgonewild.wordpress.com

Love2Code on twitter