A Concise and OPINIONATED *History* of Virtual Machines

Mario Wolczko
Architect, Oracle Labs

The following is provided for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. Oracle reserves the right to alter its development plans and practices at any time, and the development, release, and timing of any features or functionality described in connection with any Oracle product or service remains at the sole discretion of Oracle. Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.

Overview

Goals:

1. Introduce the most important ideas and systems in VM design and implementation from a historical perspective

2. Supply background for this afternoon's lab, taught by Chris Seaton.

Overview

Audience: I am assuming the typical user’s understanding of VM internals (i.e., not much).

Limitations: can’t be detailed, or anywhere near complete. I’m deliberately omitting:

- System VMs and binary translation
- Garbage collection techniques
- interpretation and compiler history (outside of language VMs)

Lab

Led by Chris Seaton, you will:

- Download a development environment for *GraalVM*
- Inspect and modify an implementation of a simple language
- Examine the outputs of the various components of the system

CS294-113
A Shameless Plug

- In 2015 I was invited to teach a graduate course on VMs at UC Berkeley.
- The result is CS294-113: Virtual Machines and Managed Runtimes.
- Available on the web (slides, video, exercises) at www.wolczko.com/CS294.
- ~30 hours of video, over 1200 slides. Estimated 200+ hours to complete coursework.
- Guest appearances: Deutsch & Schiffman, Ungar, Click, Bak, Bolz, Würthinger, Van De Vanter
- Taken and completed by >20 UCB students.

Who am I?

- Architect at Oracle, formerly a Distinguished Engineer at Sun. Before that I did a PhD and postdoc at Manchester.

- I’ve been mostly in research, with occasional forays into product development.

- I started my career in VMs at the beginning of a golden era (1983), and have seen many developments up close (esp. Java, at Sun).

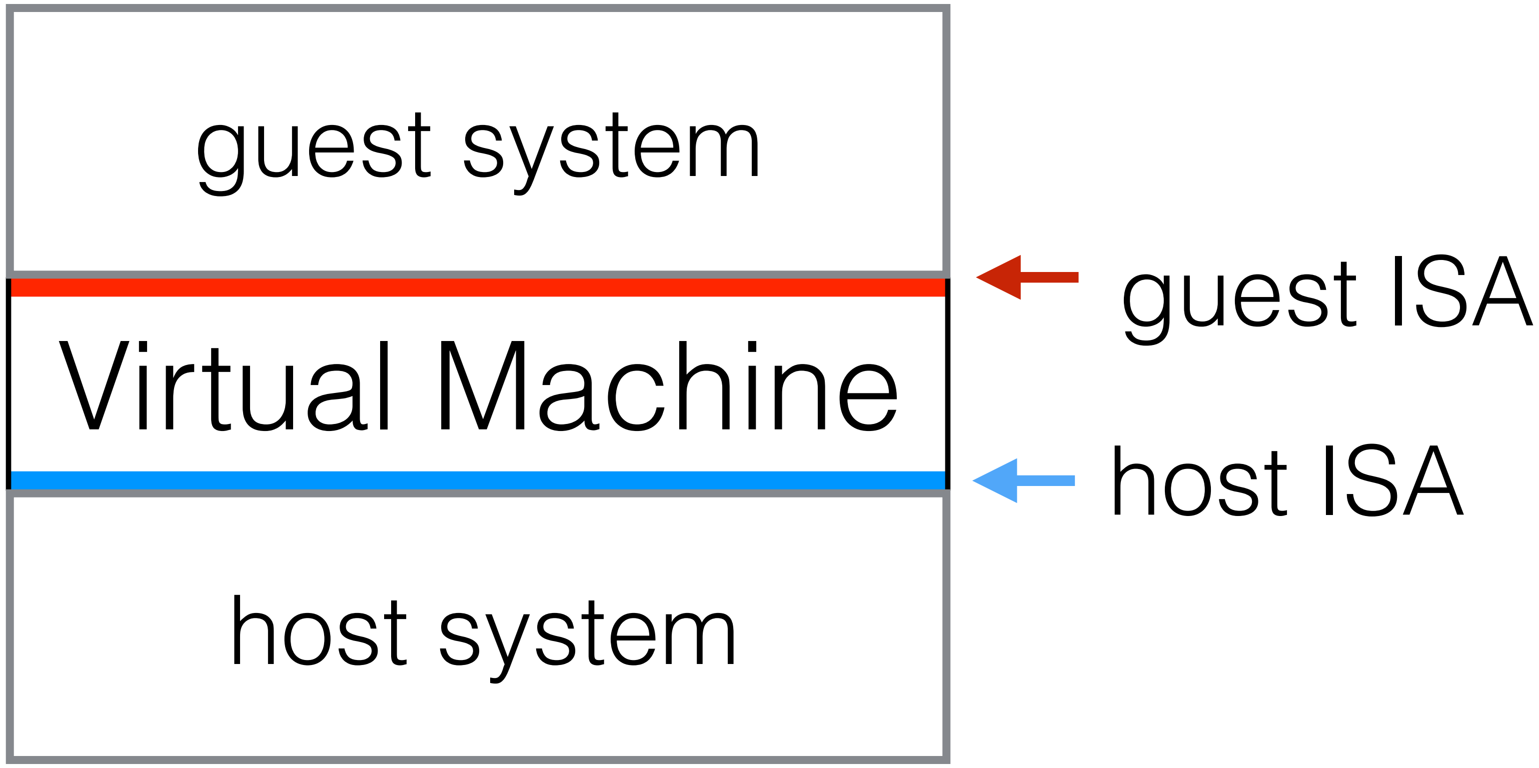
- I’ve been fortunate to work with and talk to many VM pioneers.

What do I mean by “history”?

- I am not a professional historian
- I don’t even play one on TV
- I am more a participant and witness than a chronicler
- Hence, this is a subjective history
- It is laced with my opinions — some of which ~~may be~~ are wrong.

1. Introduction to VMs

Generic VM architecture

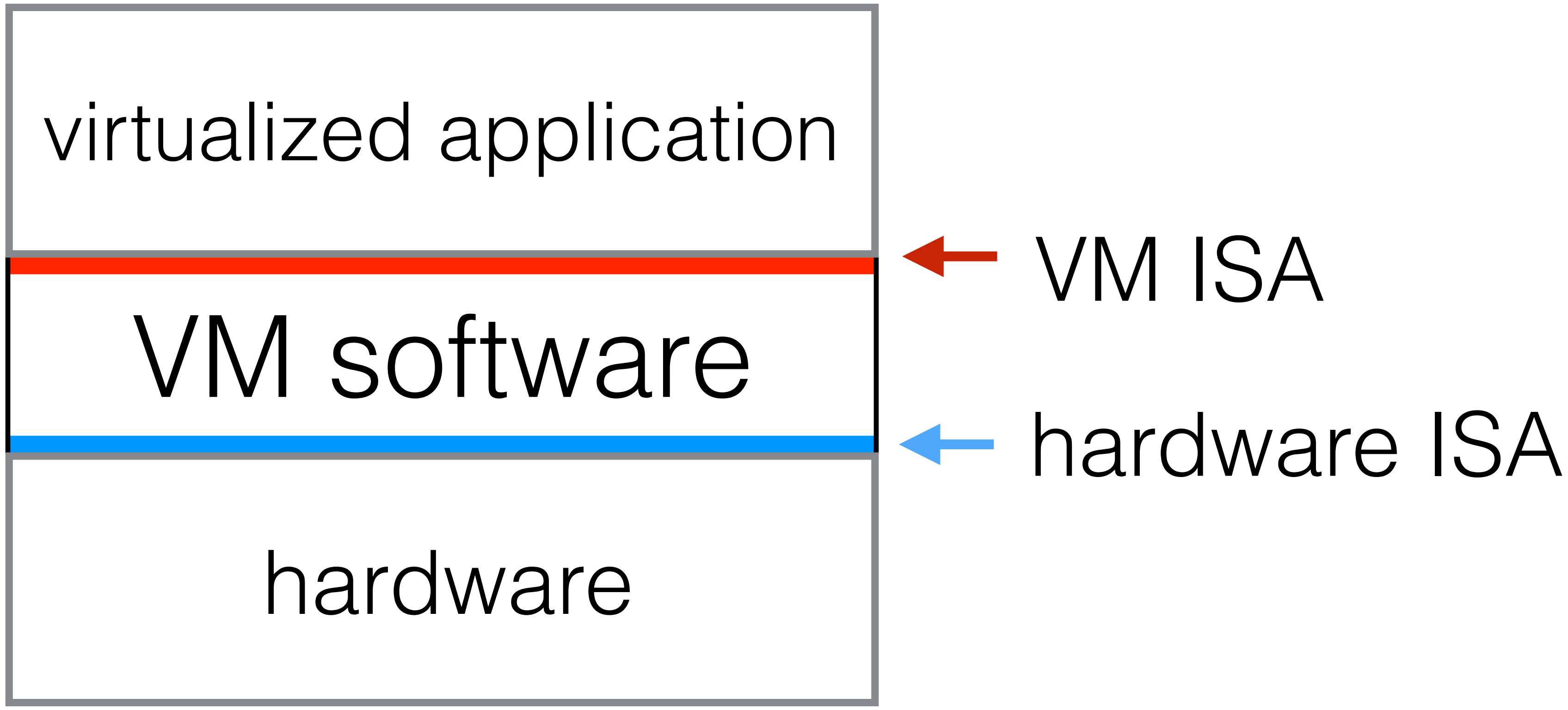


ISA = Instruction Set Architecture

What is a Virtual Machine?

- A software implementation of a machine architecture
  - Software needs hardware to run, so hardware is implied too
- Two machine architectures are involved: *guest* and *host*
- The guest may be defined only by software or be an emulation of a real machine (i.e., also available as a hardware implementation)
- The host is usually hardware, but need not be (e.g., a language VM written in Java running on the JVM)

A typical VM


virtualized application
VM software
hardware
VM ISA
hardware ISA

Process and System VMs

Process and System VMs

- A **Process VM** implements an **ABI** (Application Binary Interface: the combination of a user-level ISA and an OS system call interface)

Process and System VMs

- A **Process VM** implements an **ABI** (Application Binary Interface: the combination of a user-level ISA and an OS system call interface)

- A **System VM** implements *both* the hardware user and system ISA (i.e., including privileged instructions)

Process and System VMs

- A **Process VM** implements an **ABI** (Application Binary Interface: the combination of a user-level ISA and an OS system call interface)

- A **System VM** implements *both* the hardware user and system ISA (i.e., including privileged instructions)

Will not consider these further here — see the full course for more info.

Language VMs

- A *language-specific* process VM
- The VM presents an OS-like interface to applications in the chosen language, as well as an ISA designed specifically for the semantics of the chosen language.

2. Interpretation
(and compilation)

What is an interpreter?

An interpreter for a source language *L* is a mechanism for the direct execution of all programs from *L*, which executes each element of the source program in turn without reference to other elements

Some consequences

- Performance is typically **uniform** and **predictable** (e.g., every execution of the same node is the same and has the same performance, modulo micro-architectural effects in the host machine [caching, branch prediction, etc.])

- Typically **slow**, as there is much overhead in deciding how to interpret a specific language element, and no scope for optimization across time or program space

In contrast: compilation

- A compiler transforms a program in a source language S to an equivalent program in a target language T (S≠T).
- It does not execute the source program at all (cf. interpretation)

Interpretation is usually preceded by some kind of compilation

- It is rare that the source program of any non-trivial language is executed directly by an interpreter; usually it is transformed by a parser or compiler into some intermediate representation (IR)

- The IR removes lexical noise such as comments, white space and other formatting

    - Don't want to penalize commentary and formatting by increasing execution time

- Lexemes are either condensed into "atoms" (identifiers, constants) or abstracted/combined into operations (keywords, operators)

- Elements are reordered into execution order (e.g., operators in an expression)

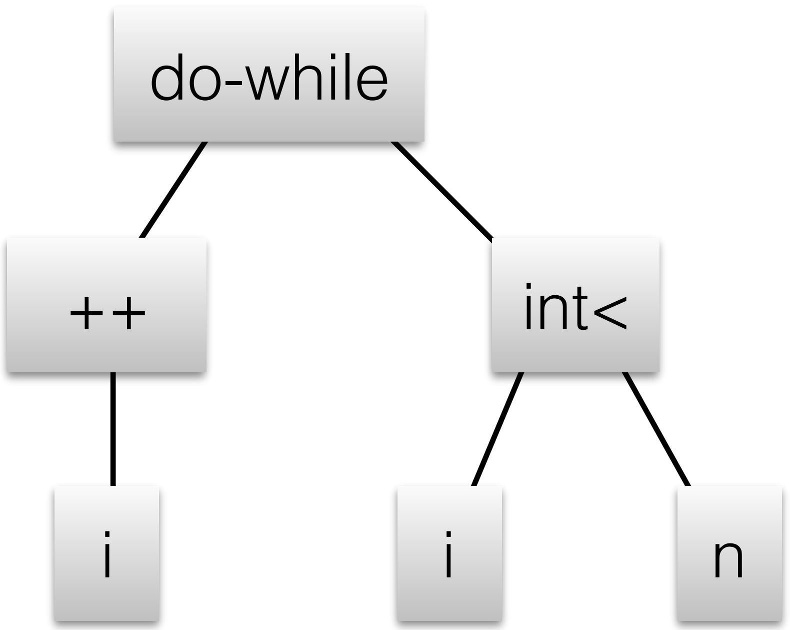
The interpreter illusion

- One common goal of an advanced VM implementation is to preserve the illusion of interpretation while maximizing performance (e.g., by compiling on-the-fly).
- Can be challenging! For example, compilers commonly reorder actions.
- The trick is to carefully define/discover what is observable, and what is not
    - Or: “cheat, but don’t get caught”

AST interpreters for high-level languages

- AST = Abstract Syntax Tree
- The tree produced by a parser of a high-level language compiler

```
do {
  i++;
} while (i < n);
```



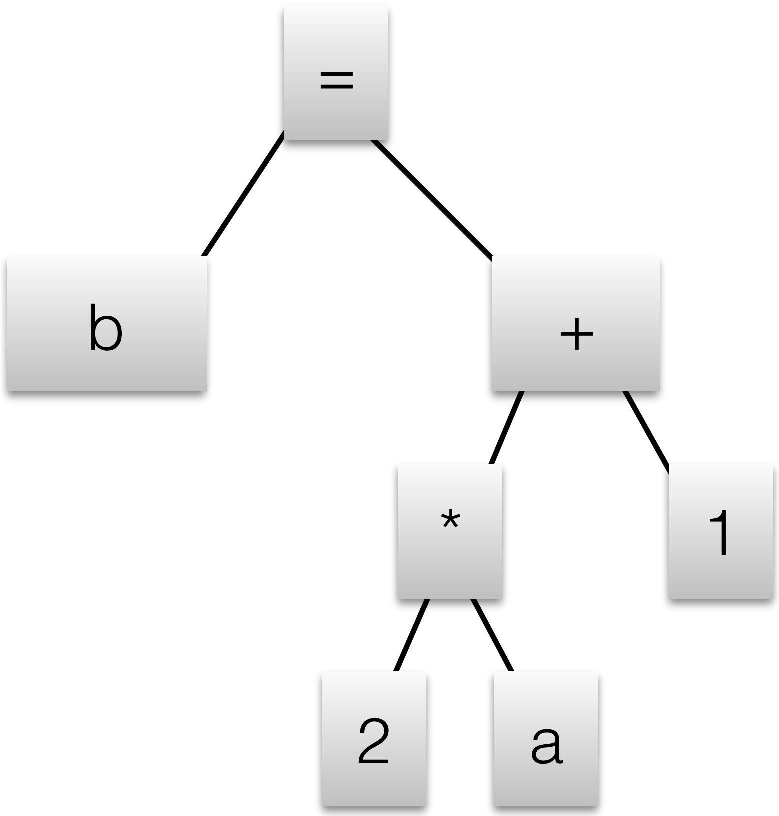
Example: interpreting a simple expression language

- Language elements: variables holding an integer; integer constants; expressions involving simple arithmetic; assignments

```
3+4
b=2*a+1
x*x+x+5
```

example input

b=2*a+1



Implementation using objects

- ASTs and interpreters are a natural fit for object-oriented programming.

- Each kind of node is a class

  - Use inheritance for better factoring

- Use method dispatch for evaluation

In Java…

```
class ASTNode {
 …stuff common to all nodes…}

abstract class ExprNode extends ASTNode
{
  abstract int eval();
}
```

```
// all trivial constructors elided

class ConstNode extends ExprNode {
  final int val;

  int eval() { return val; }
  …
}
```

```
class AddNode extends ExprNode {
  ExprNode left, right;
  int eval() {
    return left.eval()+right.eval(); }
  …
}
```

Other operator nodes look almost identical.

```
class VarNode extends ExprNode {
  int val;
  void set(int val) { this.val = val; }
  int eval() { return this.val; }
  …
}
```

```
class AssignNode extends ExprNode {
  VarNode var;
  ExprNode rhs;
  int eval() {
    int rhsVal=rhs.eval();
    var.set(rhsVal);
    return rhsVal; }
  …
}
```

Adding statements

```
abstract class StmtNode extends ASTNode {
  abstract void eval();
}

class SimpleStmtNode extends StmtNode {
  ExprNode e;
  void eval() { int ignored = e.eval(); }
}

class SeqNode extends StmtNode {
  ArrayList<StmtNode> seq;
  void eval() {
   for (StmtNode s : seq) { s.eval(); }
  }
}
```

Control flow is easy… sometimes

…when the semantics are local and equivalent in the implementation language. Example: consider interpreting a do-while expression.

```
class DoWhileNode extends StmtNode {
  StmtNode body; ExprNode cond;
  void eval() {
    do body.eval() while (cond.eval() != 0);
  }
}
```

But sometimes not so easy…

- Example: **`break`** from within a loop
- Why doesn’t this work?

```
class BreakNode extends StatementNode
{
  void eval() { break; }
}
```

Solutions to control flow problems

- Add a break-out path for each possible enclosing node type
  - Messy when you need to return a value *and* a break-out indication if the implementation language can't return a pair
- Use host language exceptions

Break, using exceptions

```
StmtNode: abstract void eval() throws
BreakException

class BreakNode … {
  void eval() throws BreakException {
    throw new BreakException();
  }
}
```

Break, using exceptions

```
class DoWhileNode … {
  StmtNode body;
  ExprNode cond;

  void eval() throws BreakException {
    try {
      do {
        body.eval();
      } while (cond.eval() != 0);
    catch (BreakException b) {};
  }
```

Interpreter performance considerations

Let’s look at what it takes to interpret b=2*a+1

```
eval( ⟦b=2*a+1⟧ )
  eval( ⟦2*a+1⟧ )
    eval( ⟦2*a⟧ )
      eval( ⟦2⟧ ) → $i$
      eval( ⟦a⟧ ) → $j$
      $i * j \rightarrow k$
    eval( ⟦1⟧ ) → $l$
    $k + l \rightarrow m$
  assign $m$ to ⟦b⟧
```

- Every eval() is a virtual call and return
  - How predictable is the flow?
- Max stack depth is 4 frames
- How many loads just to walk the tree?

AST interpretation: conclusions

- **Slow (typically 100—1000x slower than best possible)**
- ASTs are big
- But, **easy to write and reason about**; portable
- Source code as the distribution medium (cf. compilation) — lots of pros and cons (some of which are non-technical)
    - See [Franz ’94] for another variation on the theme
- Tightly coupled to the source language
- Languages change faster than instruction sets…which leads us to: **bytecodes**.
- We’ll revisit AST interpretation near the end, to see how it can be made fast.

3. Language VMs

Part 1, 1966—circa 2000

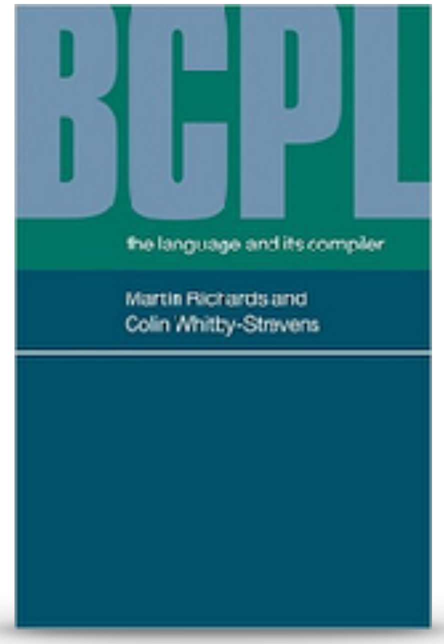
What is a Language VM?

- A language-specific Process VM
  - The VM presents an OS-like interface to applications as well as an ISA
- Often created together with, or during the evolution, of the associated language.
- Typically embodies language-specific concepts and semantics.
  - A relatively small jump from language semantics to VM interface.

Timeline

• BCPL	1966
• Pascal	1972–1977
• Smalltalk	1976–1984
• Self	1987–1994
• Java	1997—
• JavaScript	2008—
• Truffle/Graal	2014—

BCPL



- Basic CPL (Combined Programming Language)
  - CPL was a broad-spectrum language conceived by Christopher Strachey at Cambridge and others in the early 1960s.
  - Martin Richards* (Cambridge) designed the BCPL subset [Basic CPL] in 1966 (which was implemented in 1967)
- Used for systems programming (compilers, operating systems)
- BCPL was a major influence on the design of C
- The compiler emitted OCODE, which could be translated to native machine code; Cintcode was a **bytecode** for interpretation

* Of Richards benchmark fame

Interpretation technique #2: Bytecode interpretation

- Idea: Real machines (ie hardware) don’t have the issues of AST interpretation; let’s mimic a real machine
- Design an instruction set architecture for the language being interpreted, and write an interpreter for that ISA
- Real machines expose many details which are irrelevant to the guest language (e.g., the address of a variable on the stack)
    - Omit these details from the ISA spec., usually by abstraction
    - Example: use a stack, instead of registers

Example

- Spec. of expression language machine, using a stack of ints:

push *n* … Push an integer constant on the stack

push *v …* Push the value of variable *v* on the stack

op … (op=add|sub|mul|div) Pop the top two ints, apply *op*, push the result

pop *v*… Pop the stack into variable v

Example: control flow

- Index the bytecodes in the program
- Add conditional and unconditional branch instructions

jump *n* … Jump to bytecode at offset *n*

jeq *n …* Pop a value from the stack and jump to n if it is equal to zero

Ditto jne, jlt, jle, …

Bytecode interpreter

```
pc = address_of_first_instruction();
forever do:
    b = fetch_bytecode(pc);
    switch (opcode(b))
    case push:  push(get_value(field(b));
    case pop:    store(pop(), field(b));
    case add:    push(pop() + pop());
    …
    case jump:  pc = pc + field(b);
    …
```

Performance

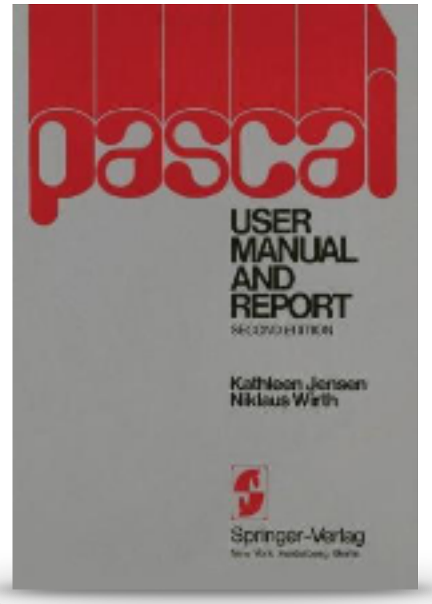
- A bytecode interpreter is typically a little faster than an equivalent AST interpreter — but not by much.

- It spends most of its time figuring out **what** to do, and only a little doing it.

- It's an easy way to make your language implementation portable.

- Modest optimizations can be done in the compiler from language to bytecode.

Pascal p-code



- Origins in a Pascal compiler developed in the mid-1970s at ETH Zurich

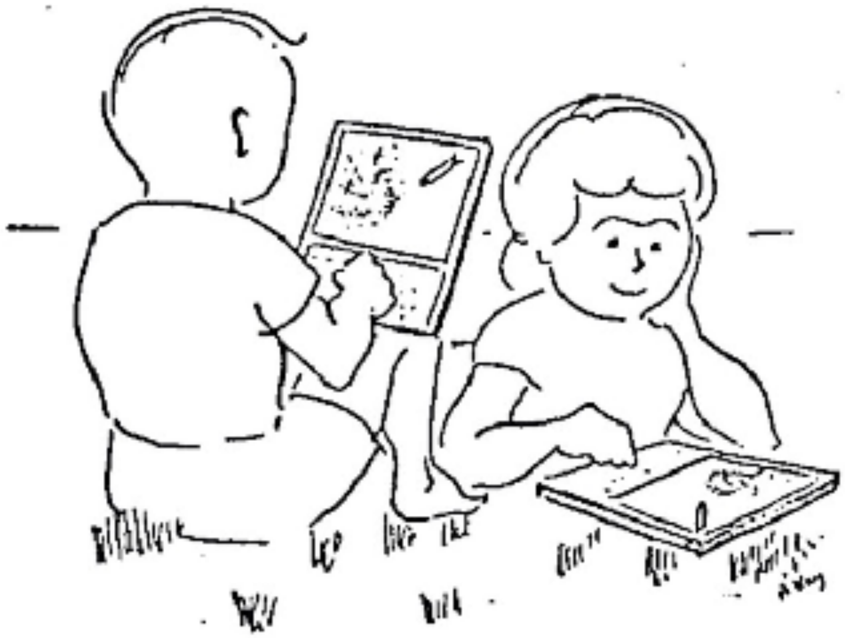
- Used in the UCSD p-System (OS) released in 1978, deployed widely for commercial use

- Stack machine, very simple, originally interpreted

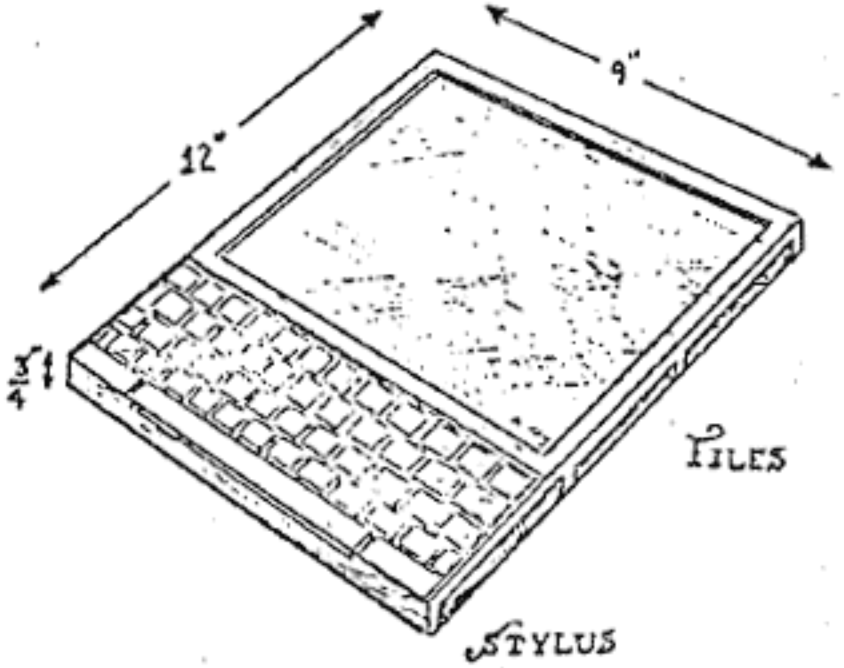
- Later: Hardware implementations: Western Digital’s Pascal MicroEngine, NCR, later Lilith (Modula-2 M-code)

Smalltalk

From the mid-1970s to the mid-1980s, Smalltalk took up the running in VM technology.



It’s 1969…

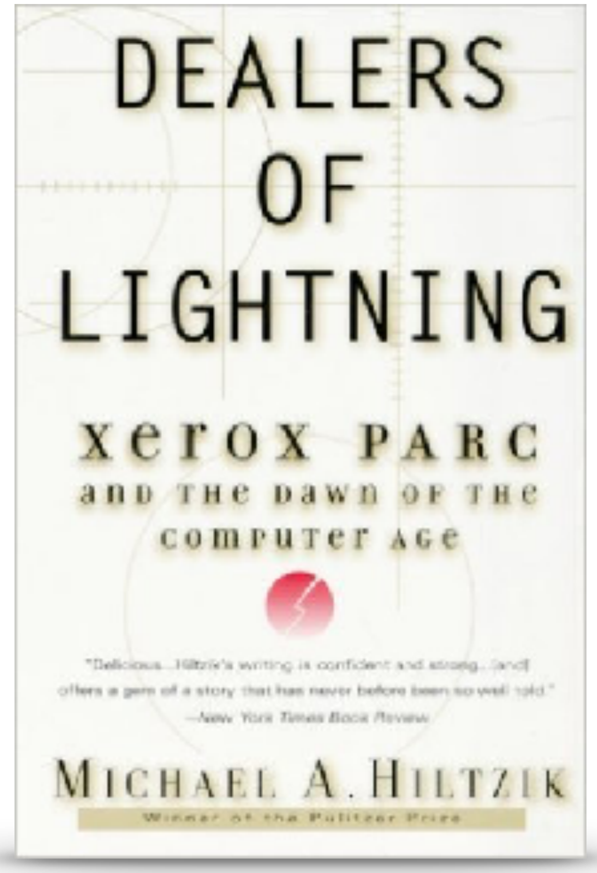


- Alan Kay’s Ph.D. thesis, *The Reactive Engine*, describes a future of personal, portable computers and speculates on how they will be programmed and used.

Early 1970s

- In 1970, Kay joins Xerox PARC (just created), forms the Learning Research Group, attracts other researchers, including Adele Goldberg.

- The Smalltalk language and system are invented and developed through several versions. The aim is to build a system capable of being used by children to learn.



DEALERS
OF
LIGHTNING

XEROX PARC
AND THE DAWN OF THE
COMPUTER AGE

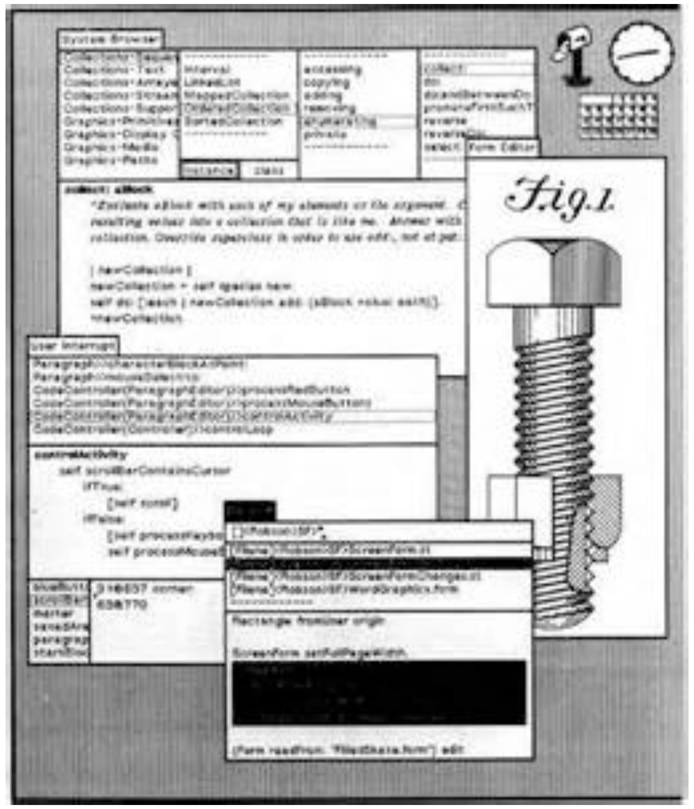
"Delicious...Hiltzik's writing is confident and strong...[and] offers a gem of a story that has never before been so well told."
—New York Times Book Review

MICHAEL A. HILTZIK

Winner of the Pulitzer Prize

Smalltalk-76 and the Alto

- PARC develops the Alto workstation — the “interim Dynabook” — a personal computer with high-resolution bitmapped graphics, local storage and a fast network connection.
- Smalltalk-76 is honed for the Alto. BitBlt, copy and paste are invented.

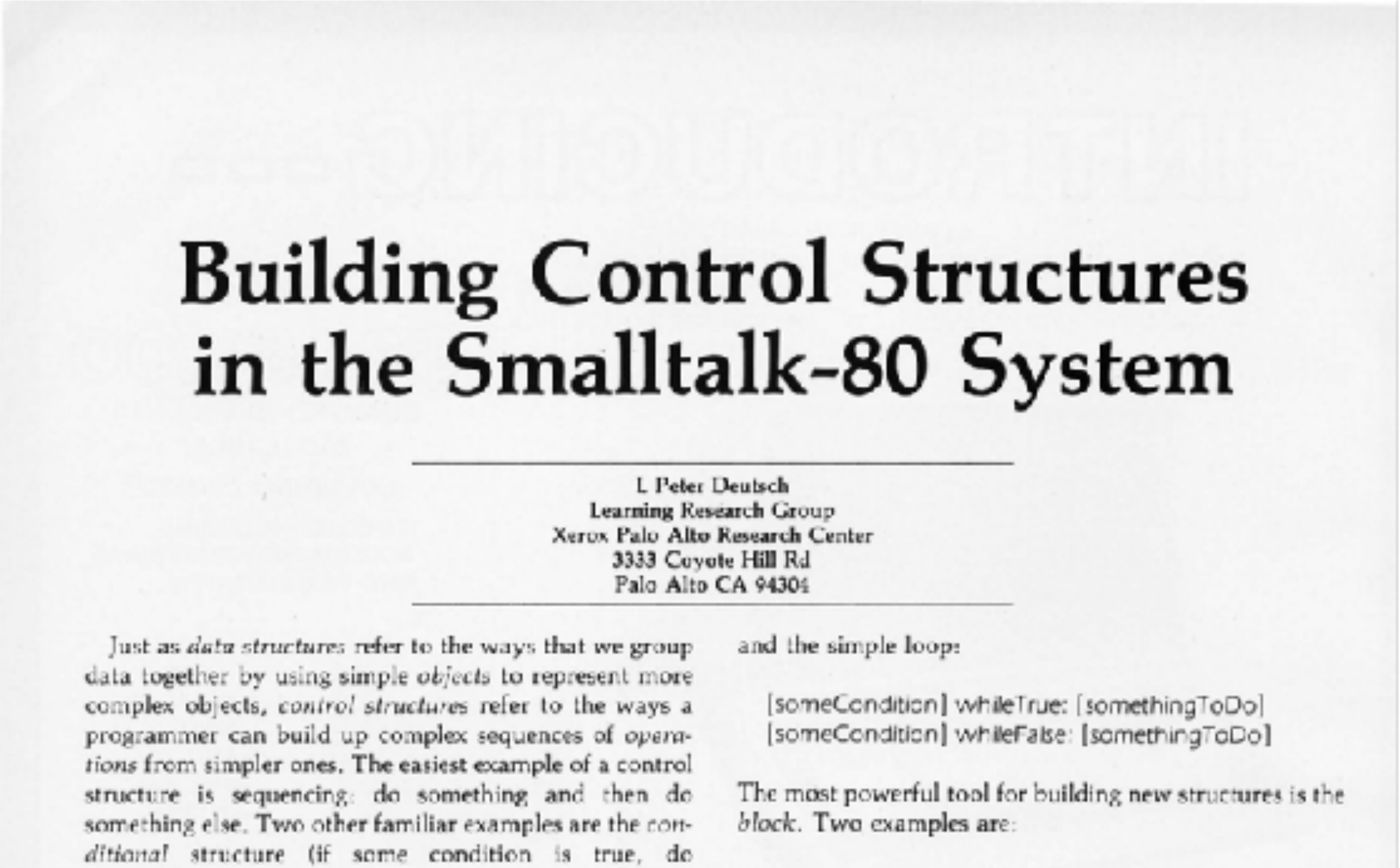


```
collect: aBlock
    | newCollection |
    newCollection ← self species new.
    self do: [:each | newCollection add: (aBlock value: each)].
    ↑newCollection
```

```
controlActivity
    self scrollBarContainsCursor
        ifTrue:
            [self scroll]
        ifFalse:
```

1981–3: Smalltalk-80 is released to the world

- *BYTE* special issue (Aug 1981)



Building Control Structures in the Smalltalk-80 System

L Peter Deutsch
Learning Research Group
Xerox Palo Alto Research Center
3333 Coyote Hill Rd
Palo Alto CA 94304

Just as *data structures* refer to the ways that we group data together by using simple *objects* to represent more complex objects, *control structures* refer to the ways a programmer can build up complex sequences of *operations* from simpler ones. The easiest example of a control structure is sequencing: do something and then do something else. Two other familiar examples are the *conditional* structure (if some condition is true, do

and the simple loop:

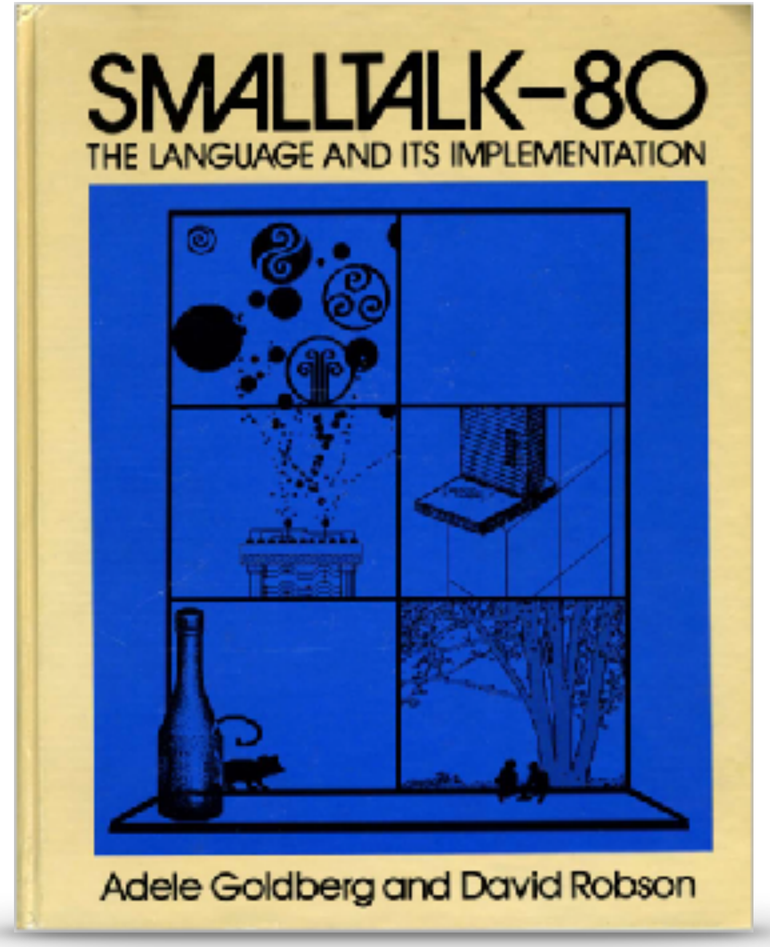
```
[someCondition] whileTrue: [somethingToDo]
[someCondition] whileFalse: [somethingToDo]
```

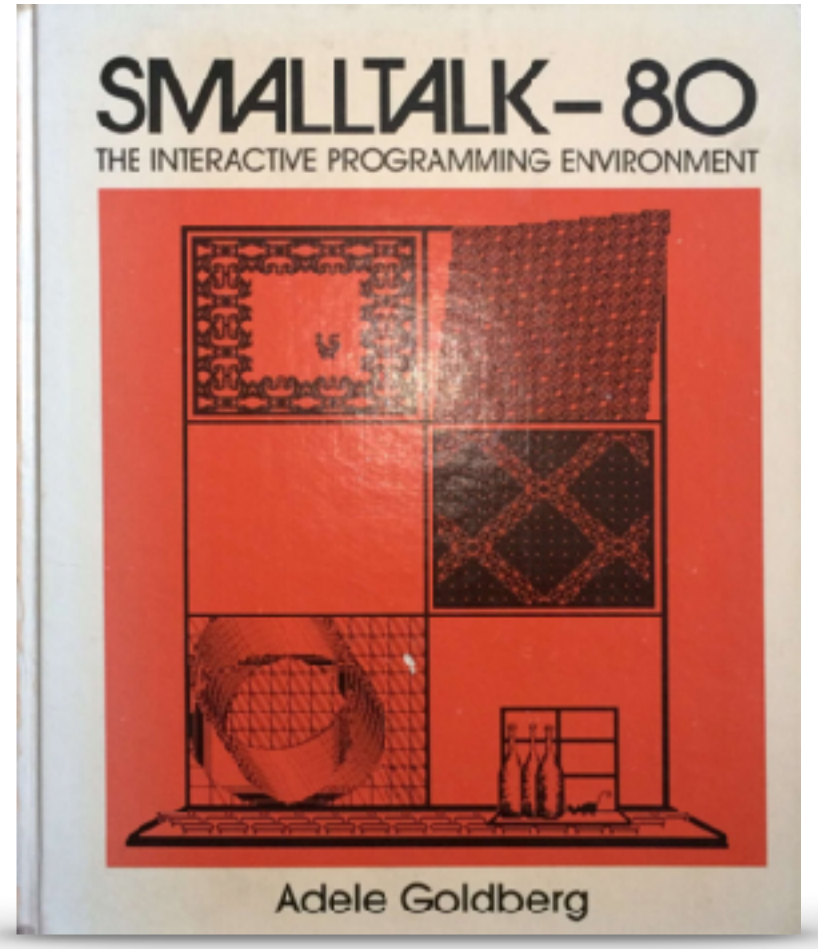
The most powerful tool for building new structures is the *block*. Two examples are:

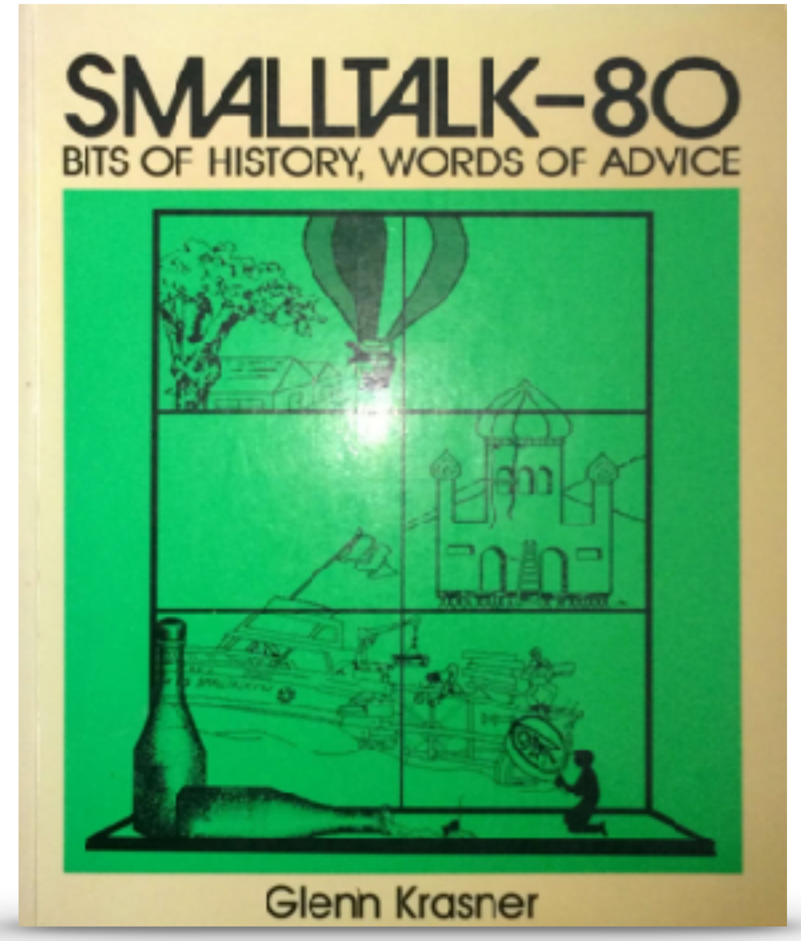
BYTE
the smal system
AUGUST 1981 Vol. 6, No. 8
$2.50 in USA/$2.95 in Canada
A McGraw-Hill Publication
SMALLTALK

1981–3: Smalltalk-80 is released to the world

- The “colored books” (1983)


SMALLTALK-80
THE LANGUAGE AND ITS IMPLEMENTATION
Adele Goldberg and David Robson



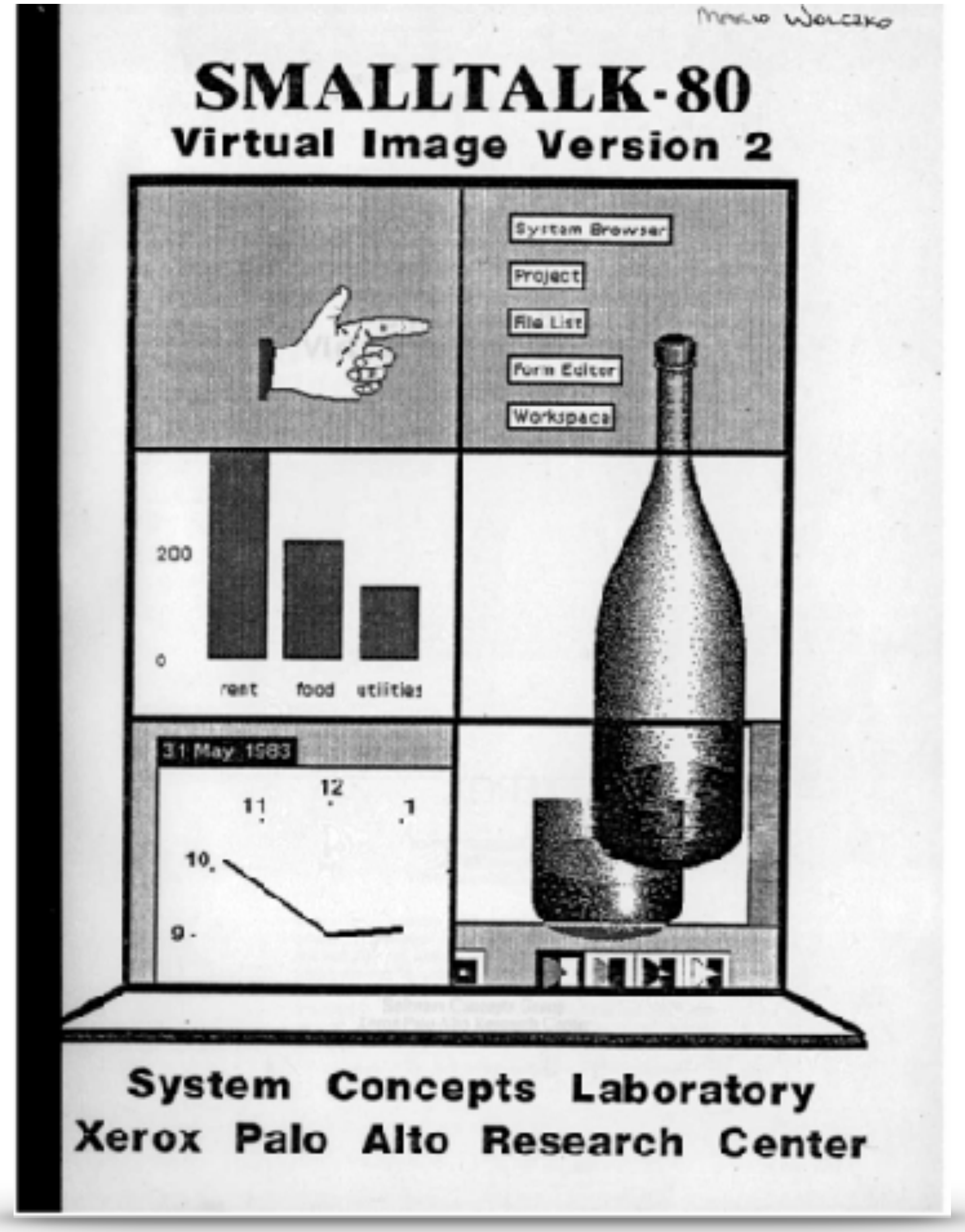


Smalltalk-80 Virtual Machine

- Defined by a reference implementation (in Smalltalk!) in the Blue Book
- Bytecode ISA, object memory
    - Classes, metaclasses, method activations are objects!

1981–3: Smalltalk-80 is released to the world

- The tape (1983)
- All the objects
- Roll your own VM!
  - Slow! (see Green Book)



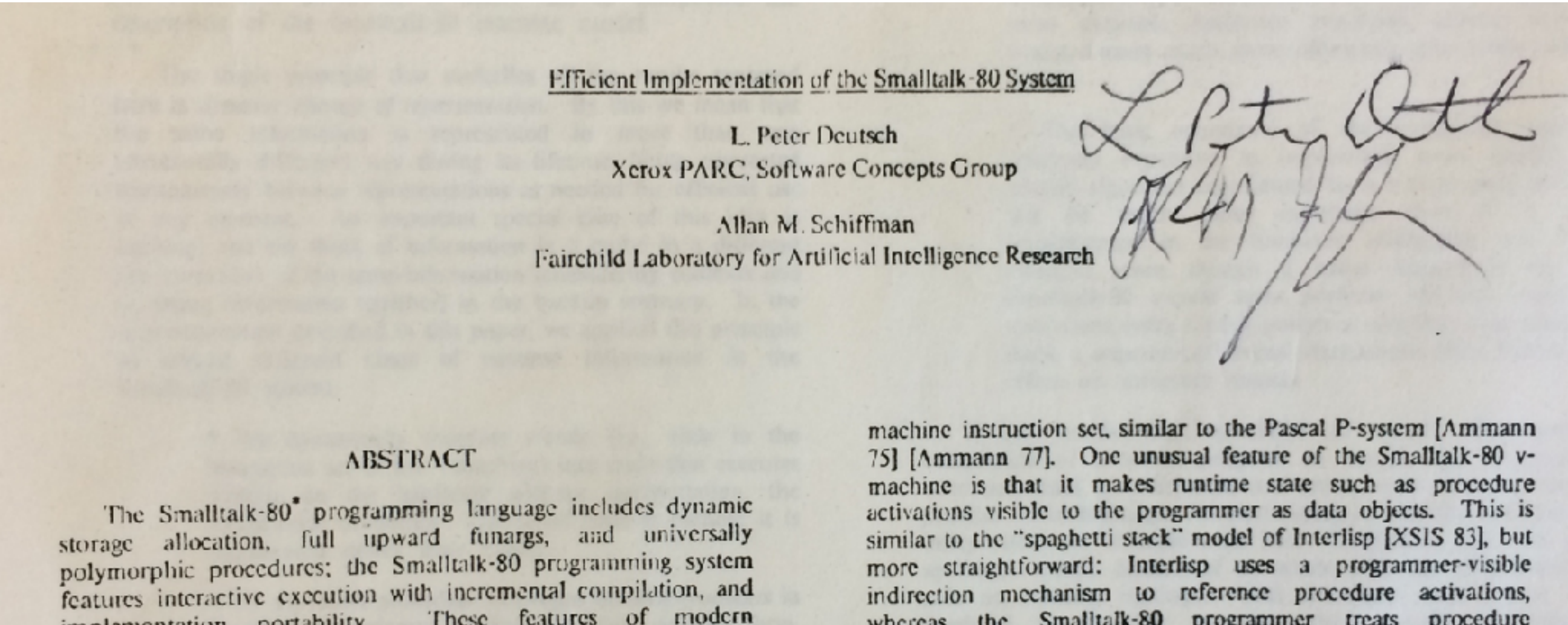
Mario Wolczko

SMALLTALK-80
Virtual Image Version 2

System Concepts Laboratory
Xerox Palo Alto Research Center

PS

- *Efficient Implementation of the Smalltalk-80 System*, L Peter Deutsch and Allan M Schiffman, POPL 1984



Efficient Implementation of the Smalltalk-80 System

L. Peter Deutsch
Xerox PARC, Software Concepts Group

Allan M. Schiffman
Fairchild Laboratory for Artificial Intelligence Research

ABSTRACT

The Smalltalk-80* programming language includes dynamic storage allocation, full upward funargs, and universally polymorphic procedures; the Smalltalk-80 programming system features interactive execution with incremental compilation, and implementation portability. These features of modern

machine instruction set, similar to the Pascal P-system [Ammann 75] [Ammann 77]. One unusual feature of the Smalltalk-80 v-machine is that it makes runtime state such as procedure activations visible to the programmer as data objects. This is similar to the "spaghetti stack" model of Interlisp [XSIS 83], but more straightforward: Interlisp uses a programmer-visible indirection mechanism to reference procedure activations, whereas the Smalltalk-80 programmer treats procedure

The paper: contributions

- **Just-In-Time (JIT) translation of Smalltalk bytecode to machine code**; code caching and lookup
- Inline caching of message send targets
- On-demand conversion of contexts (activation records) from on-stack to hybrid and heap-allocated forms
- Deferred reference counting (described in a 1976 paper by Deutsch & Bobrow)
- For more detail, watch CS294 session on youtube.

A JIT compiler eliminates interpreter dispatch overhead

- Simple elimination of interpreter overhead leads to a significant speedup. This is because we perform bytecode decoding only **once**, and the generated code does not have the overhead of the interpreter loop (or other dispatch for more advanced forms of interpretation).

Self
1987–1995

- Language designed in 1987 as a successor to Smalltalk; even simpler and more regular

- Objects, slots, methods, messages

- VM had only 8 bytecodes!

- Stanford & PARC 1987—1992
  Sun Labs 1992—1995

Self implementation innovations

From JIT to adaptive, feedback-driven optimization (to come after JIT compilation):

- **Optimizing compilation of a dynamic language**
- Type feedback
- Adaptive optimization
- PICs, maps, generational heap
- C++ implementation tricks

Challenges of dynamic languages

- Types are unknown ahead of time
- Types within an expression may vary over time
- Intermingling of structured and primitive values
- Varying code
    - Programmer changes
    - Self-modifying code

Self 3.0
(released 1993)

- **Feedback-driven adaptive optimization** (Urs Hölzle's thesis, 1994):
    - Polymorphic Inline Caches (PICs) and counters
    - Adaptive inlining
    - **Deoptimization**

Inlining

- Inlining is an important optimization, not just because it removes call overhead, but because it increases the size of the compilation unit, and exposes more code to optimization.

- Improperly applied, it can lead to code bloat. Prior to its application in dynamic compilers, automatic inlining was of marginal use, because it could not be applied wisely.

What inlining enables

- Connects data flows across methods
- Connect control flows across methods
    - Turns exceptions and other non-local control into jumps
- Combined data- and control-flow analysis can, e.g., eliminate closures entirely.
- Opens the door to further optimizations (including more inlining)

Self 3 architecture

- JIT-compile code on first execution
- Instrument the emitted code to measure execution frequencies, and observe types in use and actual call edges
- Recompile hotspots, using gathered info to drive inlining and other optimizations
- Result: rapid execution with acceptable warm-up

Dynamic deoptimization

- Many potential optimizations are speculative: they are based on the current state of the program and/or data, which may change.
  - Example: Java class loading can invalidate inlining decisions
- If this occurs, we need a technique to recover the state of the computation, abandon the incorrect optimizations, and proceed with the correct behavior.

Dynamic deoptimization

```
m() {x := 3; y := 4; foo(x, y);}
```

Dynamic deoptimization

```
m() {x := 3; y := 4; foo(x, y);}
```

```
foo(i, j) { bar(); return baz(i, j); }
```

Dynamic deoptimization

```
m() {x := 3; y := 4; foo(x, y);}
```

```
foo(i, j) { bar(); return baz(i, j); }
```

```
baz(p, q) { return p+q; }
```

Dynamic deoptimization

```
m() {x := 3; y := 4; foo(x, y);}
```

```
foo(i, j) { bar(); return baz(i, j); }
```

```
baz(p, q) { return p+q; }
```

```
m’() {bar(); return 7;}
```

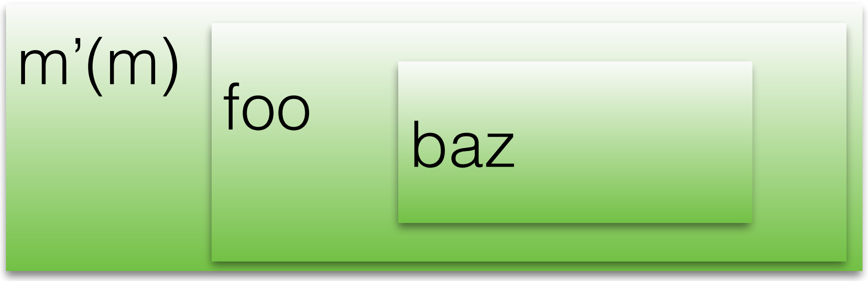
Dynamic deoptimization

```
m() {x := 3; y := 4; foo(x, y);}
```

```
foo(i, j) { bar(); return baz(i, j); }
```

```
baz(p, q) { return p+q; }
```

```
m’() {bar(); return 7;}
```



inlining tree

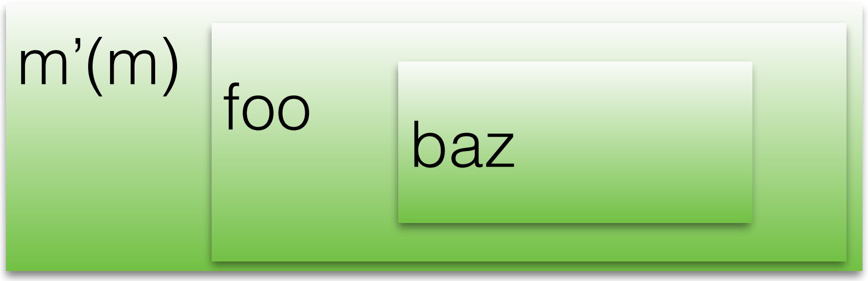
Dynamic deoptimization

```
m() {x := 3; y := 4; foo(x, y);}
```

```
foo(i, j) { bar(); return baz(i, j); }
```

```
baz(p, q) { return p+q; }
```

```
m'() {bar(); return 7;}
```



m’(m)

foo

baz

inlining tree

frame for m’:
in call to bar()

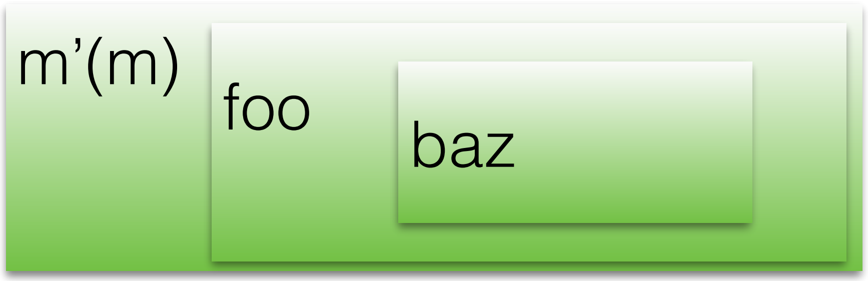
Dynamic deoptimization

```
m() {x := 3; y := 4; foo(x, y);}
```

```
foo(i, j) { bar(); return baz(i, j); }
```

```
baz(p, q) { return p+q; }
```

```
m’() {bar(); return 7;}
```

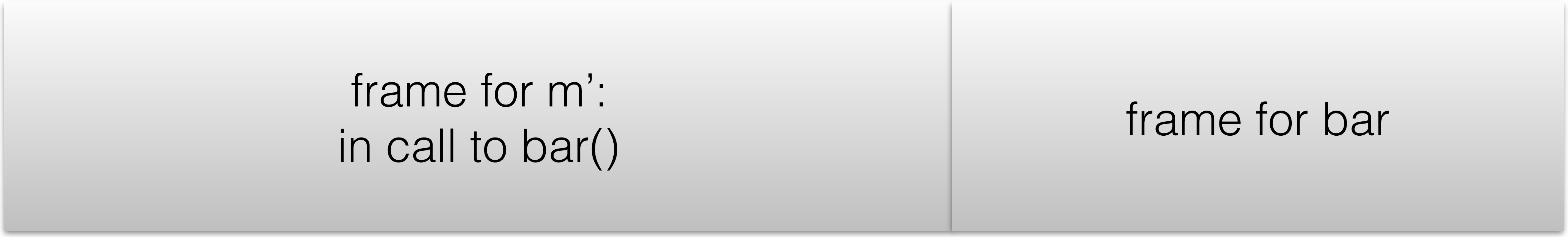


m’(m)

foo

baz

inlining tree



frame for m’:
in call to bar()

frame for bar

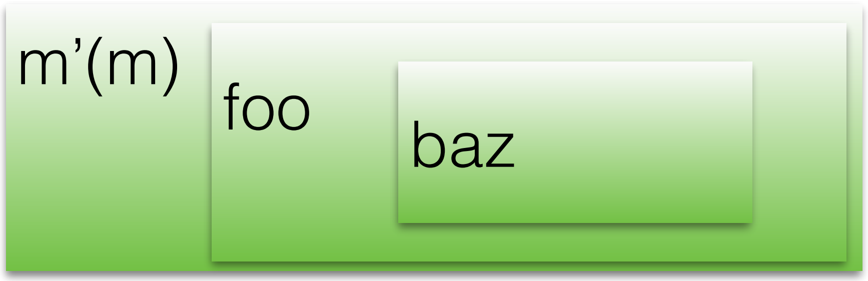
Dynamic deoptimization

```
m() {x := 3; y := 4; foo(x, y);}
```

```
foo(i, j) { bar(); return baz(i, j); }
```

```
baz(p, q) { return p+q; }
```

```
m'() {bar(); return 7;}
```



inlining tree

frame for m':
in call to bar()

frame for bar

fram

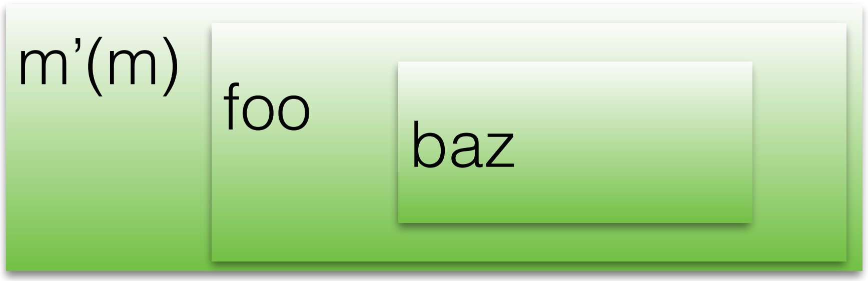
Dynamic deoptimization

```
m() {x := 3; y := 4; foo(x, y);}
```

```
foo(i, j) { bar(); return baz(i, j); }
```

```
baz(p, q) { return p*q; }
```

```
m'() {bar(); return 7;}
```



inlining tree

frame for m':
in call to bar()

frame for bar

fram

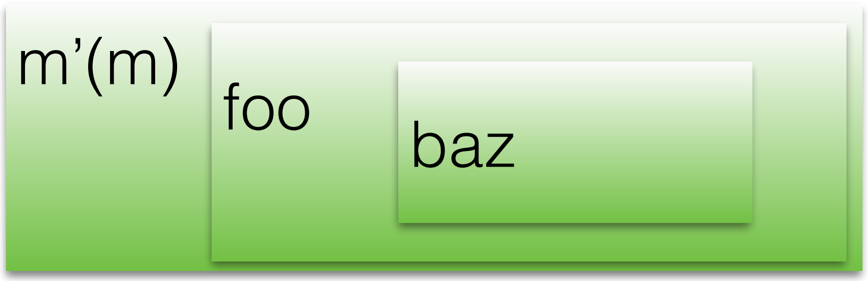
Dynamic deoptimization

```
m() {x := 3; y := 4; foo(x, y);}
```

```
foo(i, j) { bar(); return baz(i, j); }
```

```
baz(p, q) { return p*q; }
```

```
m'() {bar(); return 7;}
```



m'(m)

foo

baz

inlining tree

frame for m:
in call to foo()
x = 3
y = 4
	frame for foo:
	in call to bar()
		frame for bar	fram

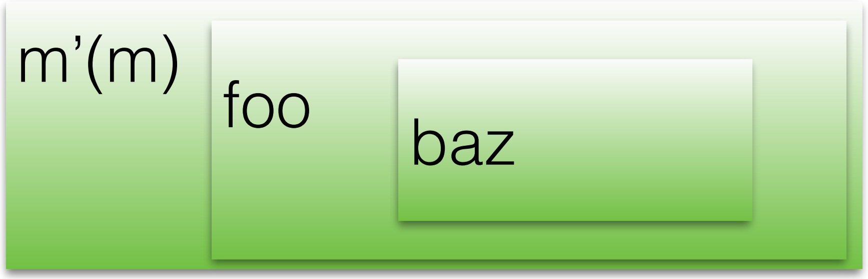
Dynamic deoptimization

```
m() {x := 3; y := 4; foo(x, y);}
```

```
foo(i, j) { bar(); return baz(i, j); }
```

```
baz(p, q) { return p*q; }
```

```
m'() {bar(); return 7;}
```



inlining tree

frame for m:
in call to foo()
x = 3
y = 4

frame for foo:
in call to bar()

frame for bar:
return to foo()

fram

On-stack replacement

Suppose we have a long-running loop. When we first execute the loop, all the methods are unoptimized. Part way through, we trigger a counter and invoke optimizing compilation of the loop and its callees. How do we transition to the optimized code before waiting for the loop to end?

Solution: use the same frame-replacement techniques, except this time replacing unoptimizing frames with their optimized counterparts.

Java

1995—present

VMs enter the mainstream

The Java Virtual Machine

- Java emerged shortly after the World Wide Web was invented; when dissatisfaction with C and especially C++ as an application language was high; and when OOP was hugely popular.
    - Portable binaries + type-safe + objects

The JVM

- The JVM was based on familiar ideas: a machine-independent bytecode ISA; automatic memory management (GC); objects and methods.

- It added a class-level distribution format, sandbox security (applets), static typing and bytecode/class verification.

- Massive adoption made bytecode VMs and those implementation techniques ubiquitous.

JVM developments 1995–2000

- Early JVMs (1995-1998) were just playing catch-up with Smalltalk and Self.
    - Many simple JIT compilers were written
    - Java’s built-in concurrency added new challenges and opportunities

HotSpot

- The “Java HotSpot Virtual Machine” (1999), incorporated many of the techniques from Self…unsurprising, as developed by an ex-Selfer, following a pivot from Smalltalk:
  - Inlining, PICs, counters, deopt
- Added new techniques for Java’s peculiarities, and careful engineering to take advantage of static types:
  - Fast locking, virtual table dispatch, …
- A subsequent release incorporated the Server Compiler (C2), which brought SSA-based heavy-duty code optimization techniques, taking performance well beyond that of Self-era compilers (such as the first HotSpot compiler, and the Client Compiler (C1)).

Later innovations used in JVM implementations

- Escape analysis
- Biased locking
- Thread-local allocation buffers
- Separable compiler(s)
- Concurrent GC
    - Lots of techniques

4. Language VMs

Part 2, 2000–2010

Proliferation

- By the mid-2000s, language VM technology had been widely deployed (on perhaps a billion devices, from cellphones to supercomputers)

- The bulk of the implementation effort had gone into JVMs (Sun, IBM) and the CLR (Microsoft).

- In contrast, the performance of other managed languages (JavaScript, Python, Perl, etc.) was lackluster.

JavaScript Wars

- Language was invented by Eich at Netscape in 1995
- By mid-2000s, it was still the only viable language of the web, but was interpreted.
    - OK for web-page one-liners, not for web applications. AJAX made sophisticated applications possible.
- In the late 2000s, several companies invested heavily to develop high-performance JavaScript VMs:
    - Mozilla: TraceMonkey — trace compilation comes to language VMs
    - Google: V8 — very similar to Self (maps) and HotSpot
    - Apple: WebKit/SquirrelFish (later Nitro)

Trace compilation

- In a binary translator, traces are a more obvious choice for translation unit. In a language VM, the linguistic constructs are available — so why use traces?

- Linear traces are easy to compile quickly

- Inlining comes for “free” — traces span call boundaries

- Well-described by Gal et al., HotPathVM, 2006 (a JVM)

- Used in Mozilla’s TraceMonkey, c.2008

Some disadvantages of writing a VM in C/C++

- Lack of safety — occasionally essential
- Two runtimes with differing views of the world: managed and unmanaged
- Result: **building a high-performance VM in C/C++ requires extraordinary skill and great effort.**

Writing a VM in a high(er)-level language

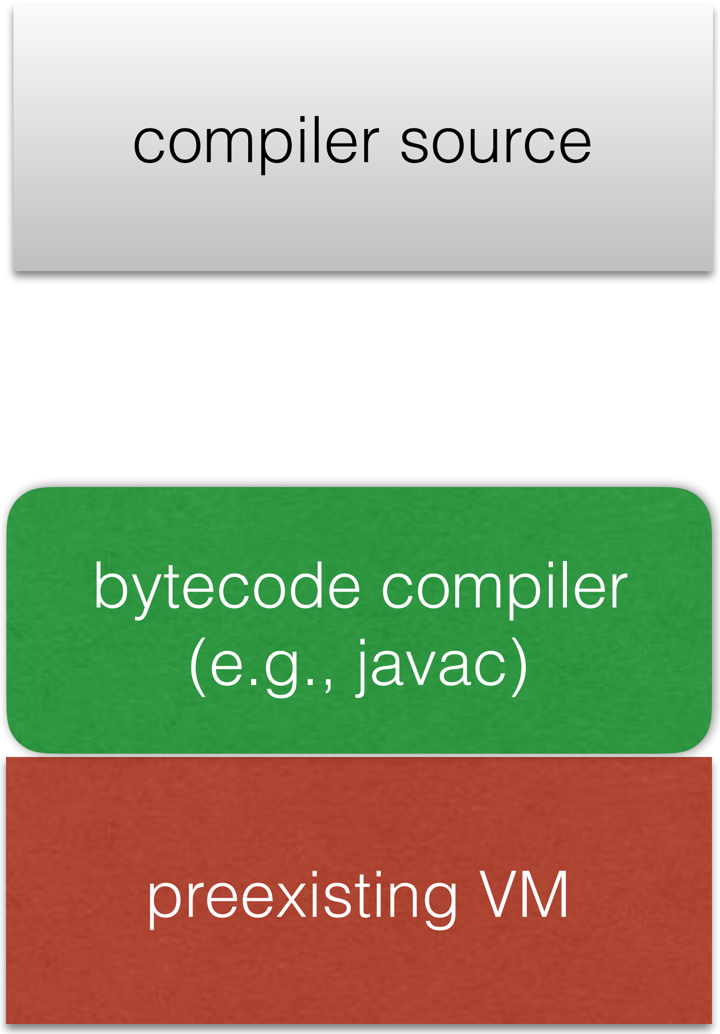
- These issues have led to attempts to write VMs in other languages, to decrease the skill and effort level required. Desiderata:
  - Higher-level (e.g., type- and memory-safe);
  - Better low-level control when needed (to avoid assembly)
  - Uniform and preferably automatic handling of references, safe points, calling conventions, etc.

Metacircularity — with performance

Some systems generated C from a higher-level language (e.g., Squeak, which used Smalltalk). That only address part of the problem.

To get performance together with the benefits of a higher-level language, we can adopt an architecture in which a single compiler can serve to build the VM and also to compile applications.

Metacircularity — with performance

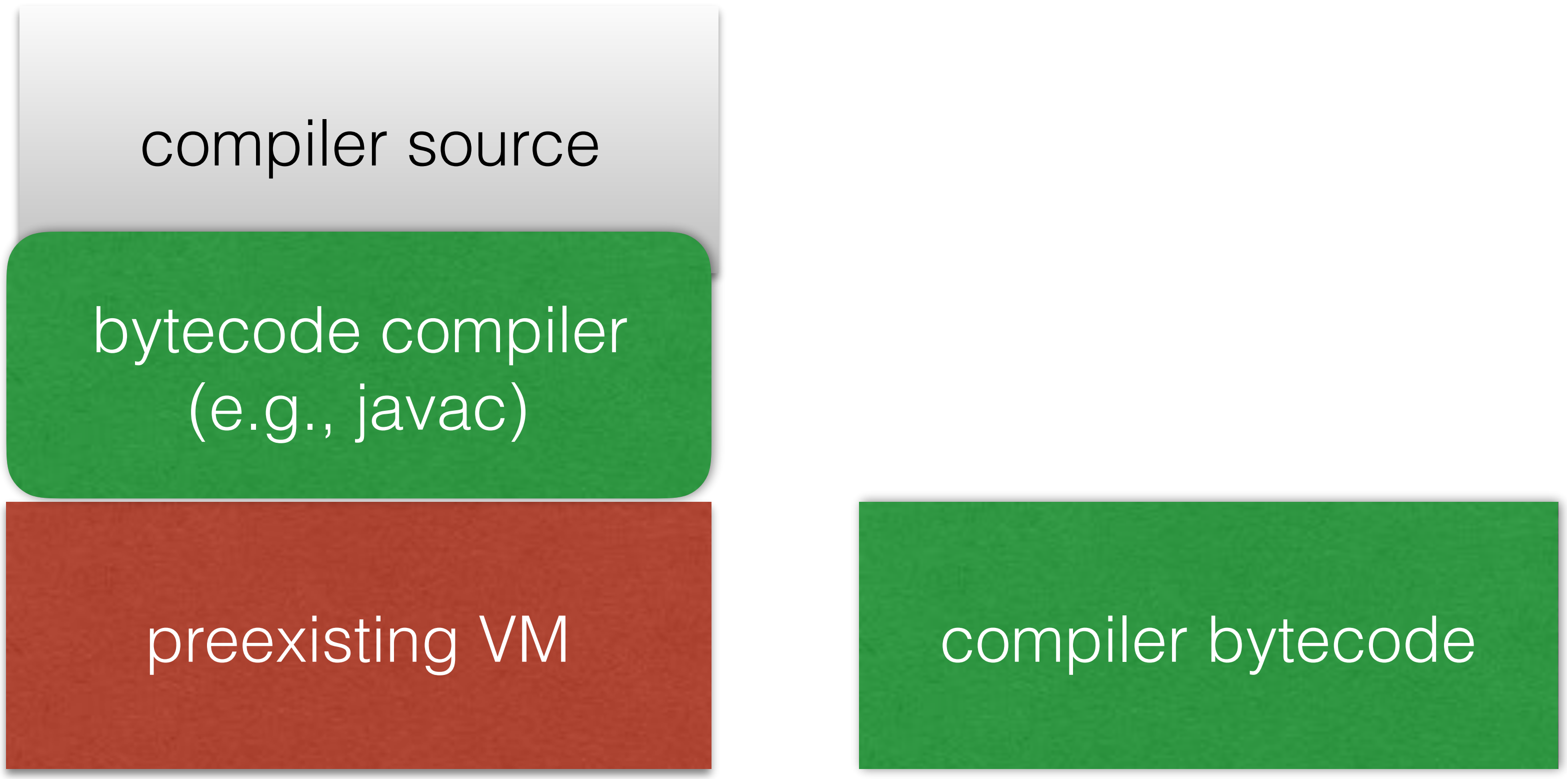


compiler source

bytecode compiler (e.g., javac)

preexisting VM

Metacircularity — with performance



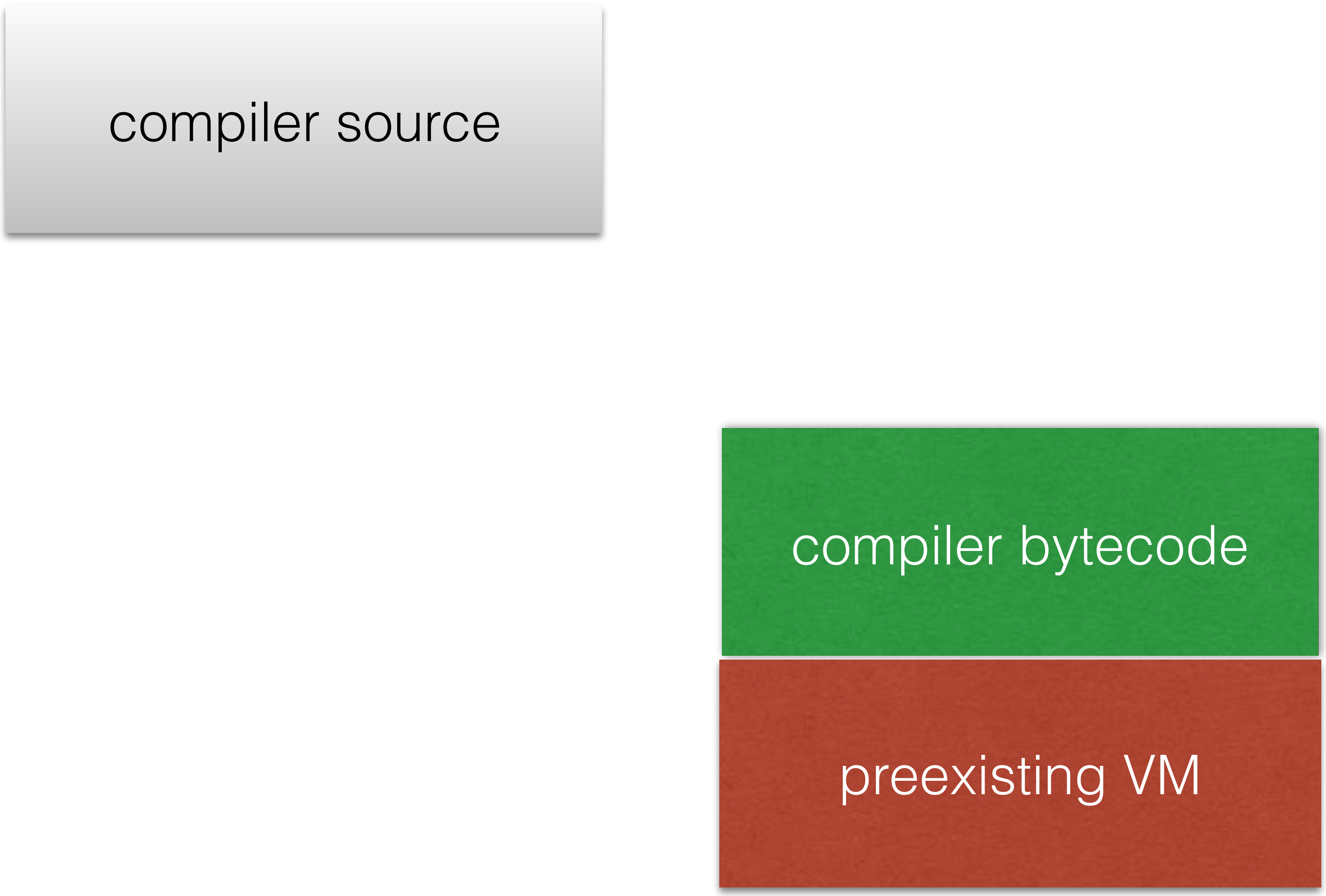
compiler source

bytecode compiler (e.g., javac)

preexisting VM

compiler bytecode

Metacircularity — with performance

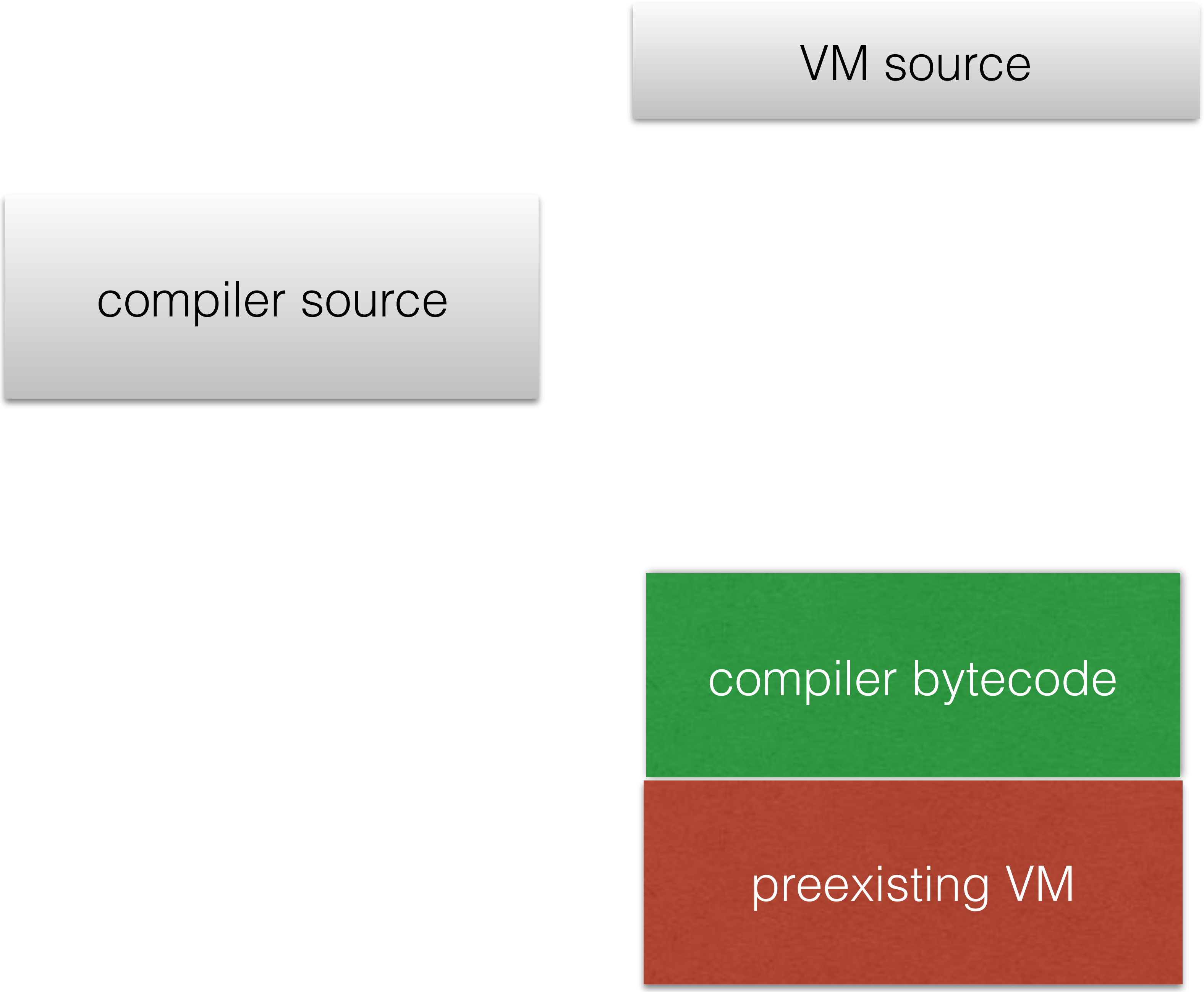


compiler source

compiler bytecode

preexisting VM

Metacircularity — with performance



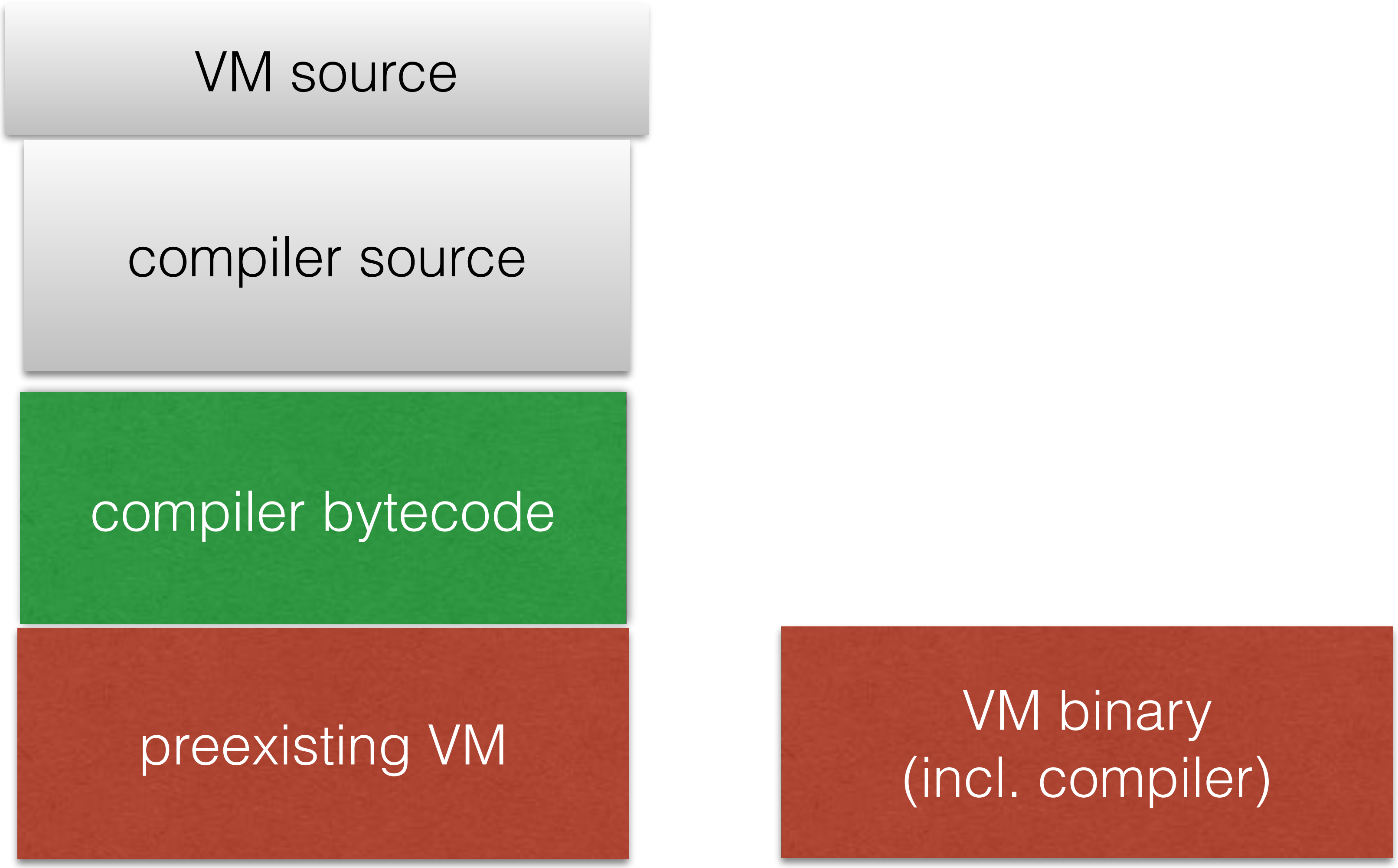
VM source

compiler source

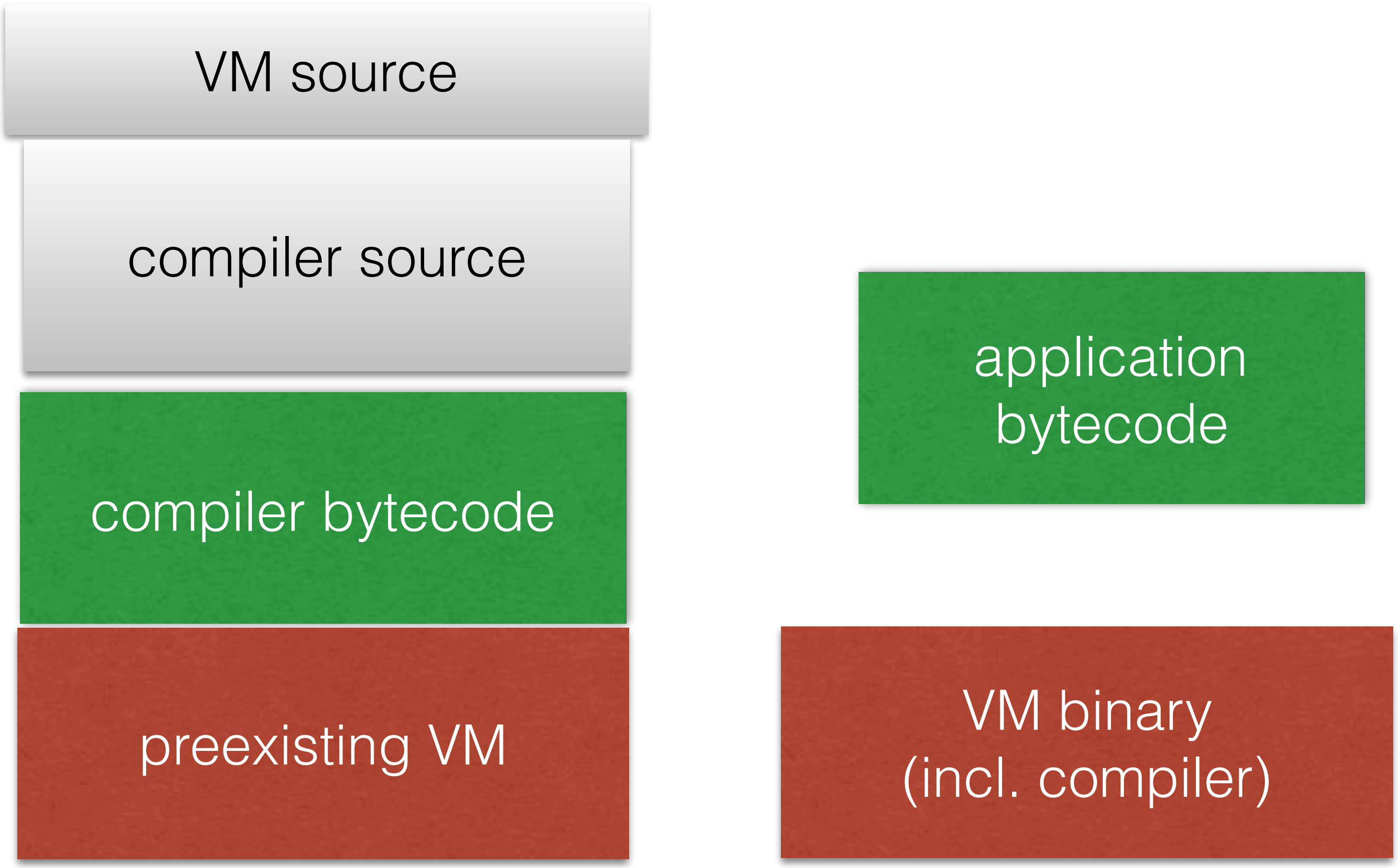
compiler bytecode

preexisting VM

Metacircularity — with performance



Metacircularity — with performance



VM source

compiler source

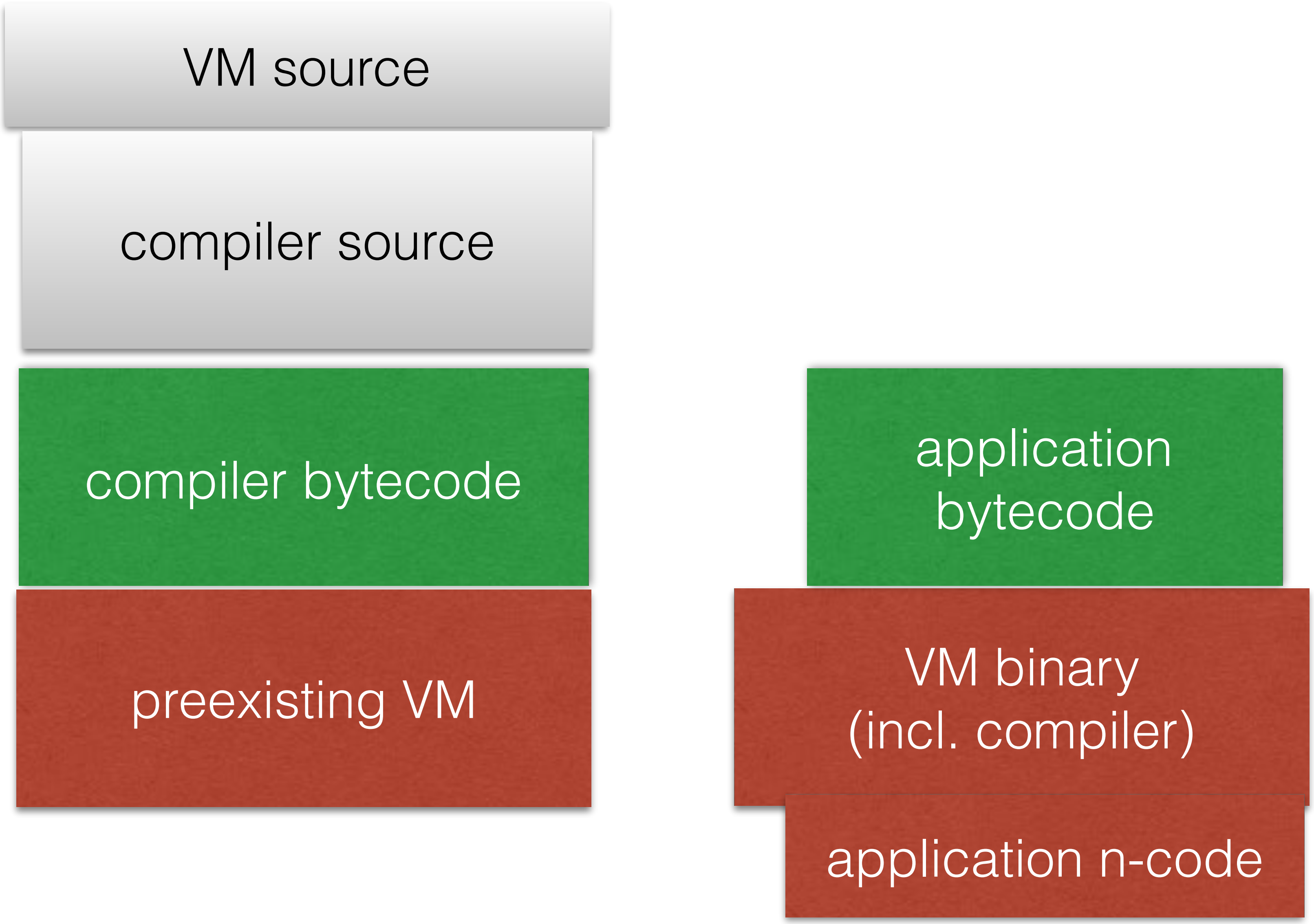
compiler bytecode

preexisting VM

application bytecode

VM binary (incl. compiler)

Metacircularity — with performance



VM source

compiler source

compiler bytecode

preexisting VM

application bytecode

VM binary
(incl. compiler)

application n-code

Advantages

The same compiler is being used for the VM as for the application

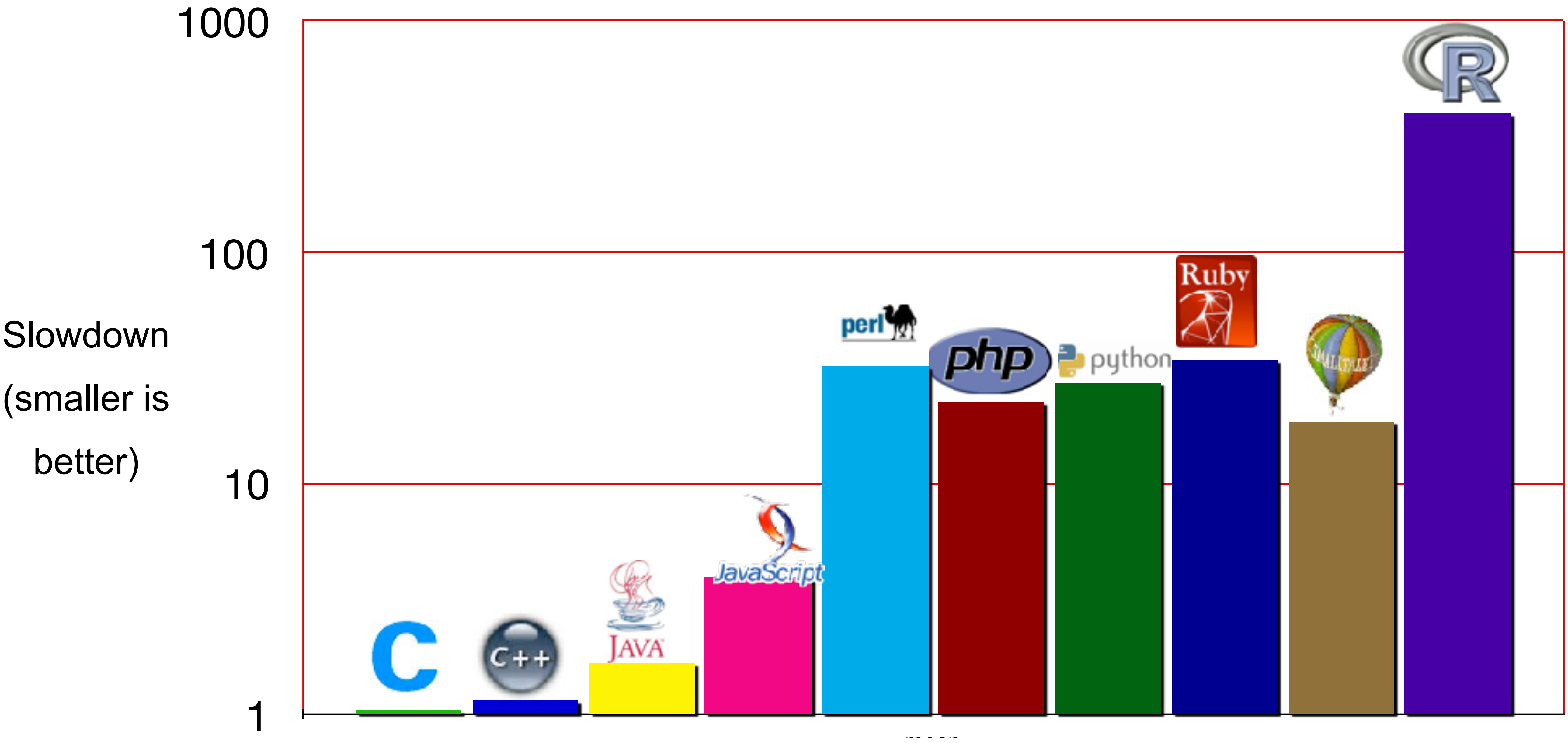
- Common calling convention
- Can inline VM code into application
- Common handling of safe points, references

5. Multilingual VM Frameworks

2010—present

Relative speeds of various languages

From the Computer Language Benchmarks Game, ~2013



Can we build a language-independent VM framework in which many languages can be implemented (more) easily?

What is needed to generate code for a user program?

1. The user program

2. Expressed semantics of each language element

Combine the semantics of each element of the user program and generate code for the combined result.

This is what a traditional compiler does. But are there alternatives?

Compilation without a guest language compiler:
1. Metatracing

- Express the language semantics as a bytecode interpreter in a relatively high-level language

- Modify the interpreter to gather bytecode execution traces from the guest program

- Combine the traces with the interpreter's actions to generate code for each trace; like unrolling the interpreter.

- Together with some hints and optimizations, can generate pretty good code.

PyPy

- Originally, a Python VM written in a subset of Python, *RPython* (*R*estricted — types can be inferred, and it is easily translated). Generated C code or LLVM IR.

- Subsequently, a framework for the implementation of multiple languages via meta-tracing.

- *Tracing the meta-level: PyPy's tracing JIT compiler*, Bolz et al., 2009.

- Good performance for a variety of languages: Python, Ruby, Prolog, PHP, …

Compilation without a guest language compiler:
2. Partial evaluation of ASTs

- Express the language semantics as an AST interpreter in a relatively high-level language

- Combine the guest application’s ASTs with the interpreter semantics; generate code

Example

Consider a simple expression AST interpreter:

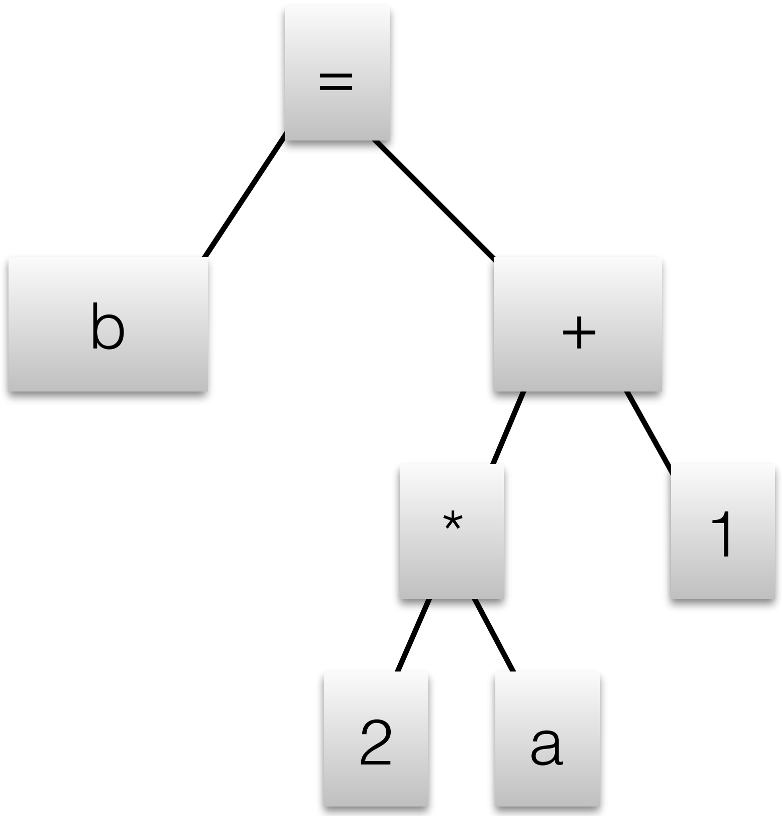
How can we compile code for an expression such as b=2*a+1?

```
class ASTNode { …stuff common to all nodes…}

abstract class ExprNode extends ASTNode {
  abstract int eval();
}

class ConstNode extends ExprNode { int val; int eval() { return val; } … }

class AddNode extends ExprNode {
  ExprNode left, right;
  int eval() { return left.eval()+right.eval(); … }
…}

class VarNode extends ExprNode {
  int val;
  void set(int v) { val=v; }
  int eval() { return val; } …
…}

class AssignNode extends ExprNode {
  VarNode var; ExprNode rhs;
  int eval() { int rhsVal=rhs.eval(); var.set(rhsVal); return rhsVal; }
…}
```

```
ConstNode>eval() { return val; }
AddNode>eval() { return left.eval()+right.eval(); }
VarNode>eval() { return val; }
AssignNode>eval() { return var.set(rhsVal.eval()); }
```

Compiling a simple expression

b=2*a+1

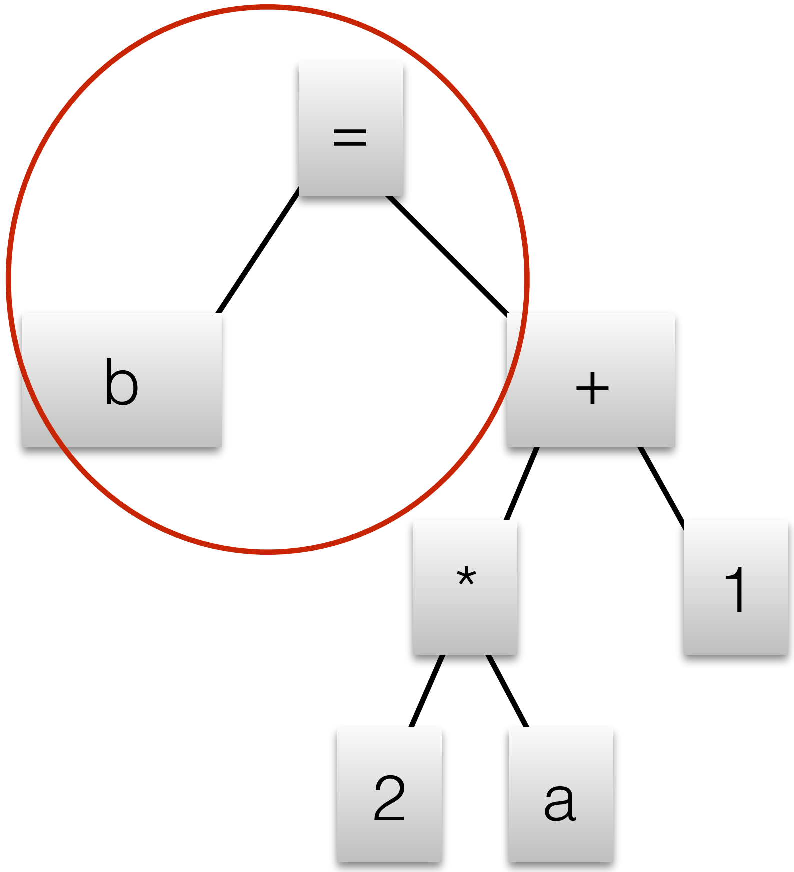


```
ConstNode>eval() { return val; }
AddNode>eval() { return left.eval()+right.eval(); }
MulNode>eval() { return left.eval()*right.eval(); }
VarNode>eval() { return val; }
AssignNode>eval() { return var.set(rhsVal.eval()); }
```

Compiling a simple expression

b=2*a+1

```
b.var.set(                                        )
```

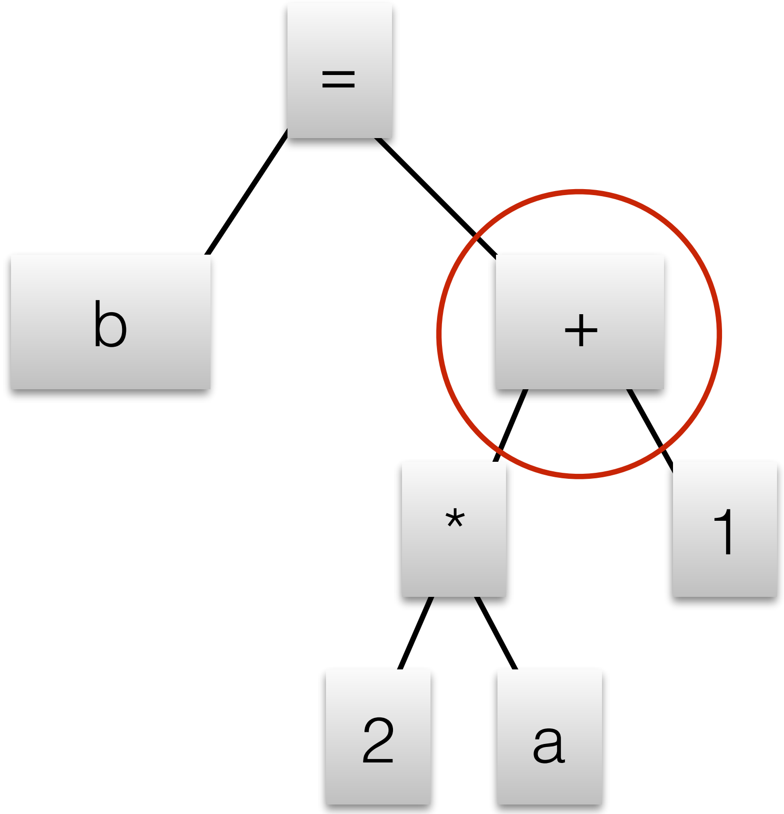


```
ConstNode>eval() { return val; }
AddNode>eval() { return left.eval()+right.eval(); }
MulNode>eval() { return left.eval()*right.eval(); }
VarNode>eval() { return val; }
AssignNode>eval() { return var.set(rhsVal.eval()); }
```

Compiling a simple expression

b=2*a+1

```
b.var.set(                ()+()                )
```

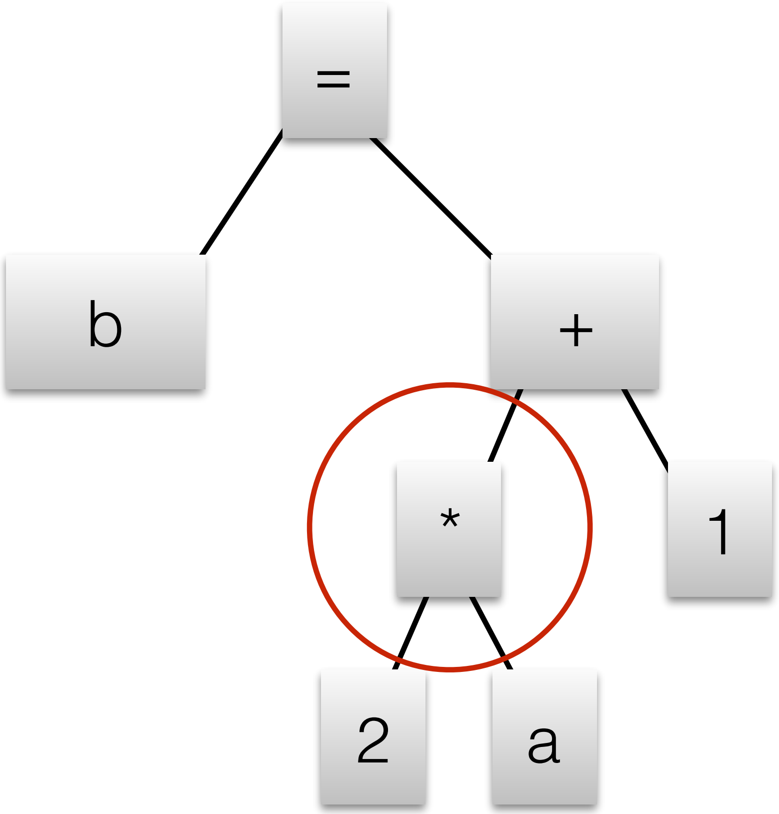


```
ConstNode>eval() { return val; }
AddNode>eval() { return left.eval()+right.eval(); }
MulNode>eval() { return left.eval()*right.eval(); }
VarNode>eval() { return val; }
AssignNode>eval() { return var.set(rhsVal.eval()); }
```

Compiling a simple expression

b=2*a+1

```
b.var.set(          ()*()+()                    )
```

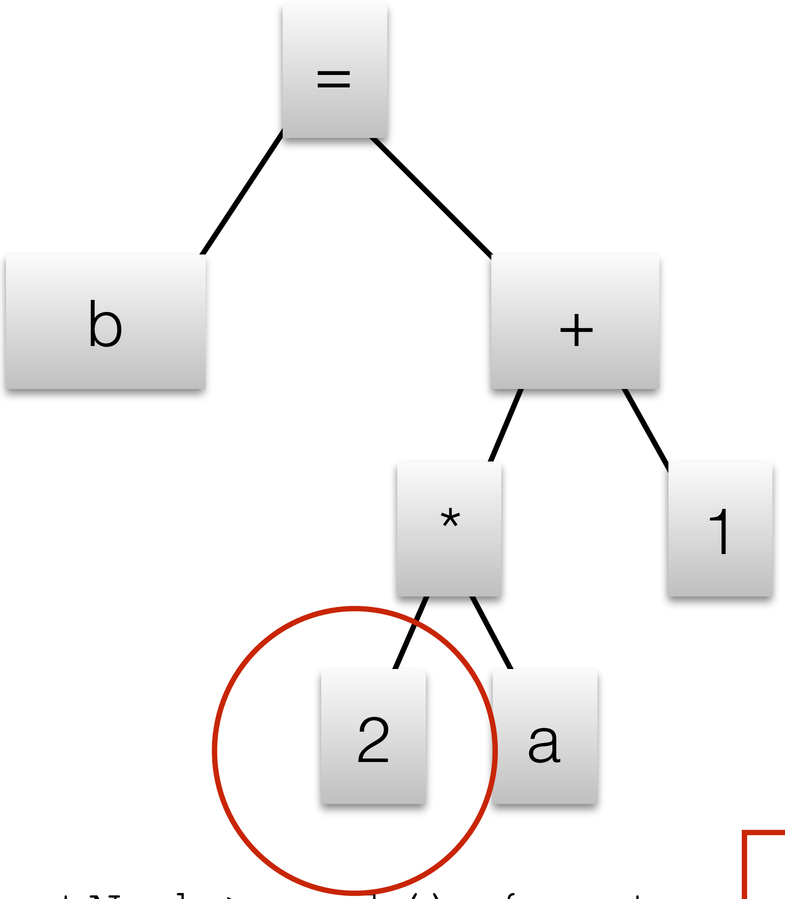


```
ConstNode>eval() { return val; }
AddNode>eval() { return left.eval()+right.eval(); }
MulNode>eval() { return left.eval()*right.eval(); }
VarNode>eval() { return val; }
AssignNode>eval() { return var.set(rhsVal.eval()); }
```

Compiling a simple expression

b=2*a+1

```
b.var.set(          2 *()+()                    )
```

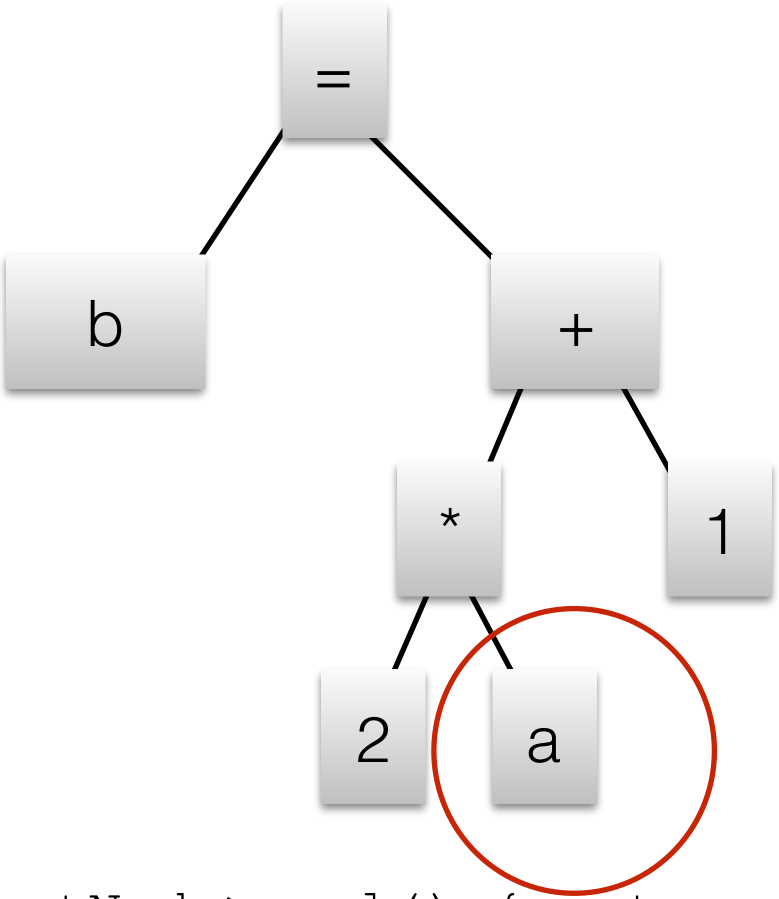


```
ConstNode>eval() { return val; }
AddNode>eval() { return left.eval()+right.eval(); }
MulNode>eval() { return left.eval()*right.eval(); }
VarNode>eval() { return val; }
AssignNode>eval() { return var.set(rhsVal.eval()); }
```

Compiling a simple expression

b=2*a+1

```
b.var.set(            2 * a.val +()            )
```

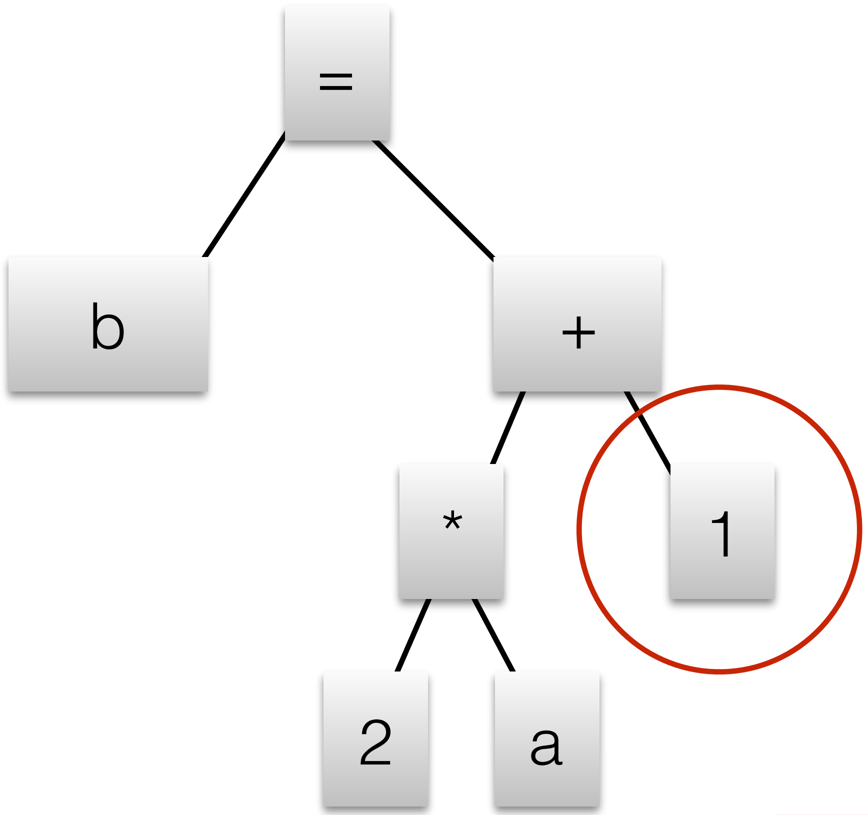


```
ConstNode>eval() { return val; }
AddNode>eval() { return left.eval()+right.eval(); }
MulNode>eval() { return left.eval()*right.eval(); }
VarNode>eval() { return val; }
AssignNode>eval() { return var.set(rhsVal.eval()); }
```

Compiling a simple expression

b=2*a+1

```
b.var.set(          2 * a.val + 1           )
```



```
ConstNode>eval() { return val; }
AddNode>eval() { return left.eval()+right.eval(); }
MulNode>eval() { return left.eval()*right.eval(); }
VarNode>eval() { return val; }
AssignNode>eval() { return var.set(rhsVal.eval()); }
```

Partially evaluation of the interpreter *is* compilation

- So one way to achieve language-independent compilation is to write a language interpreter and a partial evaluator for the language in which the *interpreter* is written

- To compile a different language, we just need a new interpreter, but not a new partial evaluator.

- Partial evaluation blends interpretation and compilation.

Partial evaluation alone is not enough

- Partial evaluation in this way has been known about for a long time [Futamura 71], but it hasn’t helped in implementing dynamic languages efficiently. Why?

- The problem is the lack of type and other behavioral information, which only becomes manifest at run time.

What is needed to generate ***good*** code for a user program?

- Express semantics of each language element
- Express what the user program is doing/likely to do in concrete terms (hotspots, types)
- Combine the semantics of each element of the user program with the usage information and generate code for the expected behavior.

What is needed to generate ***good*** code for a user program?

- Express semantics of each language element
    - ✓AST interpreter provides this directly
- Express what the user program is doing/likely to do in concrete terms (hotspots, types)
- Combine the semantics of each element of the user program with the usage information and generate code for the expected behavior.

What is needed to generate ***good*** code for a user program?

- Express semantics of each language element
  - ✓AST interpreter provides this directly
- Express what the user program is doing/likely to do in concrete terms (hotspots, types)
  - ✓Profile and specialize within the AST
- Combine the semantics of each element of the user program with the usage information and generate code for the expected behavior.

What is needed to generate ***good*** code for a user program?

- Express semantics of each language element
  - ✓AST interpreter provides this directly
- Express what the user program is doing/likely to do in concrete terms (hotspots, types)
  - ✓Profile and specialize within the AST
- Combine the semantics of each element of the user program with the usage information and generate code for the expected behavior.
  - ✓Use the interpreter and the profiles to generate specialized code. Deopt when wrong.

Specializing ASTs during interpretation

- One solution is to gather profile data during AST interpretation.

- But to get faster interpretation *and* profiling data, we can specialize the AST nodes at run-time based on each node's observed behavior.

Example: addition

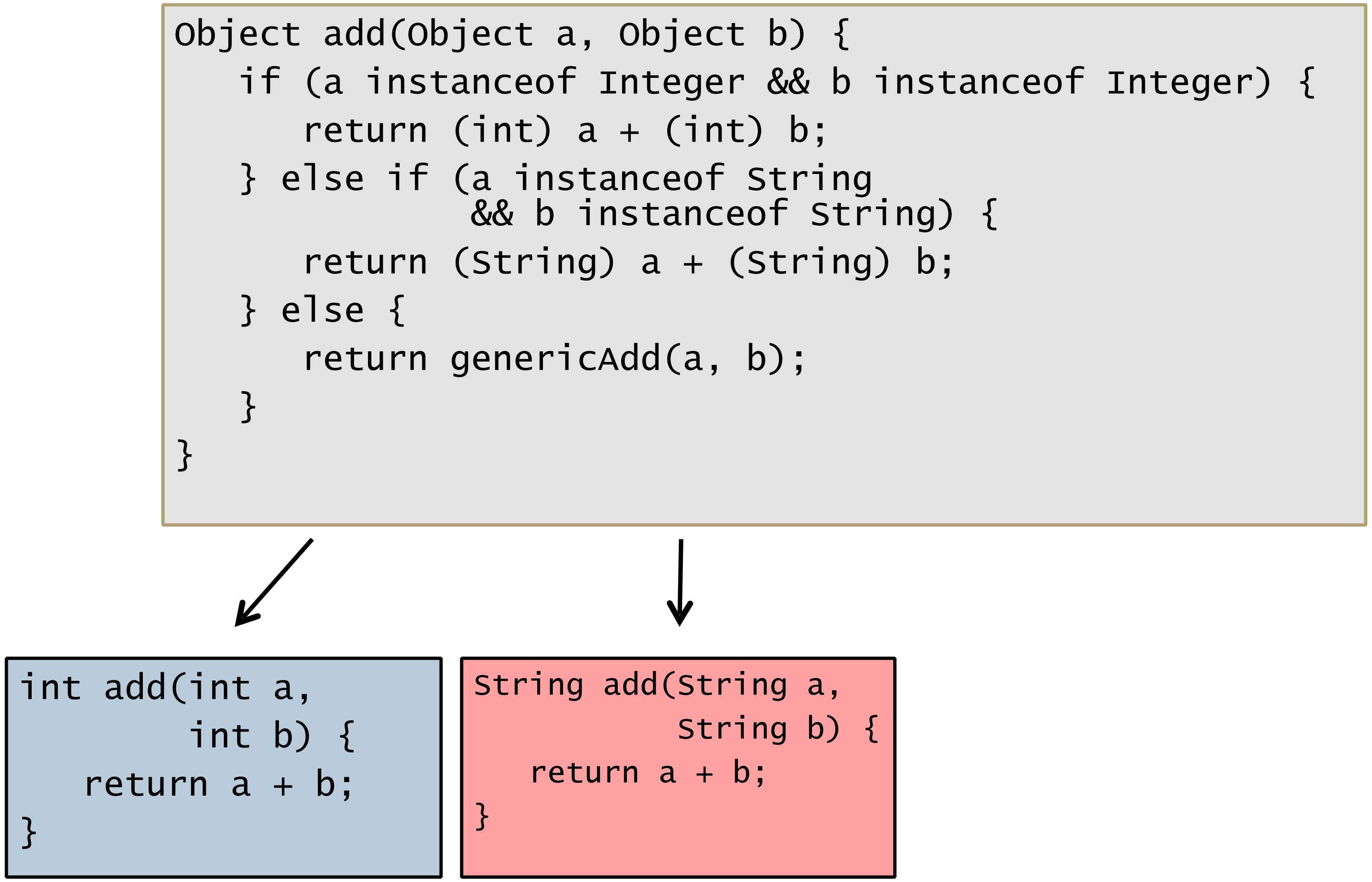
```
Object add(Object a, Object b) {
   if (a instanceof Integer && b instanceof Integer) {
      return (int) a + (int) b;
   } else if (a instanceof String
              && b instanceof String) {
      return (String) a + (String) b;
   } else {
      return genericAdd(a, b);
   }
}
```

Example: addition

```
Object add(Object a, Object b) {
   if (a instanceof Integer && b instanceof Integer) {
      return (int) a + (int) b;
   } else if (a instanceof String
              && b instanceof String) {
      return (String) a + (String) b;
   } else {
      return genericAdd(a, b);
   }
}
```

```
int add(int a,
        int b) {
   return a + b;
}
```

Example: addition

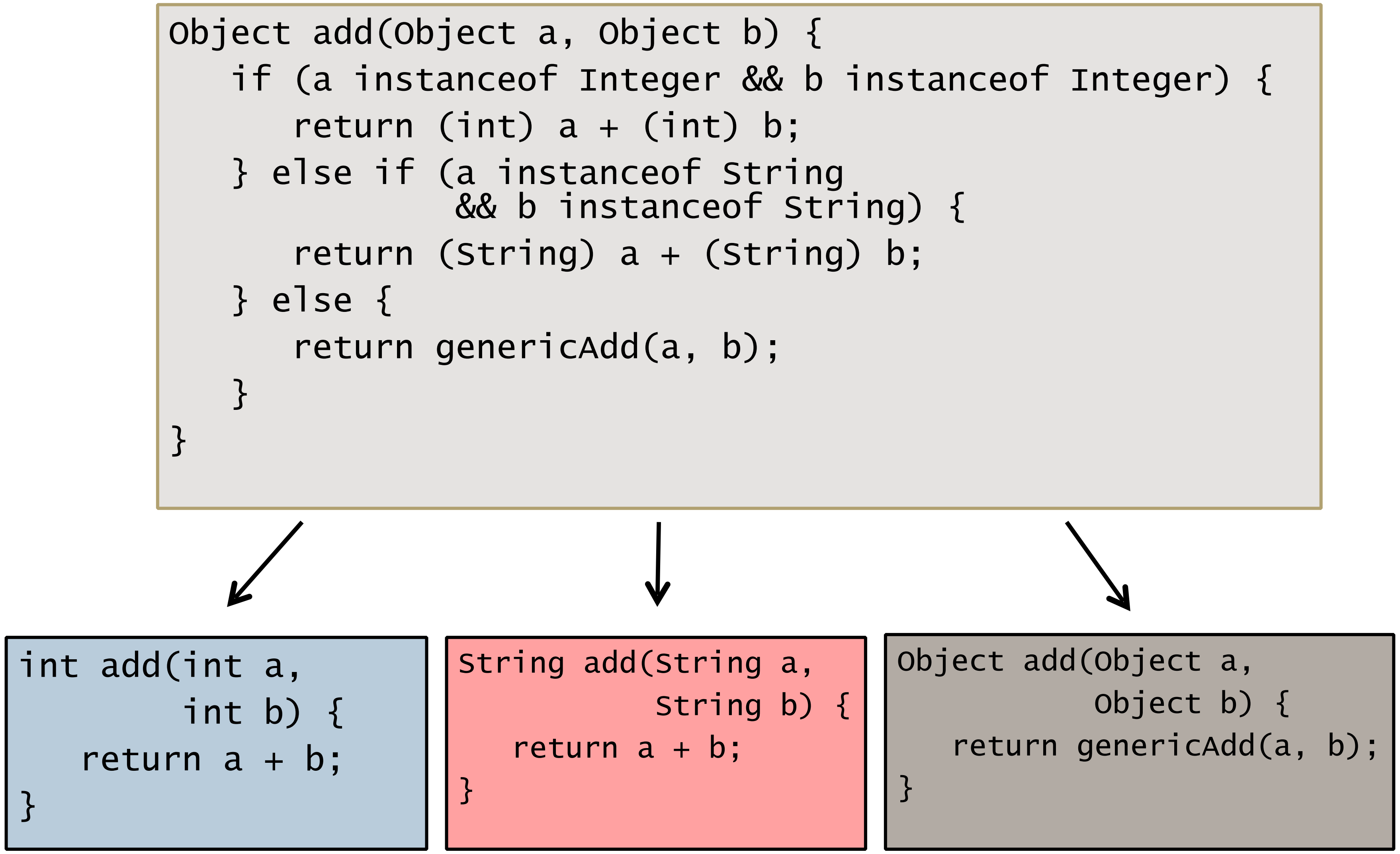


```
Object add(Object a, Object b) {
   if (a instanceof Integer && b instanceof Integer) {
      return (int) a + (int) b;
   } else if (a instanceof String
              && b instanceof String) {
      return (String) a + (String) b;
   } else {
      return genericAdd(a, b);
   }
}
```

```
int add(int a,
        int b) {
   return a + b;
}
```

```
String add(String a,
           String b) {
   return a + b;
}
```

Example: addition



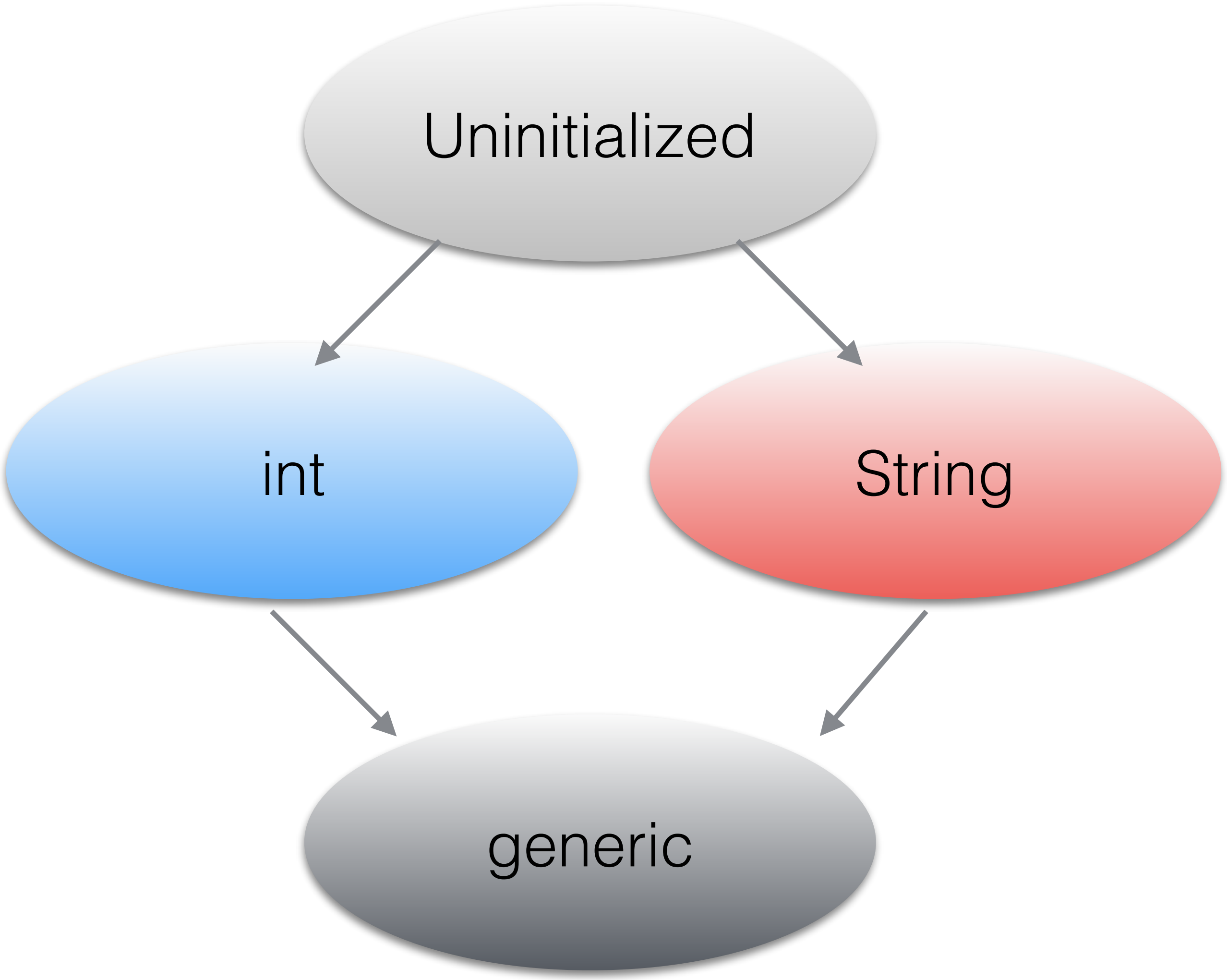
```
Object add(Object a, Object b) {
   if (a instanceof Integer && b instanceof Integer) {
      return (int) a + (int) b;
   } else if (a instanceof String
              && b instanceof String) {
      return (String) a + (String) b;
   } else {
      return genericAdd(a, b);
   }
}
```

```
int add(int a,
        int b) {
   return a + b;
}
```

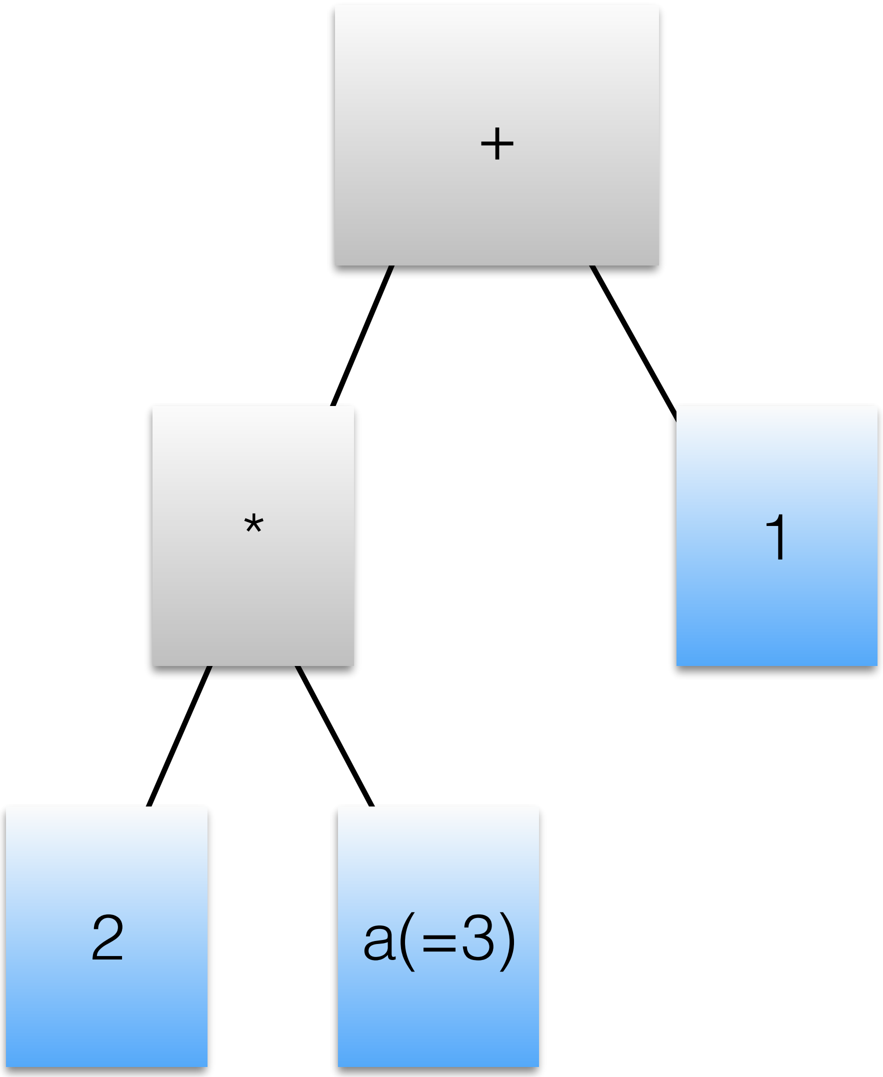
```
String add(String a,
           String b) {
   return a + b;
}
```

```
Object add(Object a,
           Object b) {
   return genericAdd(a, b);
}
```

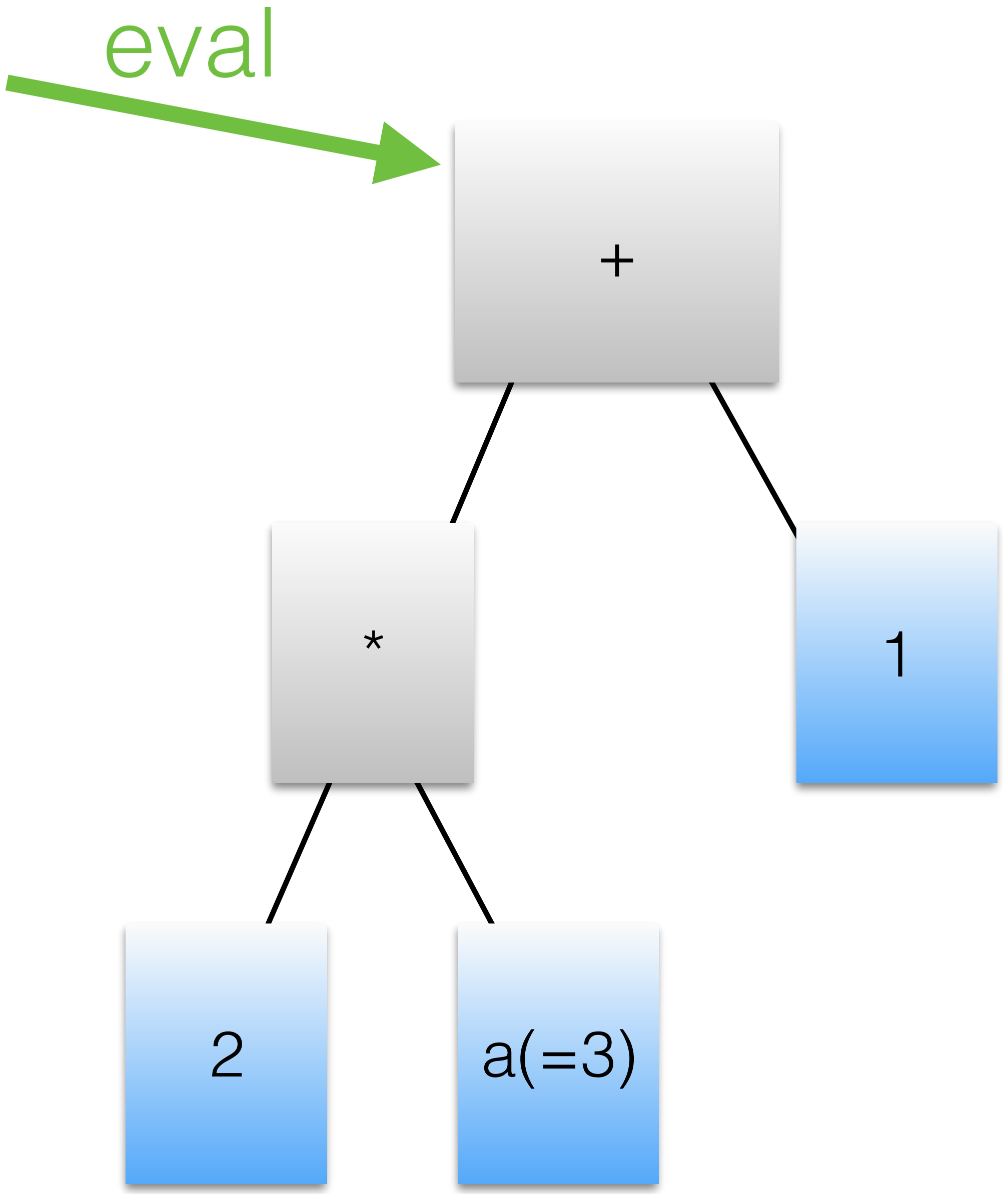
Type transitions



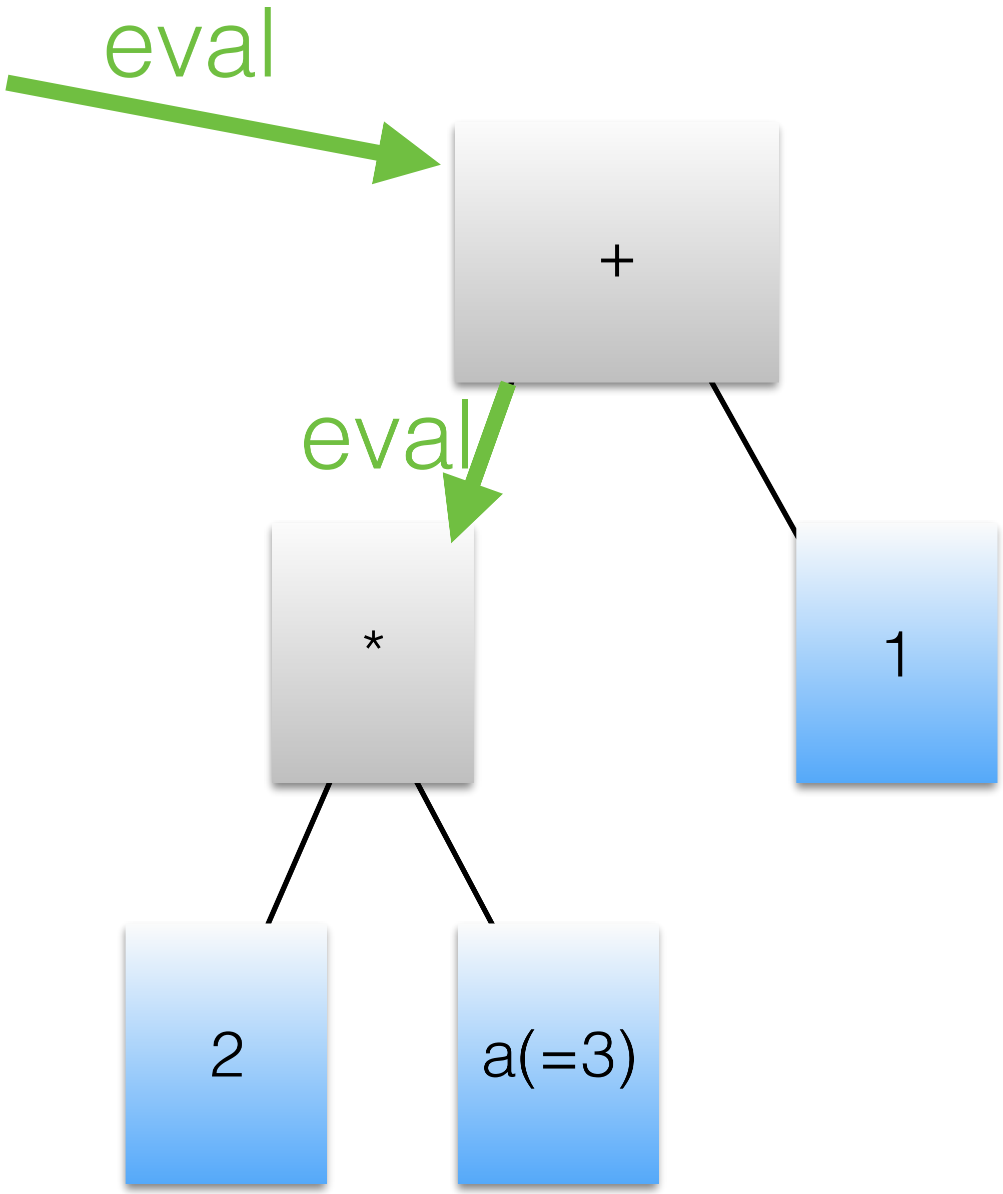
Evolution of an expression
1. Specialization



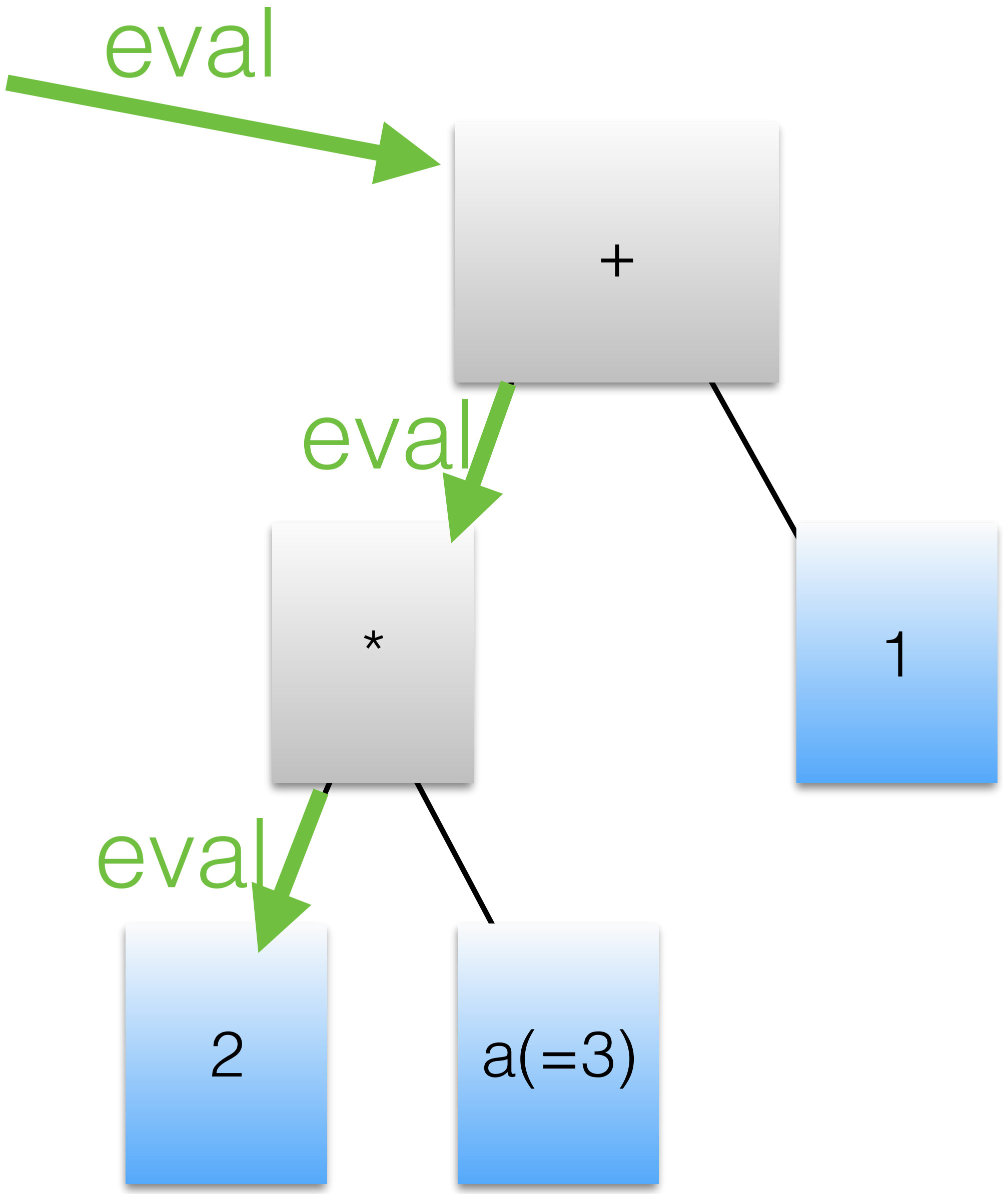
Evolution of an expression
1. Specialization



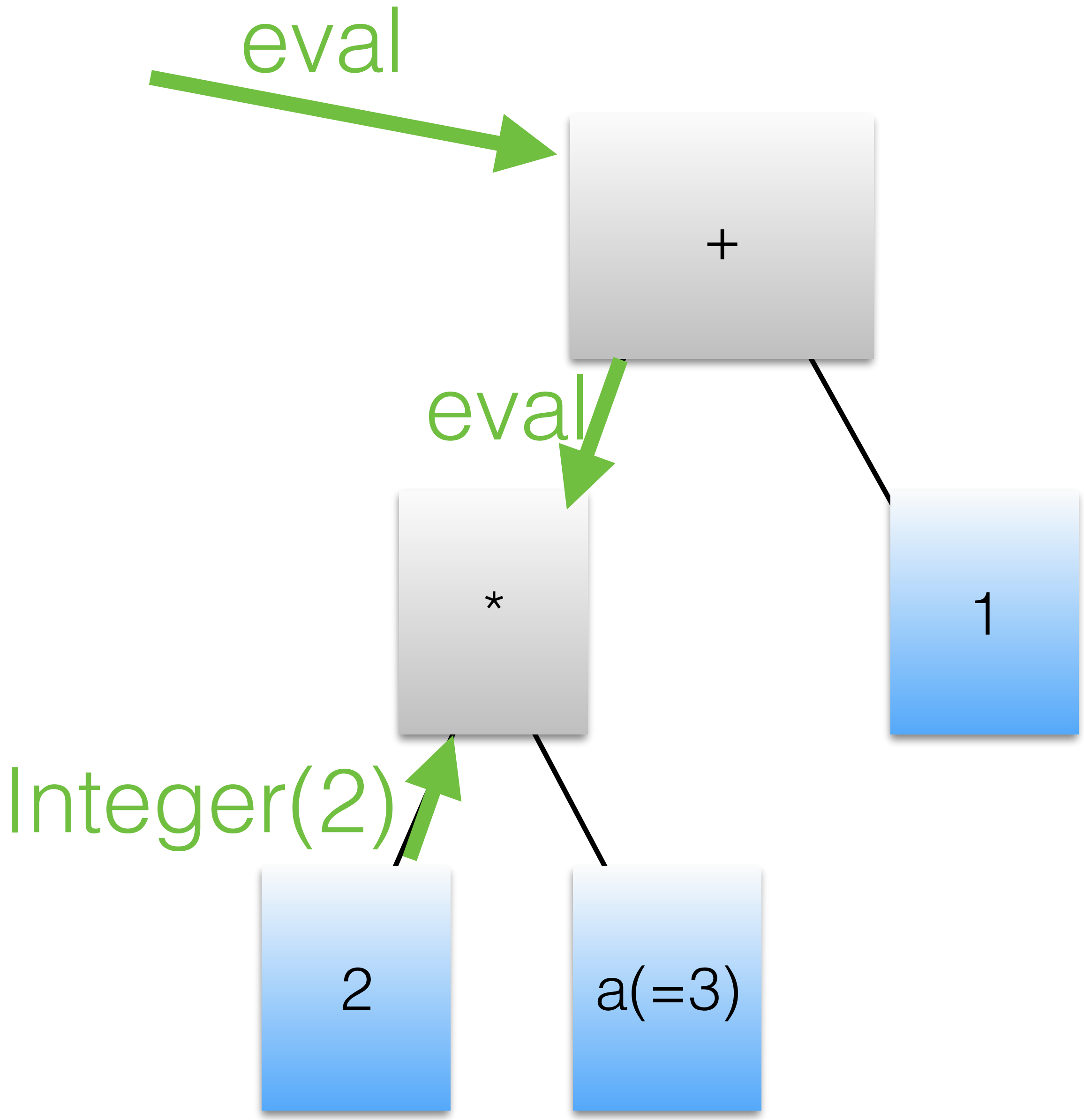
Evolution of an expression
1. Specialization



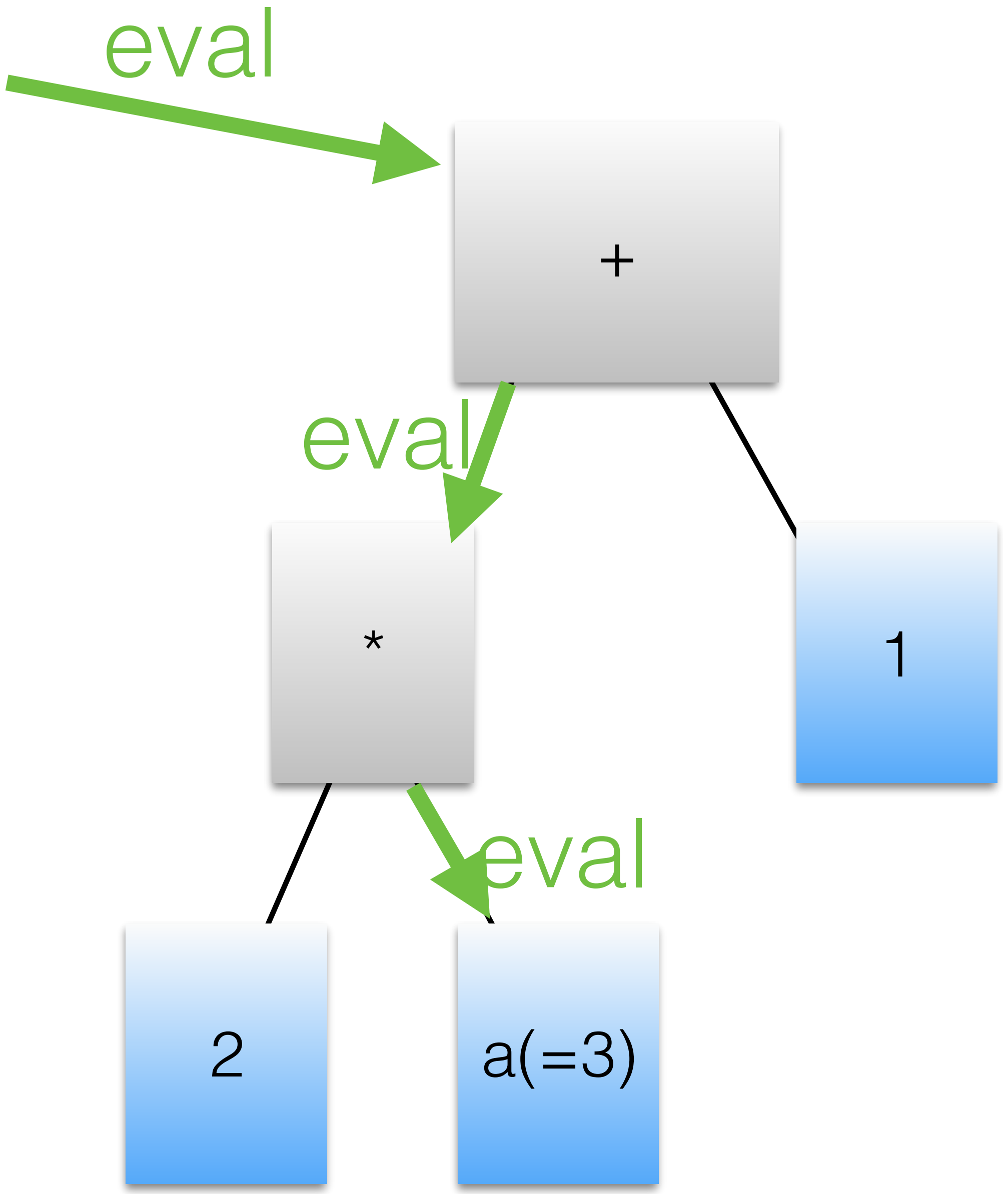
Evolution of an expression
1. Specialization



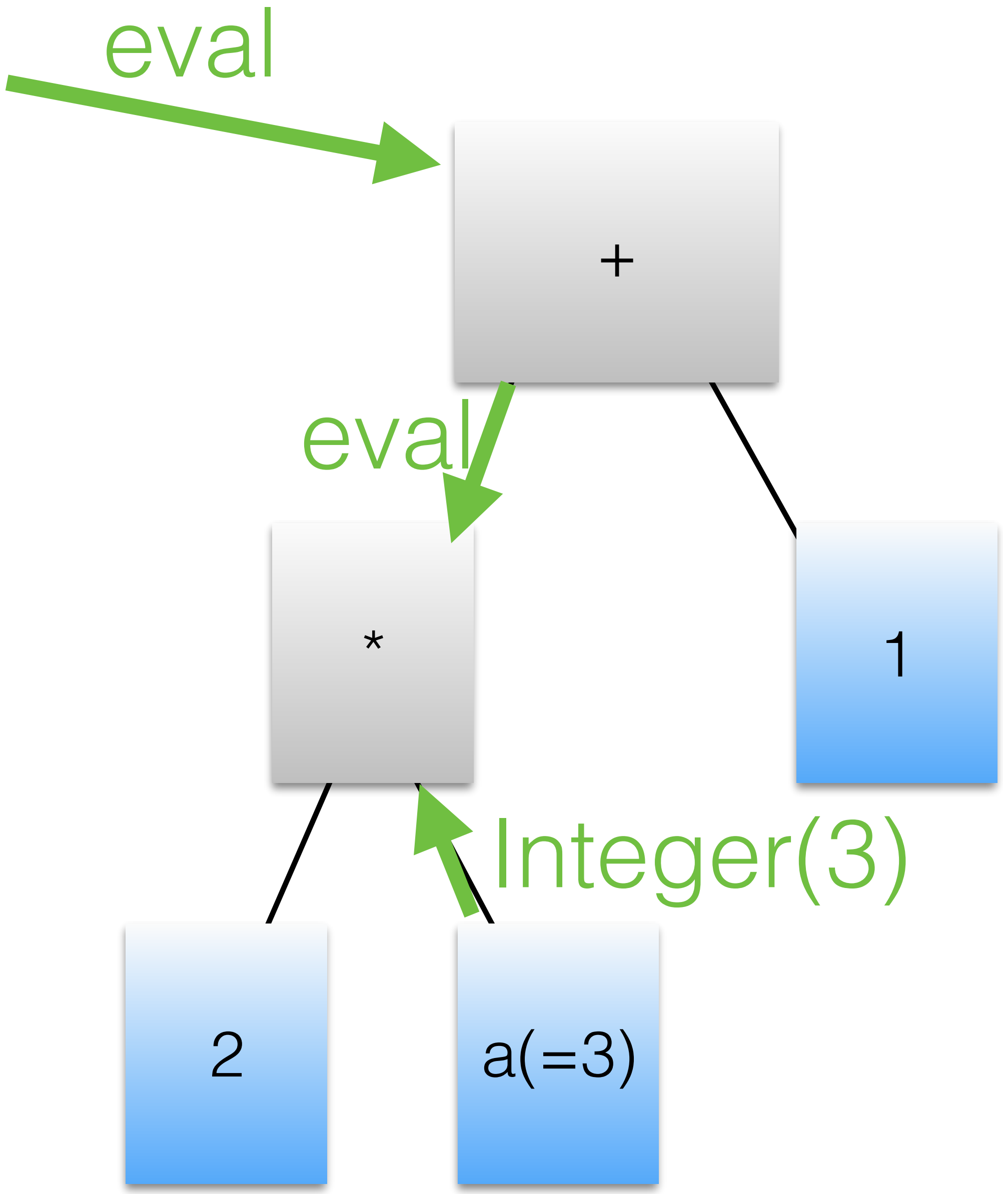
Evolution of an expression
1. Specialization



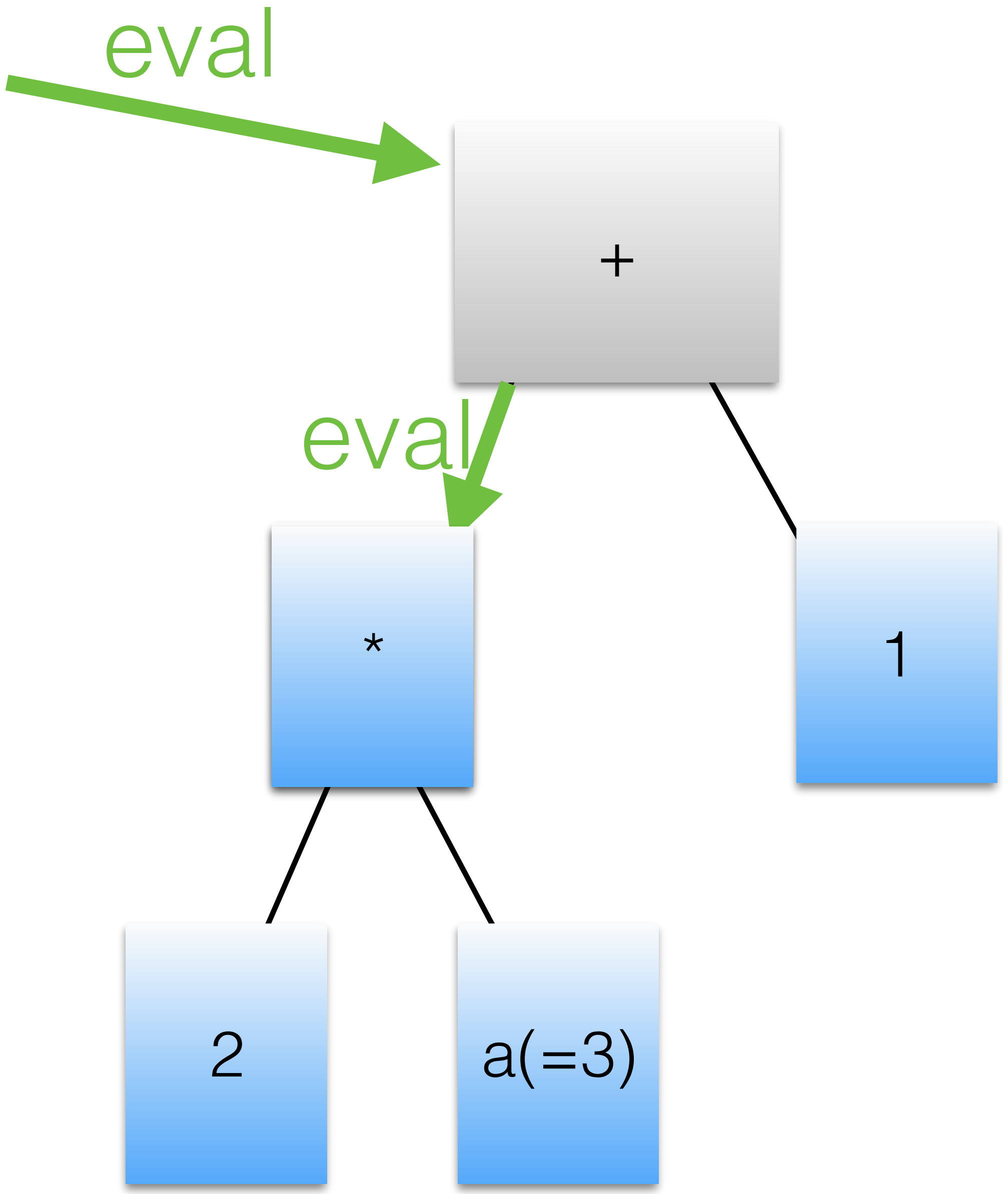
Evolution of an expression
1. Specialization



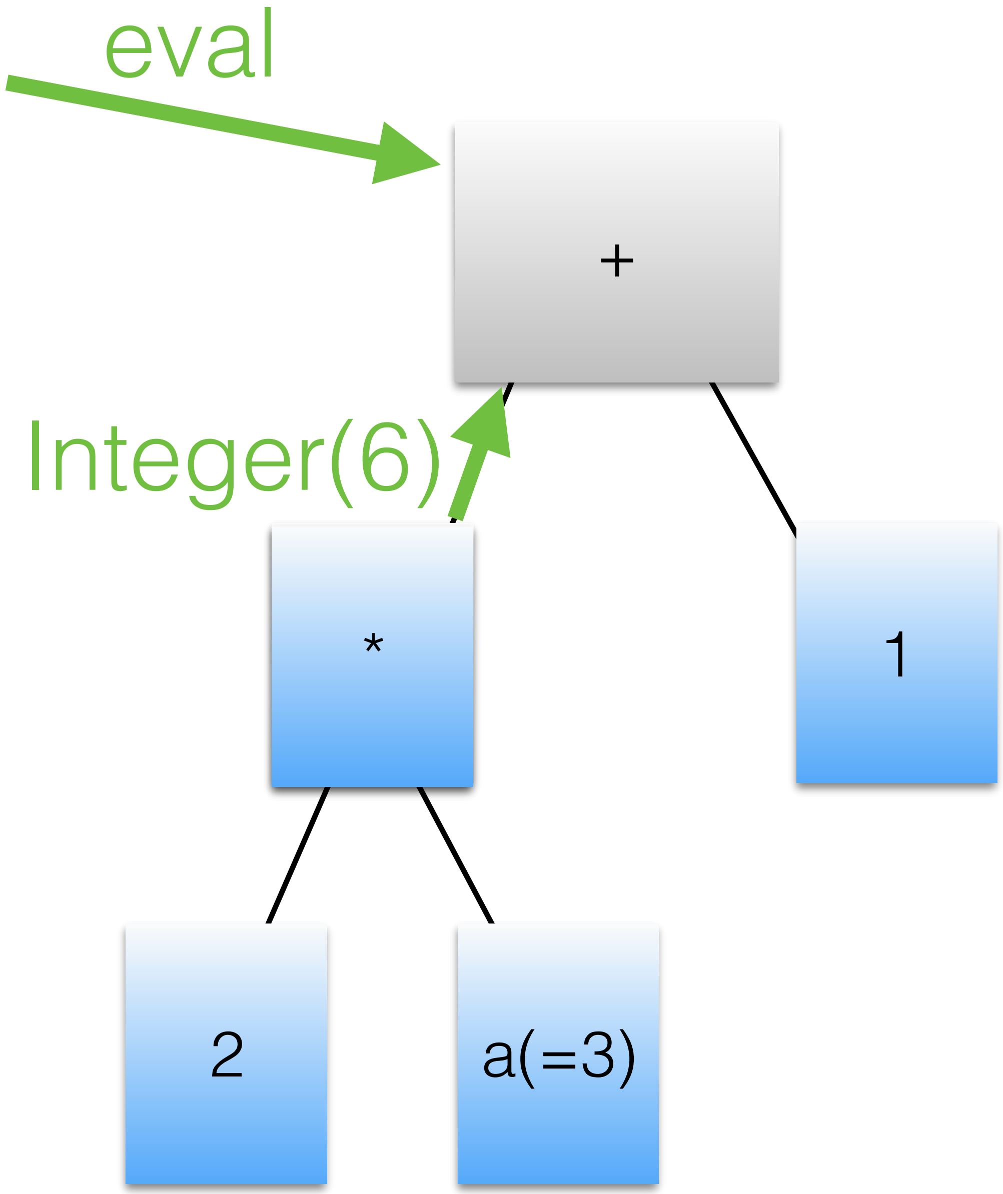
Evolution of an expression
1. Specialization



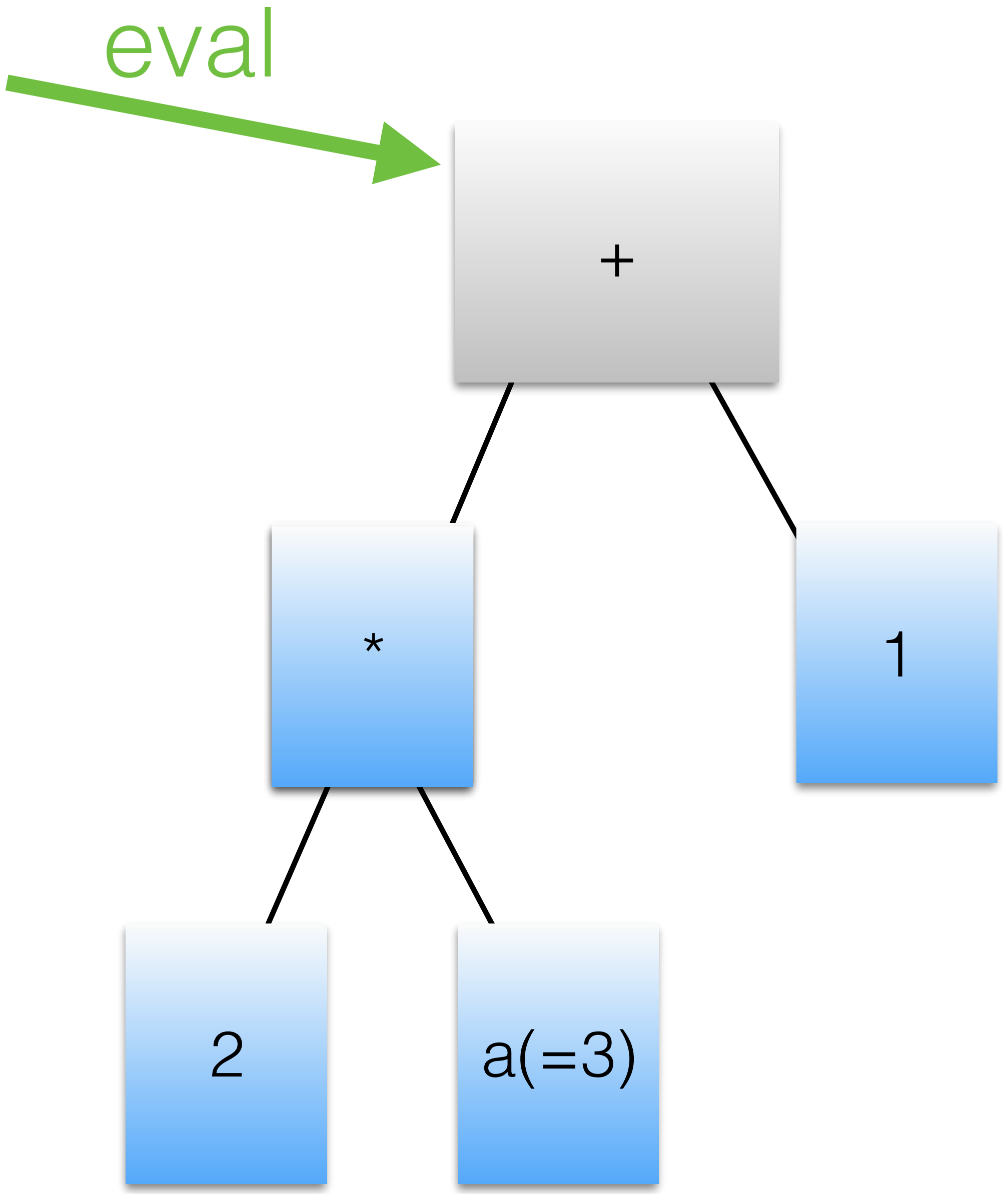
Evolution of an expression
1. Specialization



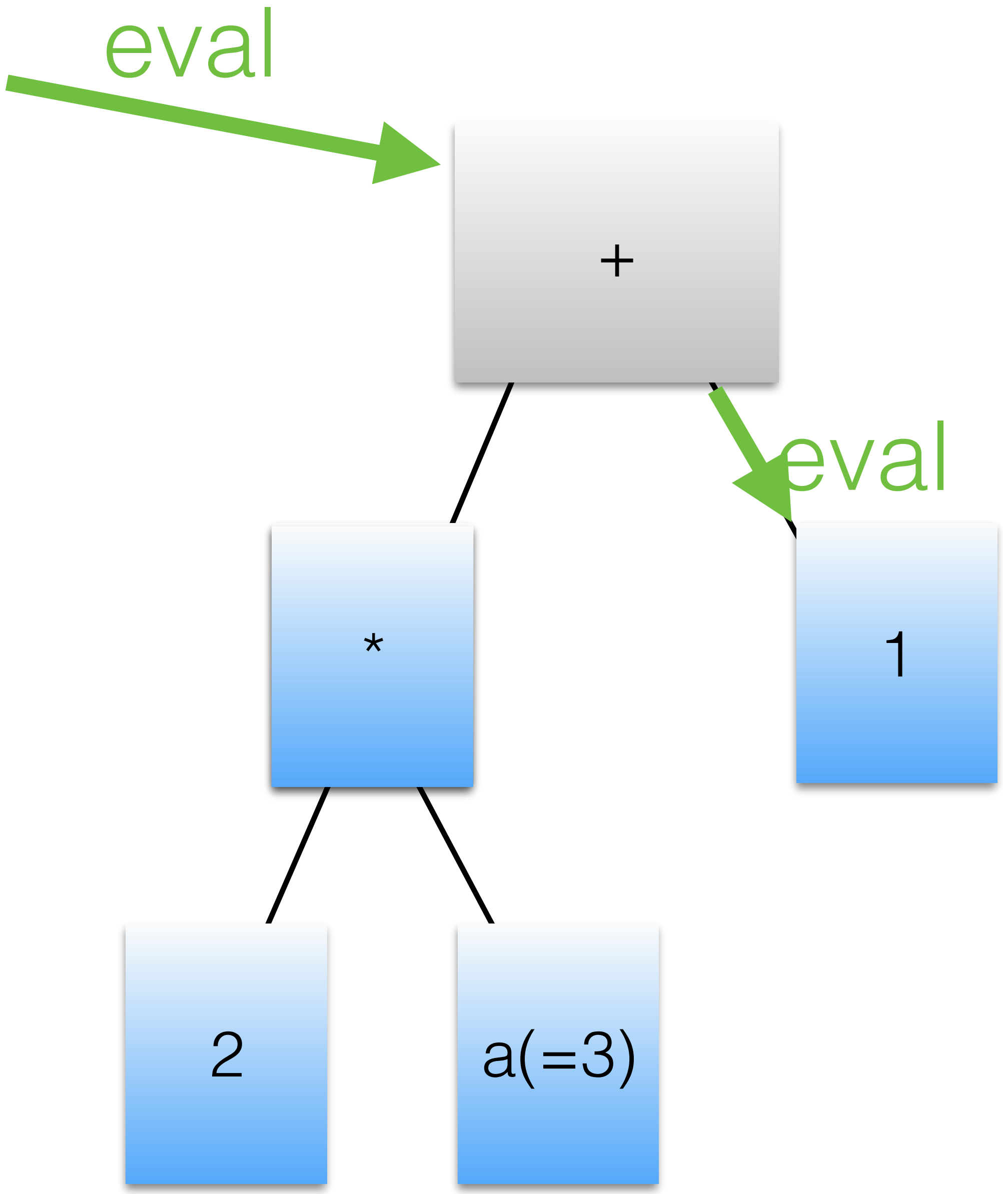
Evolution of an expression
1. Specialization



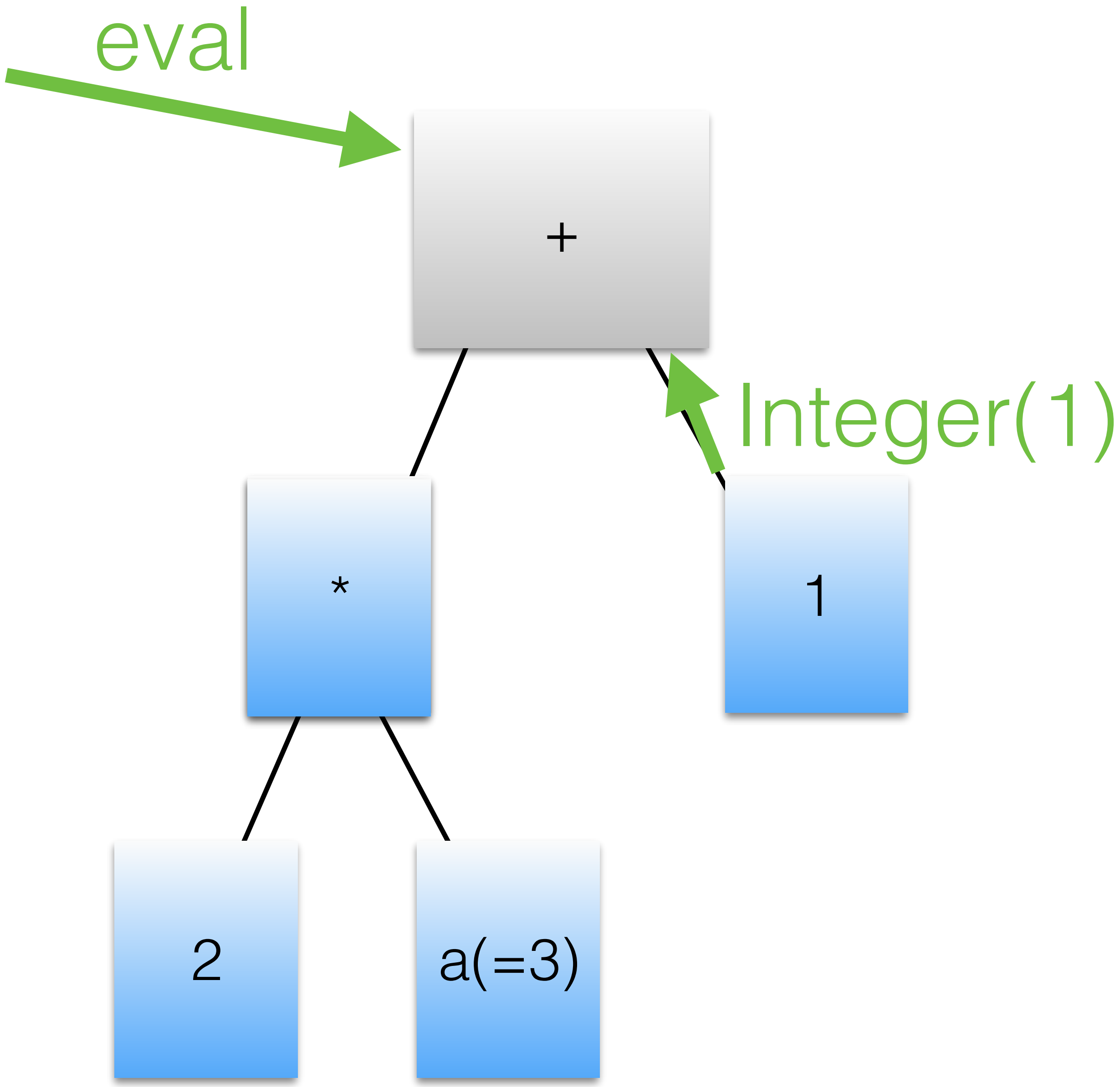
Evolution of an expression
1. Specialization



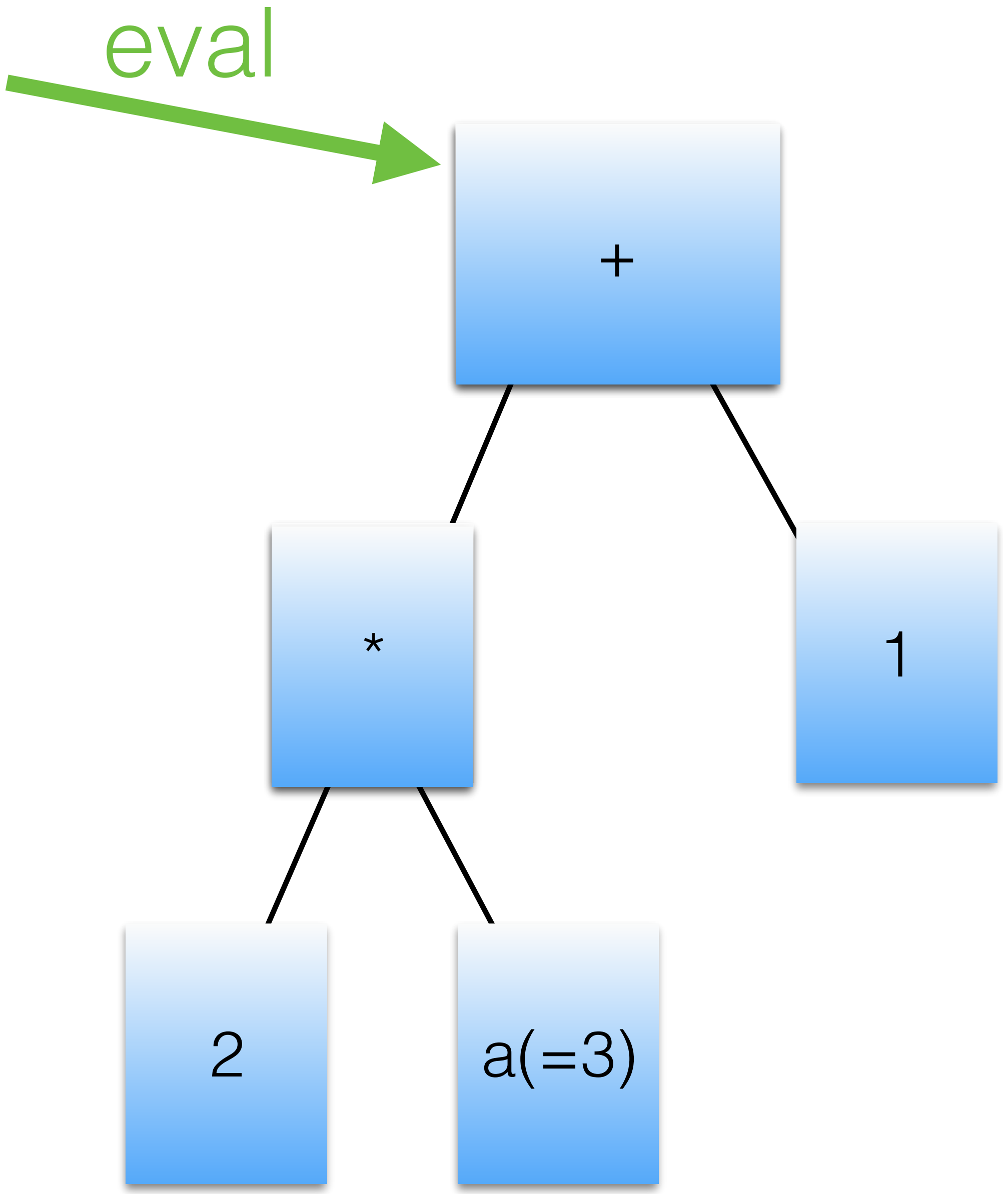
Evolution of an expression
1. Specialization



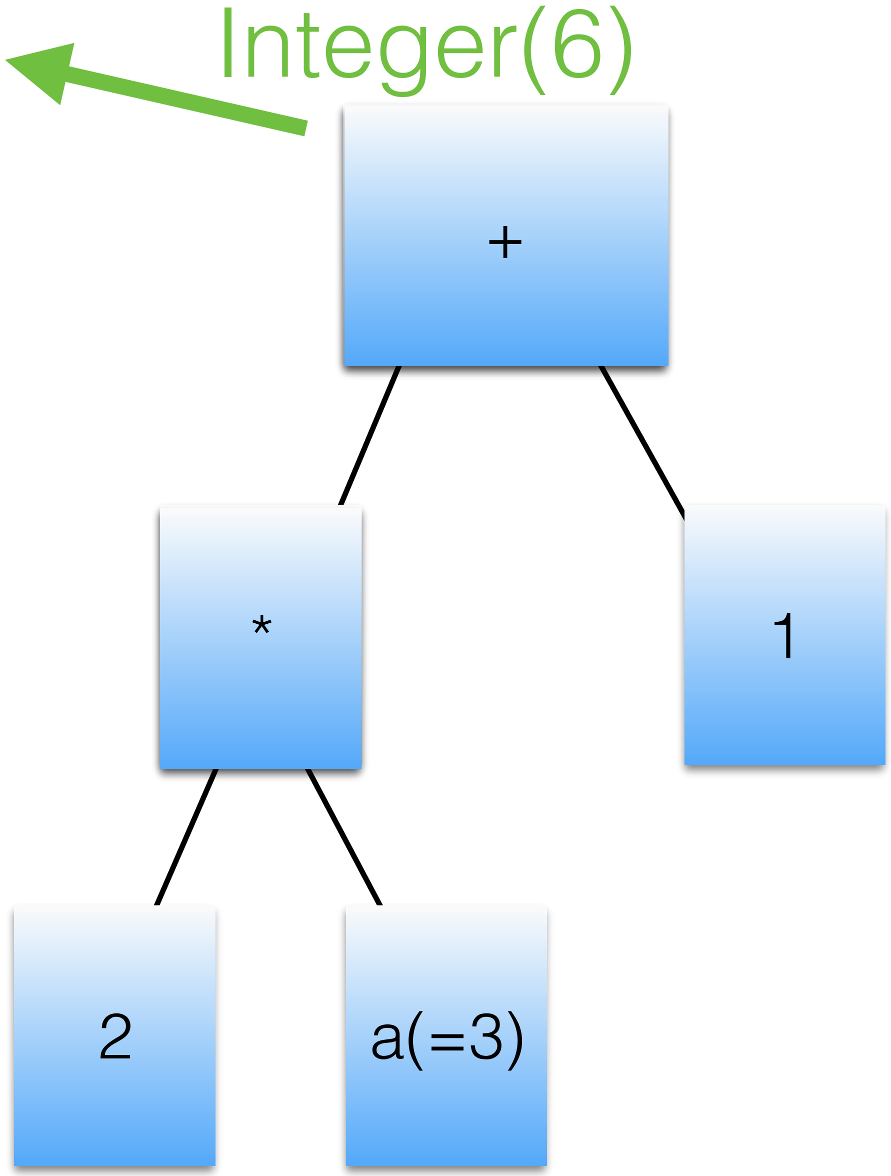
Evolution of an expression
1. Specialization



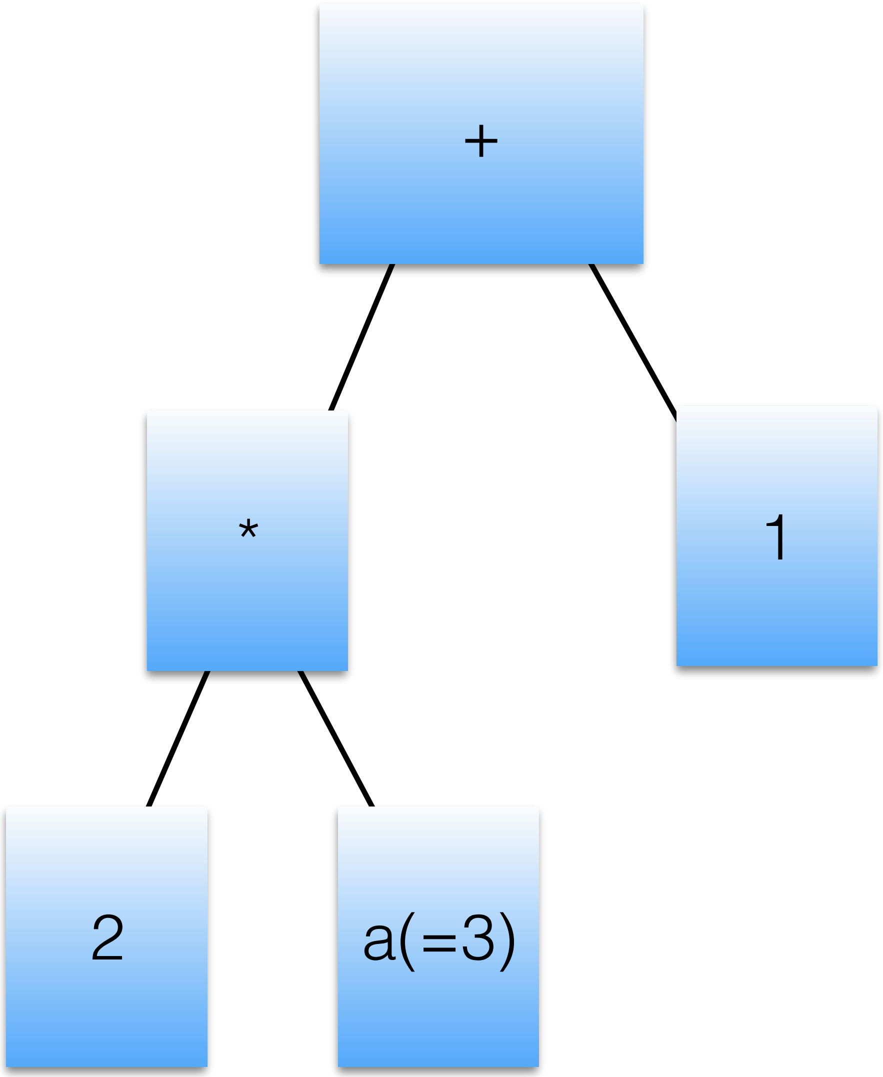
Evolution of an expression
1. Specialization



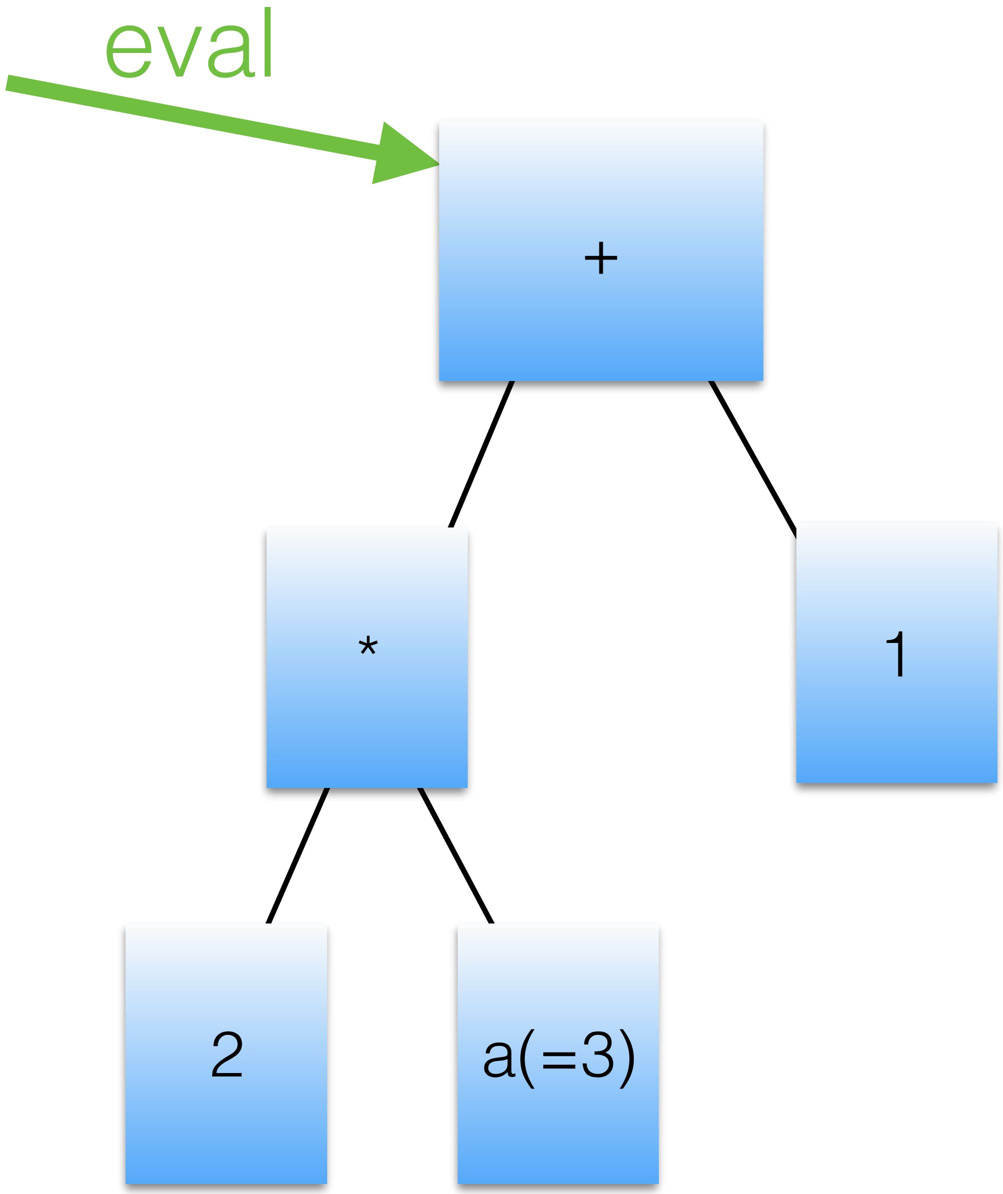
Evolution of an expression
1. Specialization



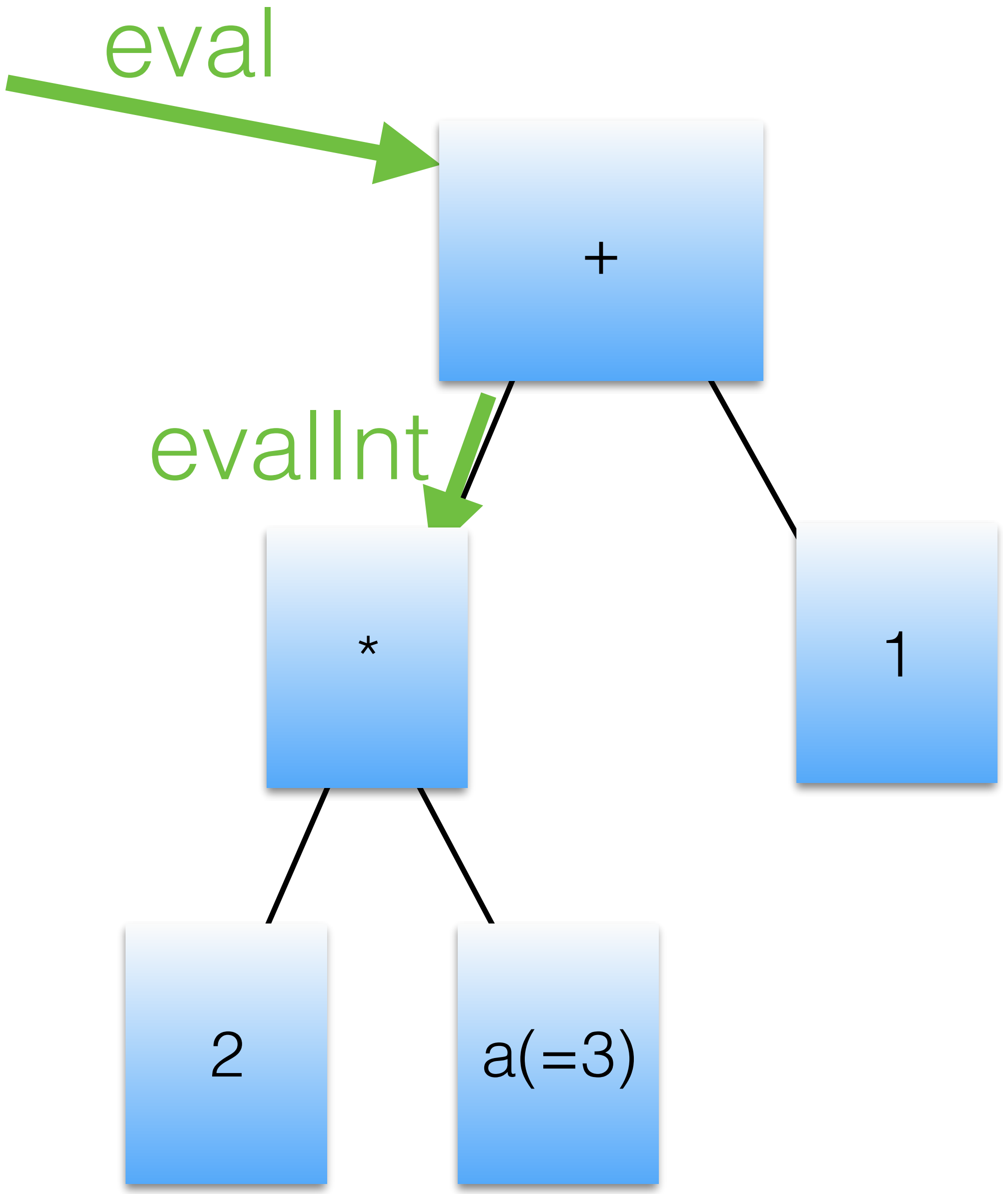
Evolution of an expression
2. Repeated execution



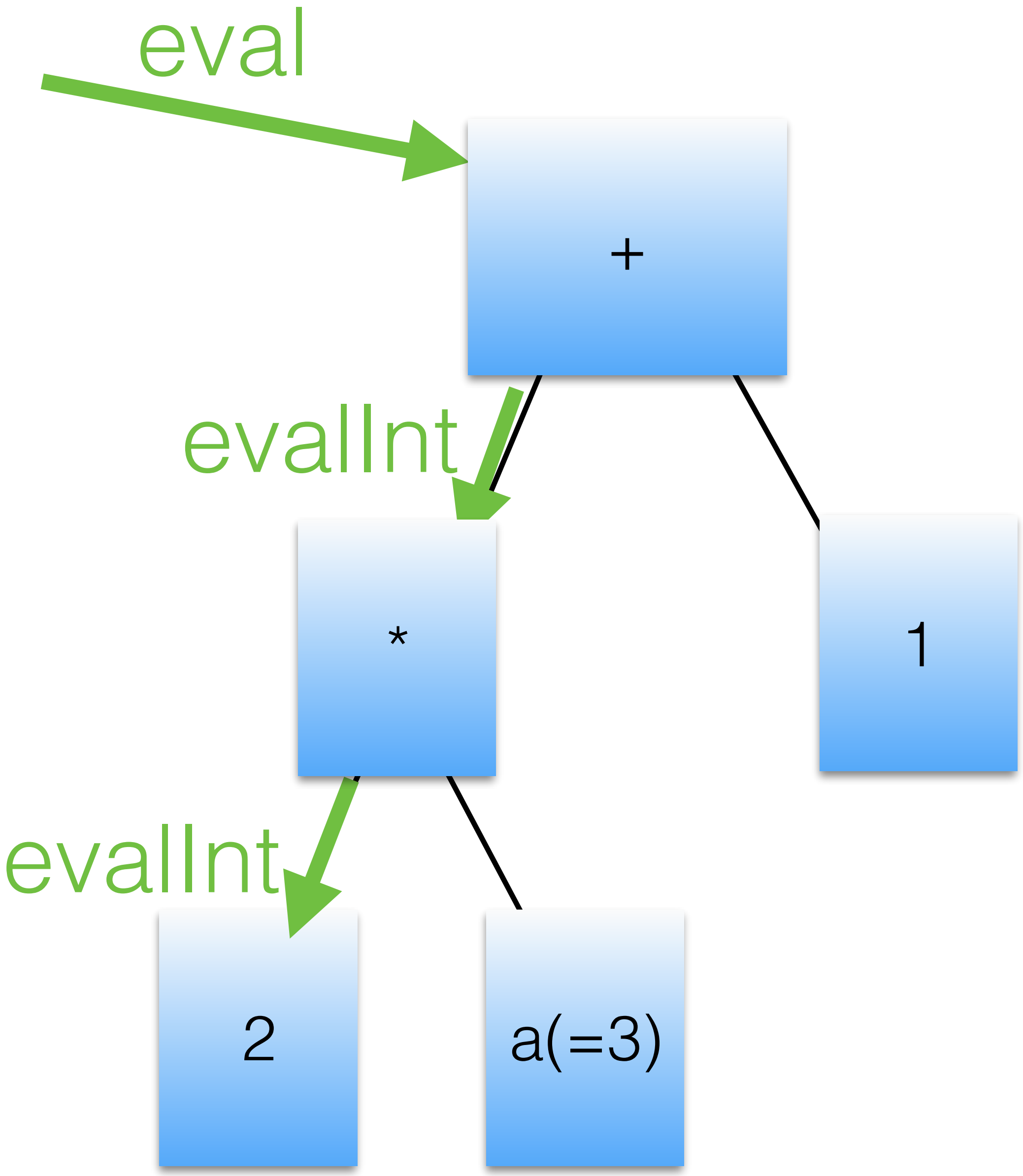
Evolution of an expression
2. Repeated execution



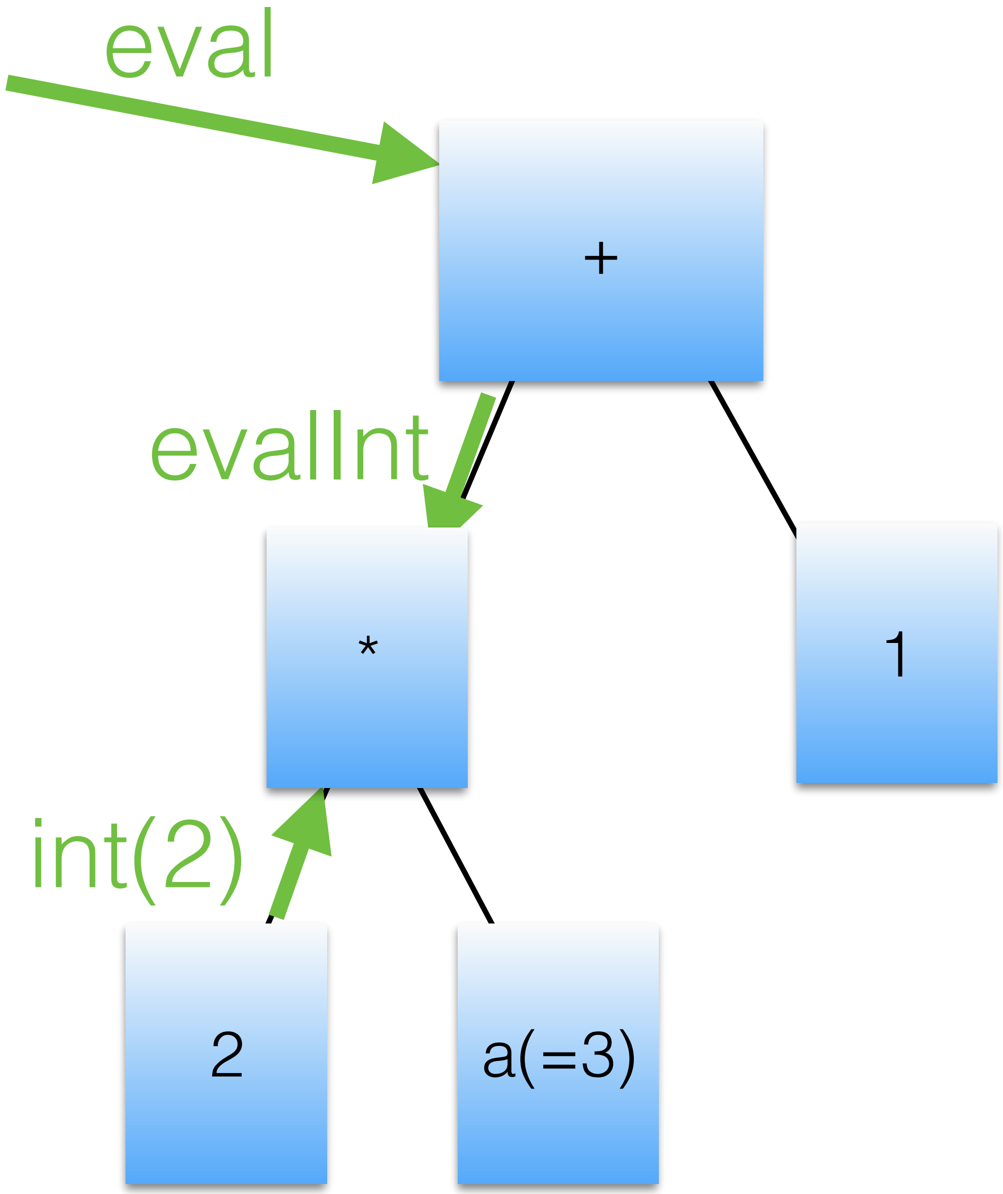
Evolution of an expression
2. Repeated execution



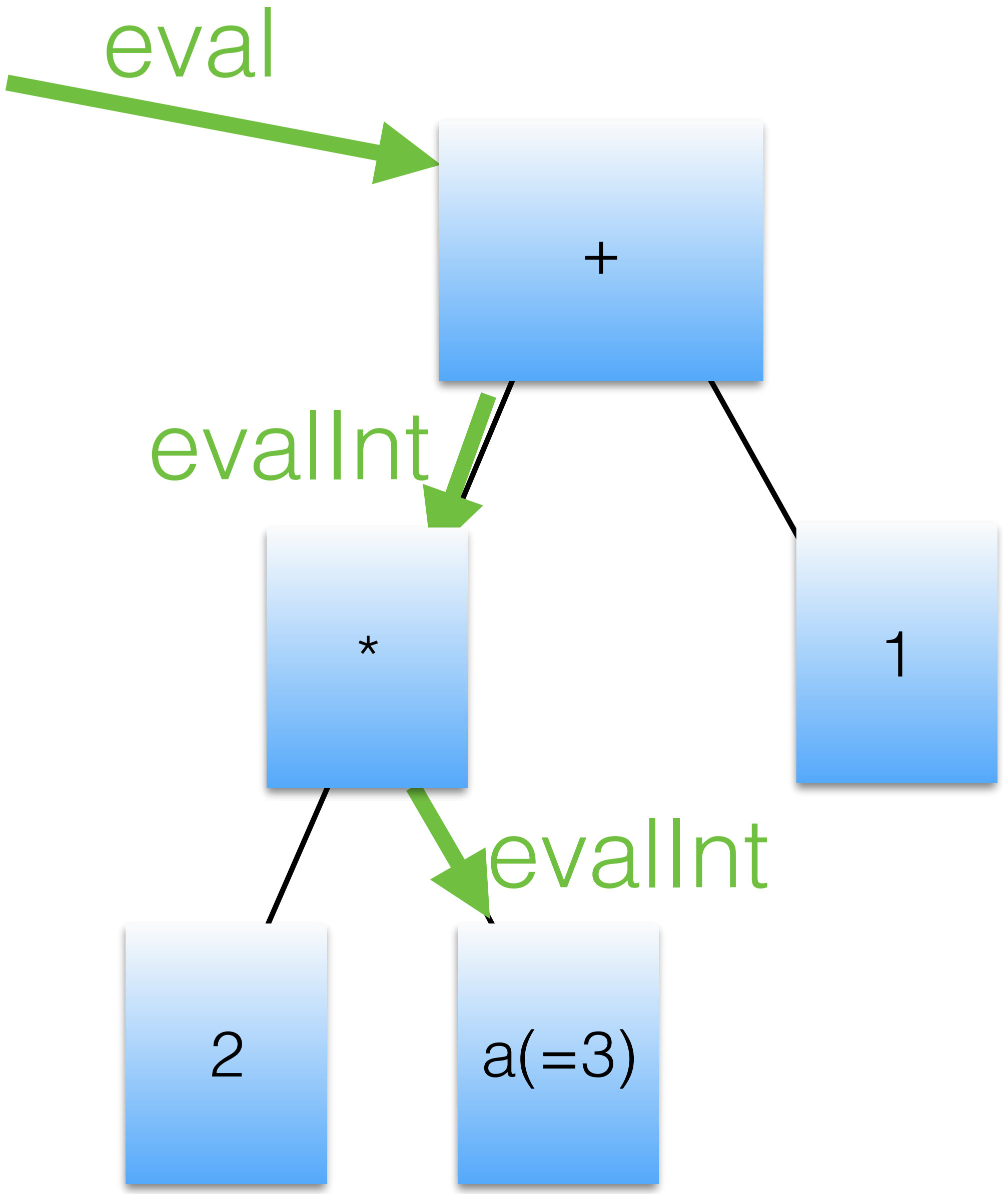
Evolution of an expression
2. Repeated execution



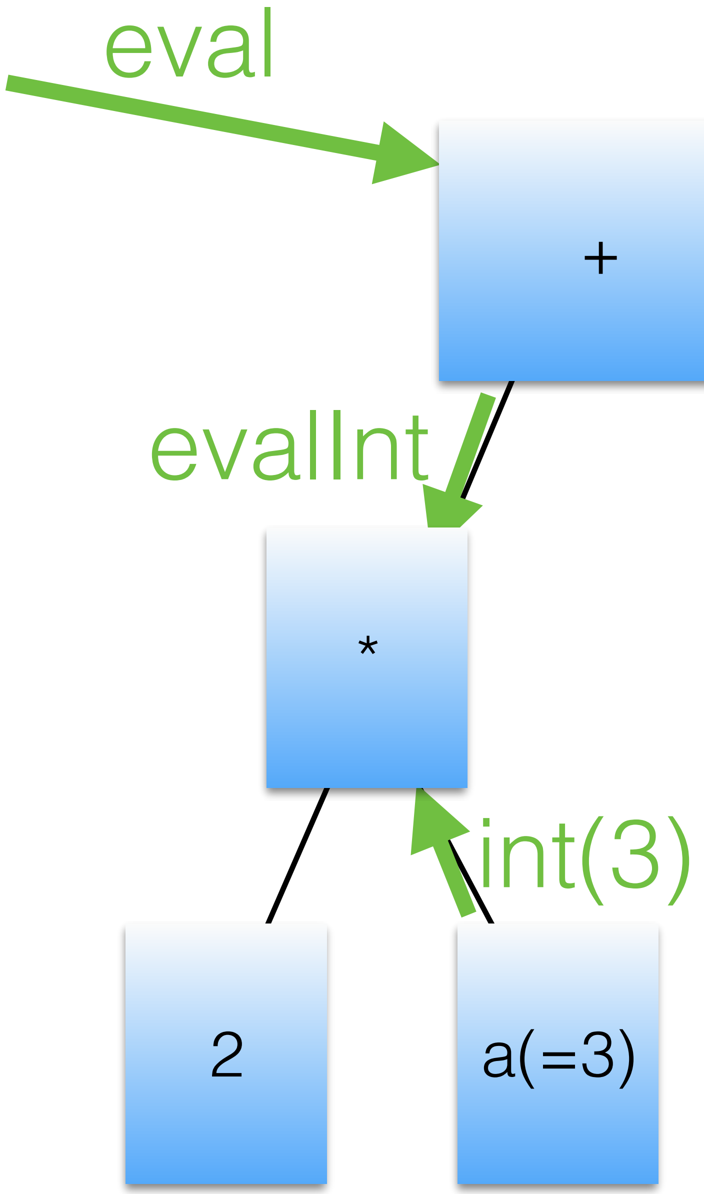
Evolution of an expression
2. Repeated execution



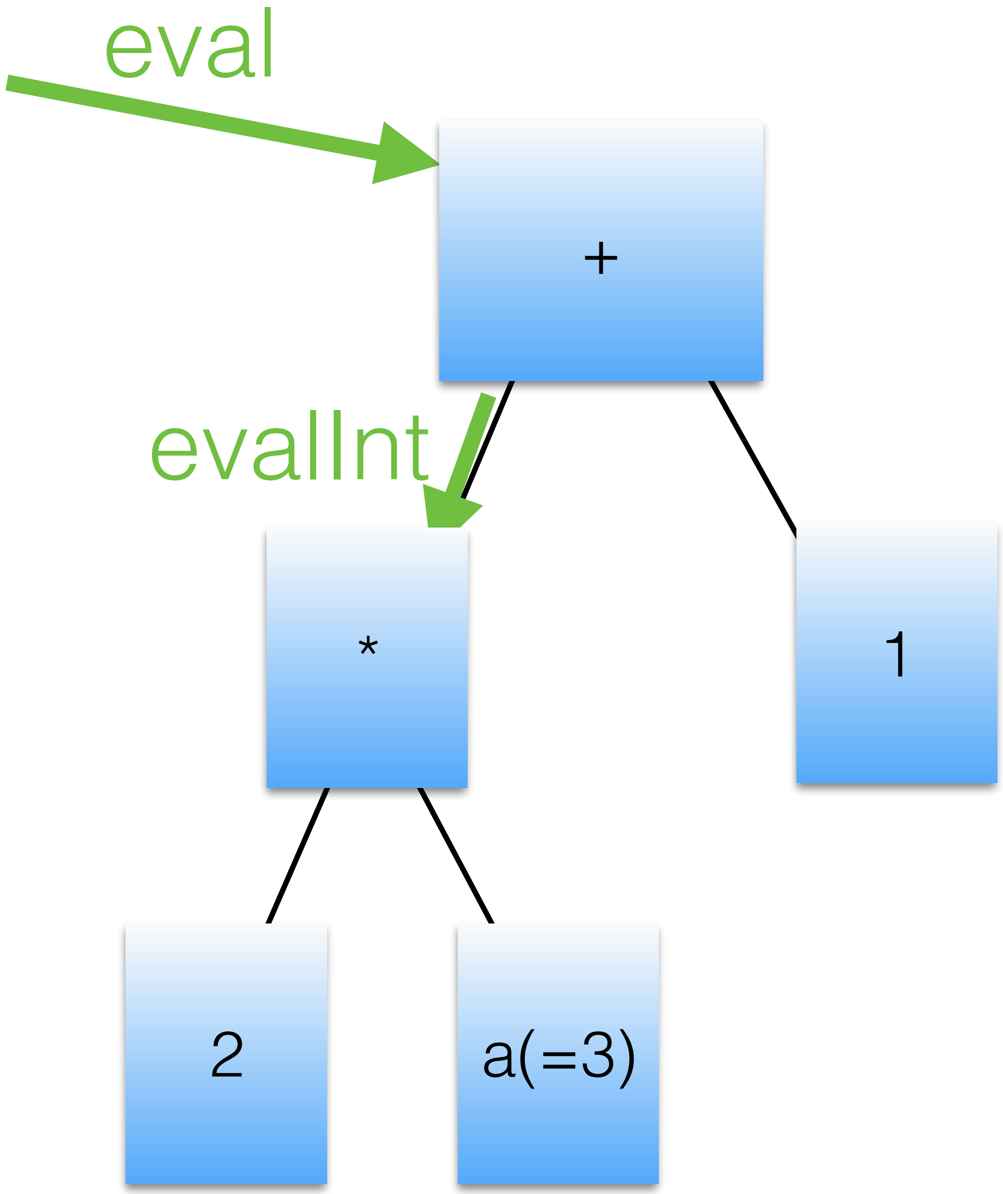
Evolution of an expression
2. Repeated execution



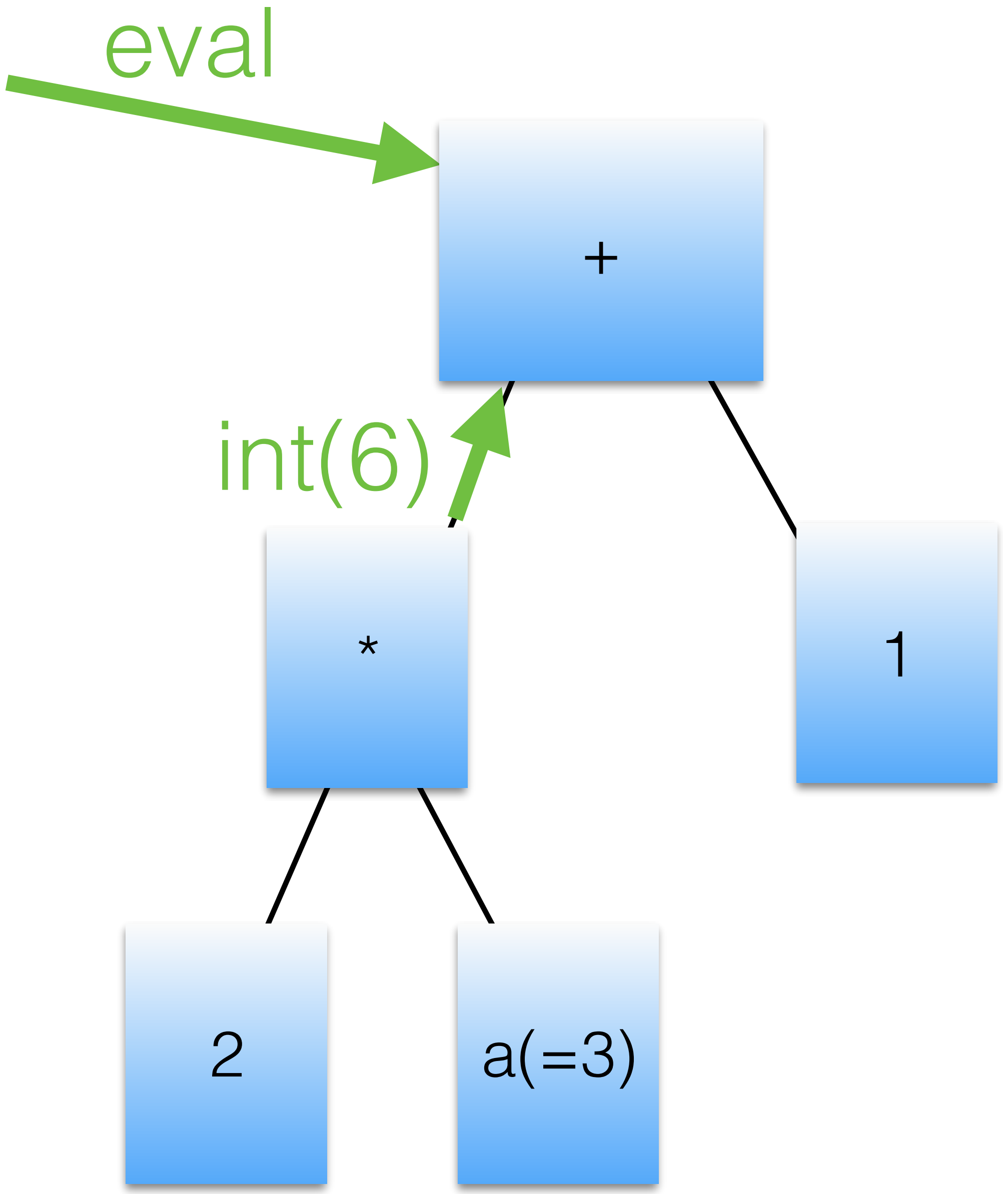
Evolution of an expression
2. Repeated execution



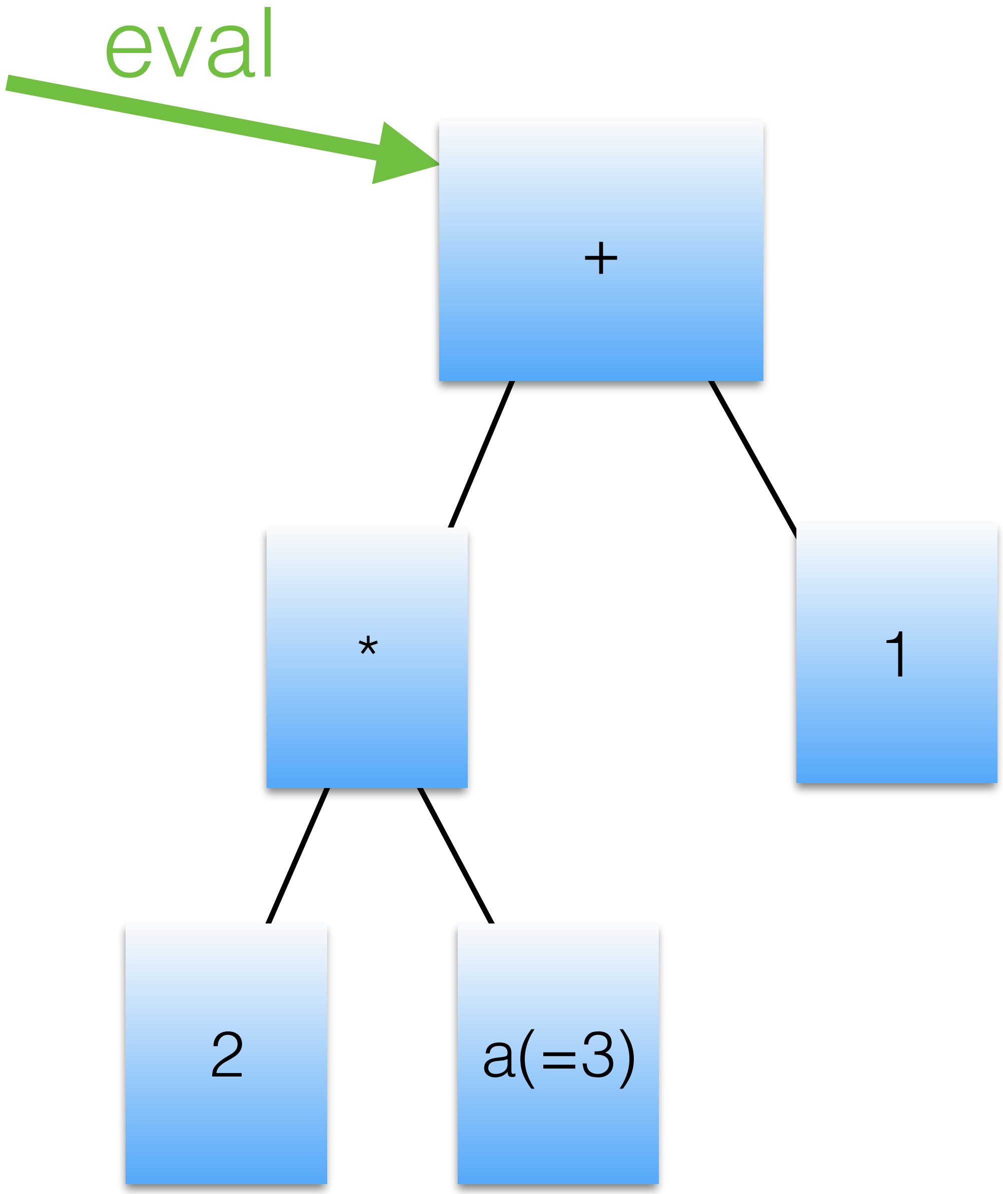
Evolution of an expression
2. Repeated execution



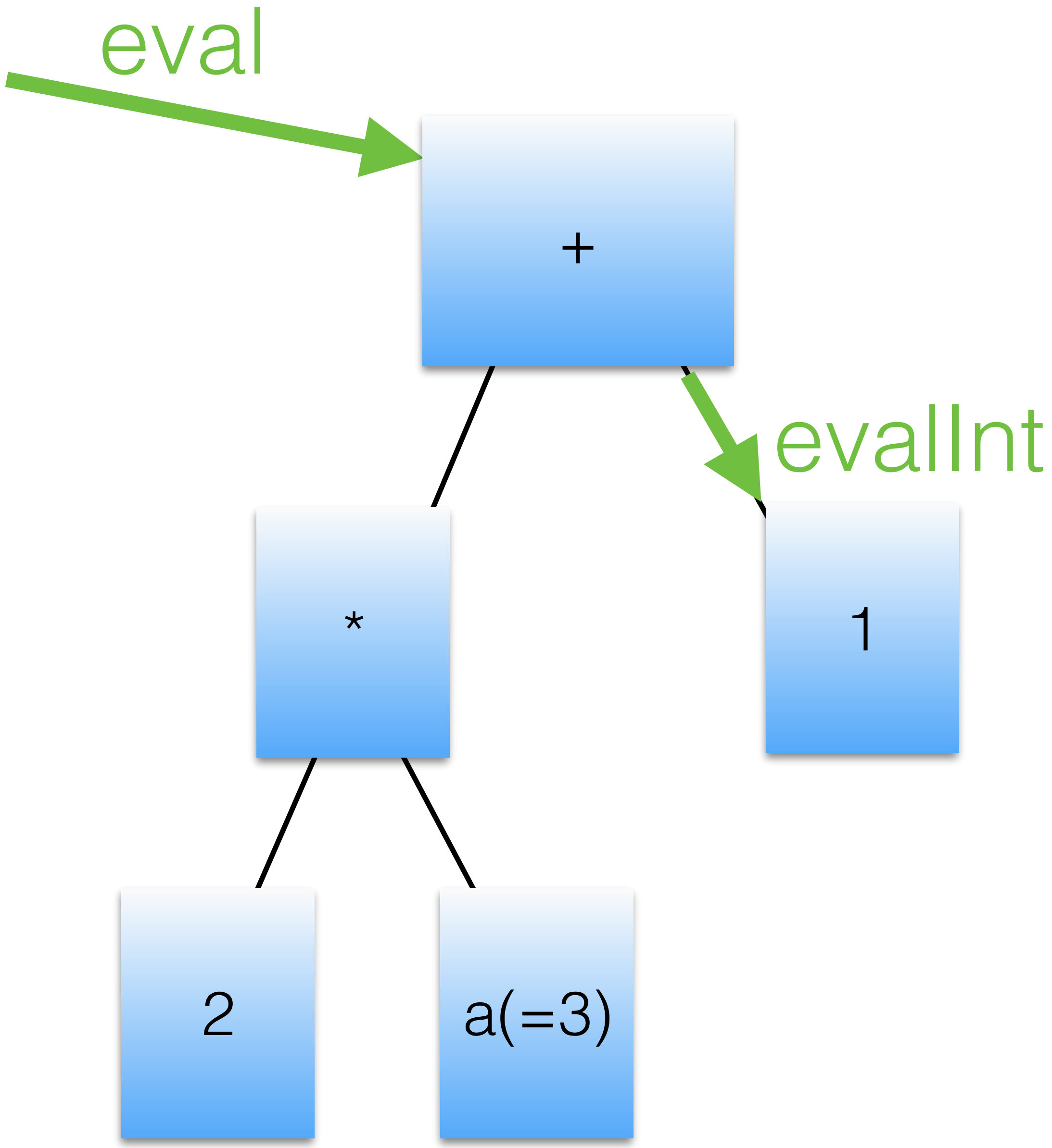
Evolution of an expression
2. Repeated execution



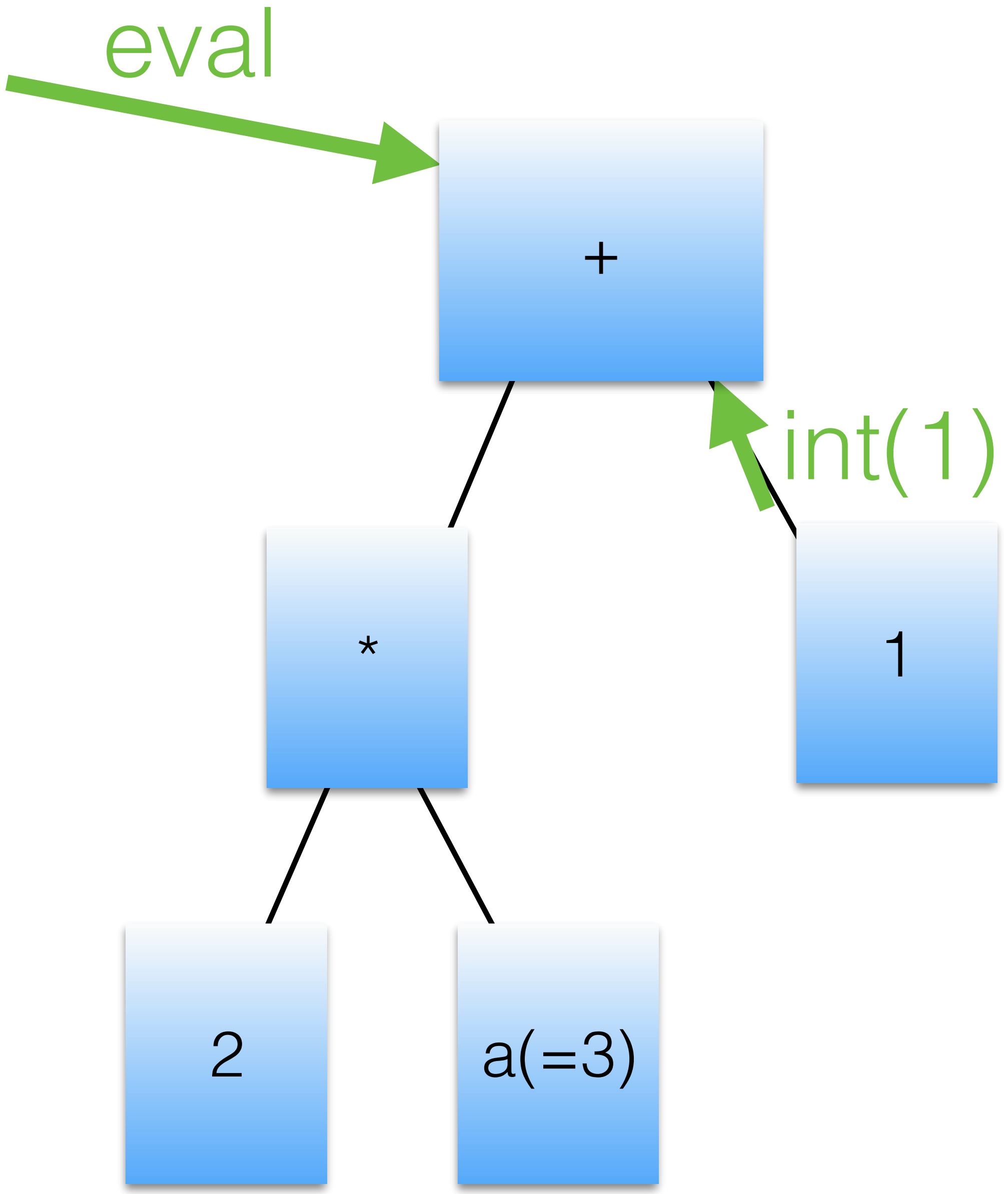
Evolution of an expression
2. Repeated execution



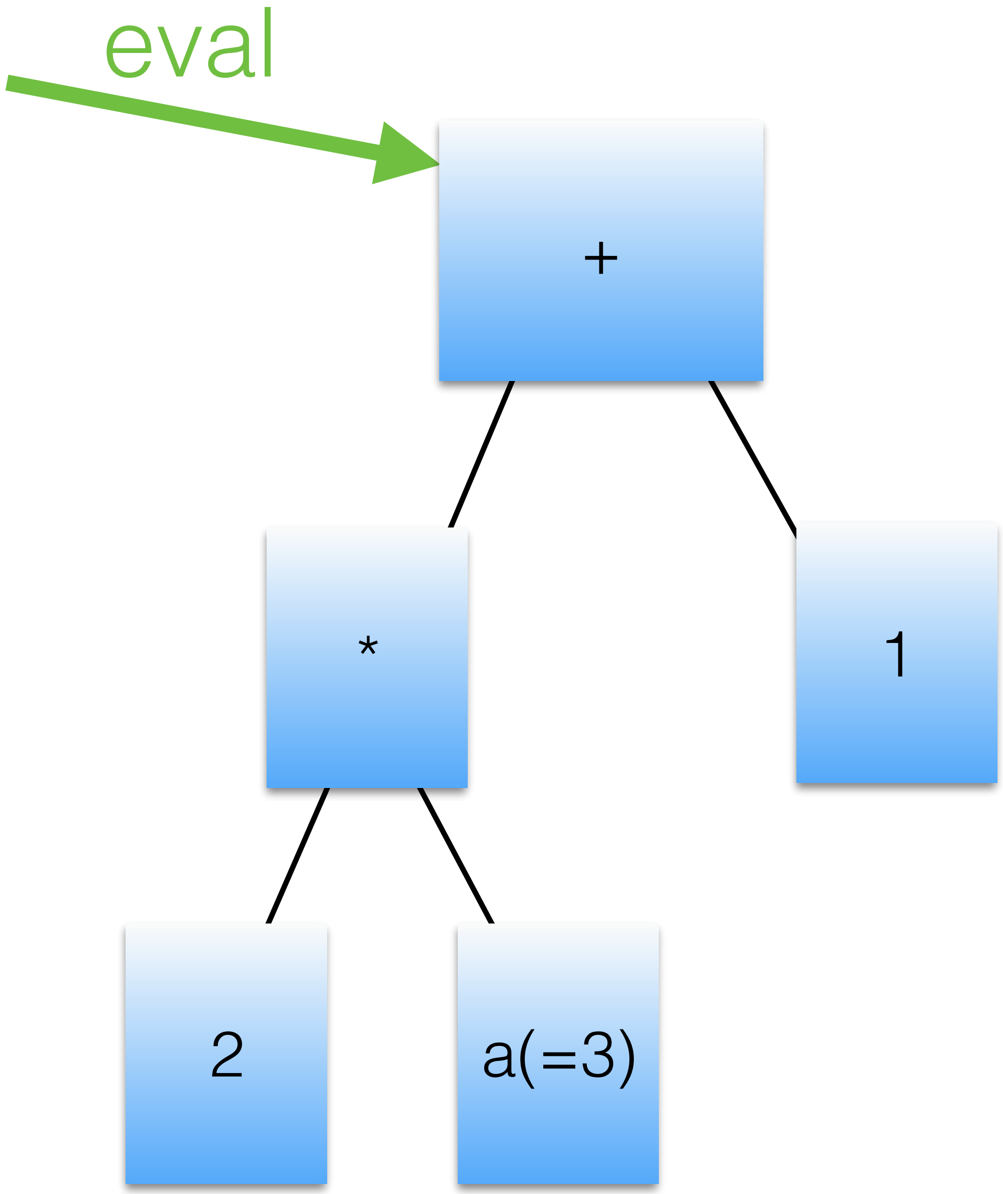
Evolution of an expression
2. Repeated execution



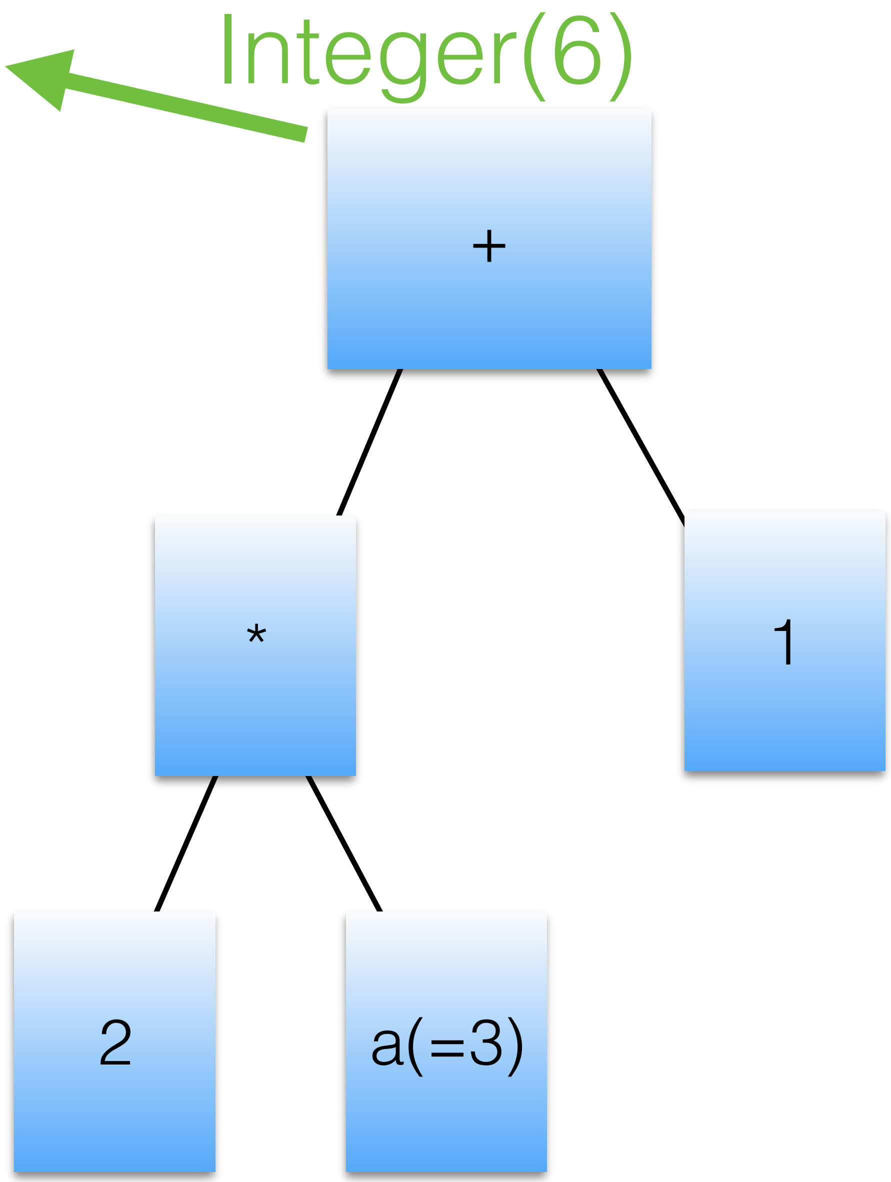
Evolution of an expression
2. Repeated execution



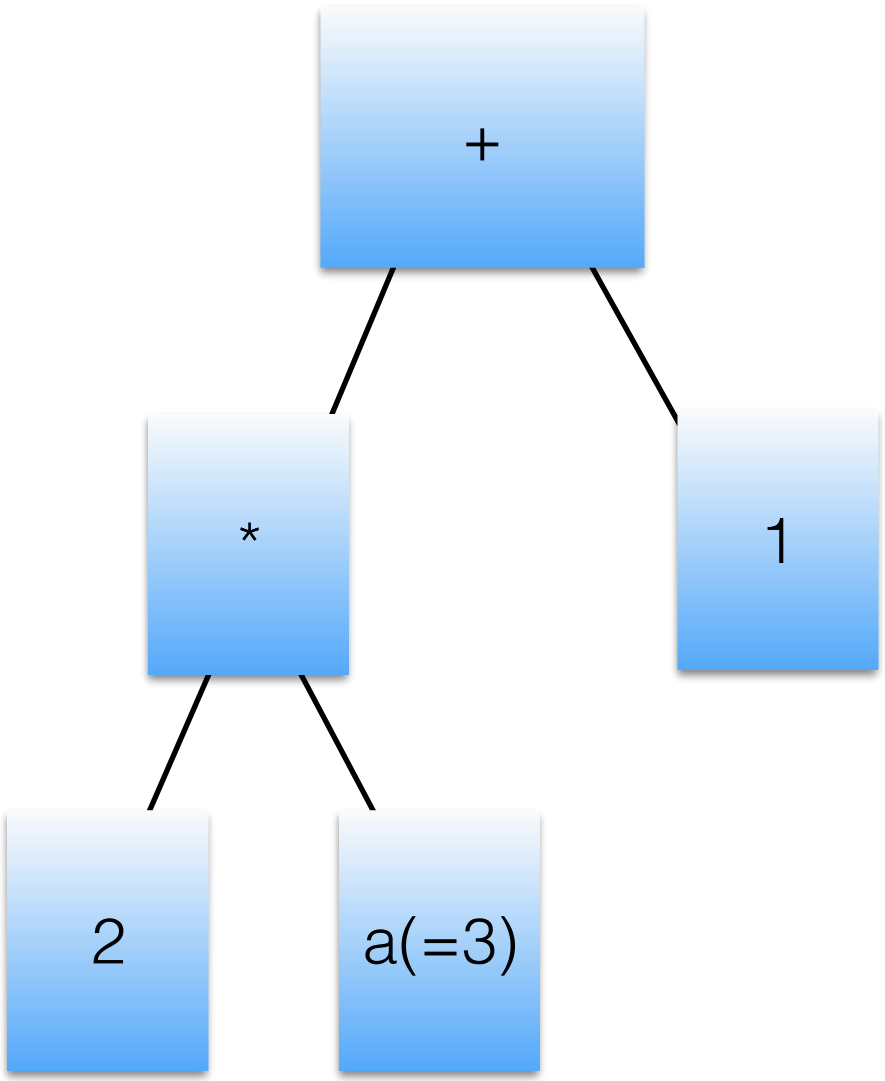
Evolution of an expression
2. Repeated execution



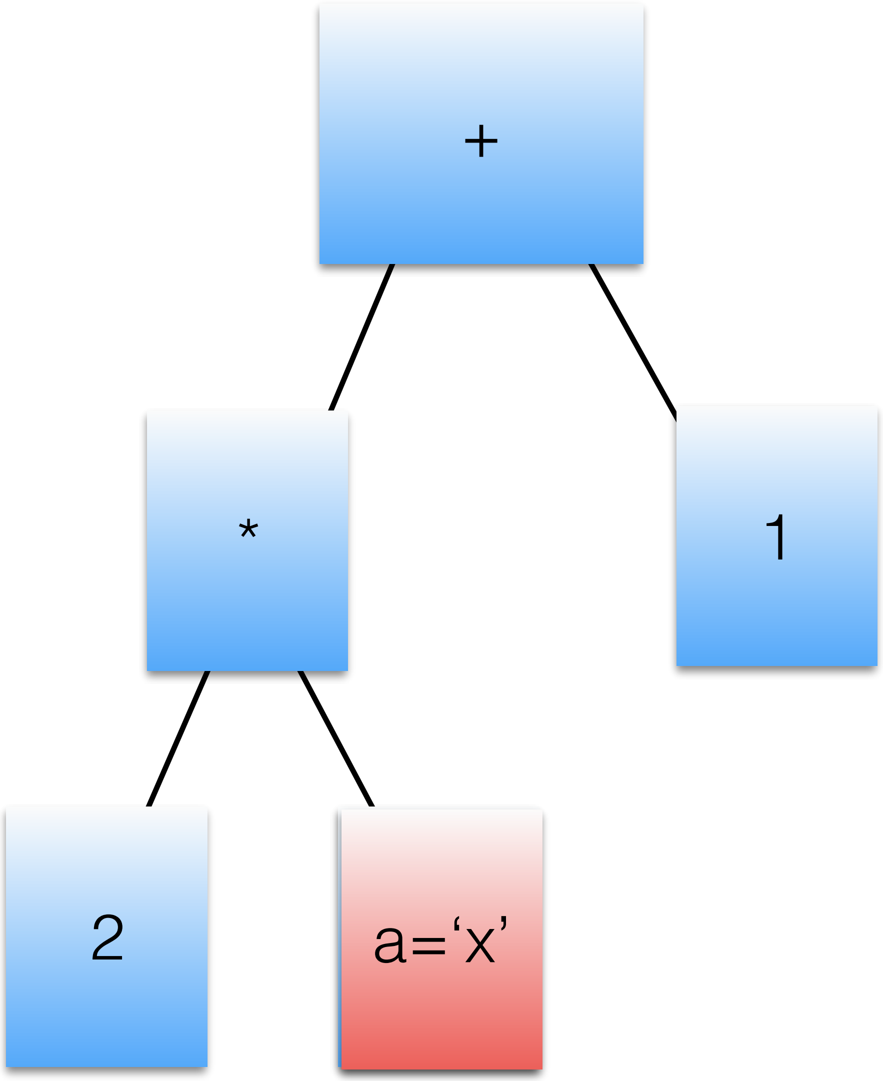
Evolution of an expression
2. Repeated execution



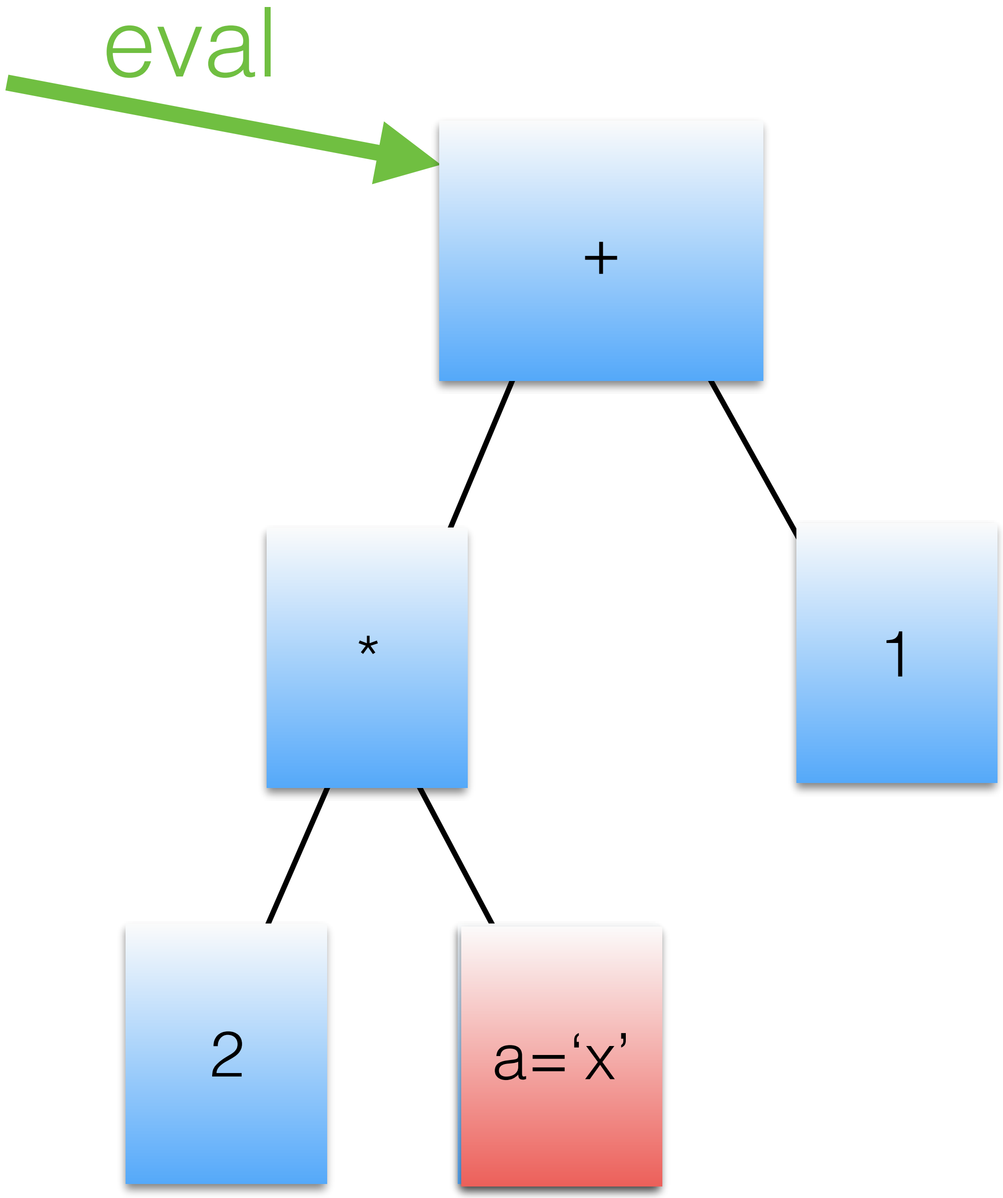
Evolution of an expression
3. Mismatch



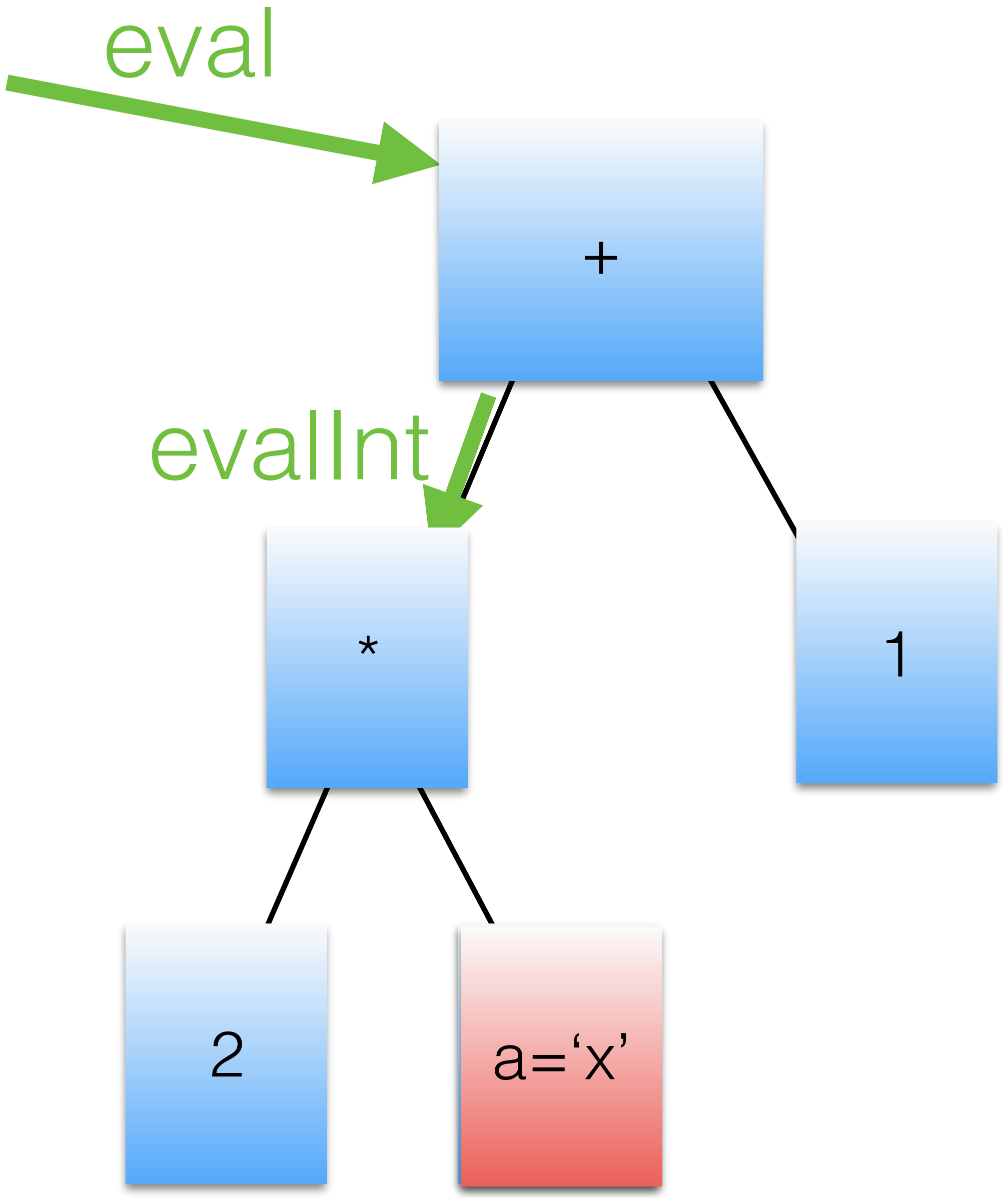
Evolution of an expression
3. Mismatch



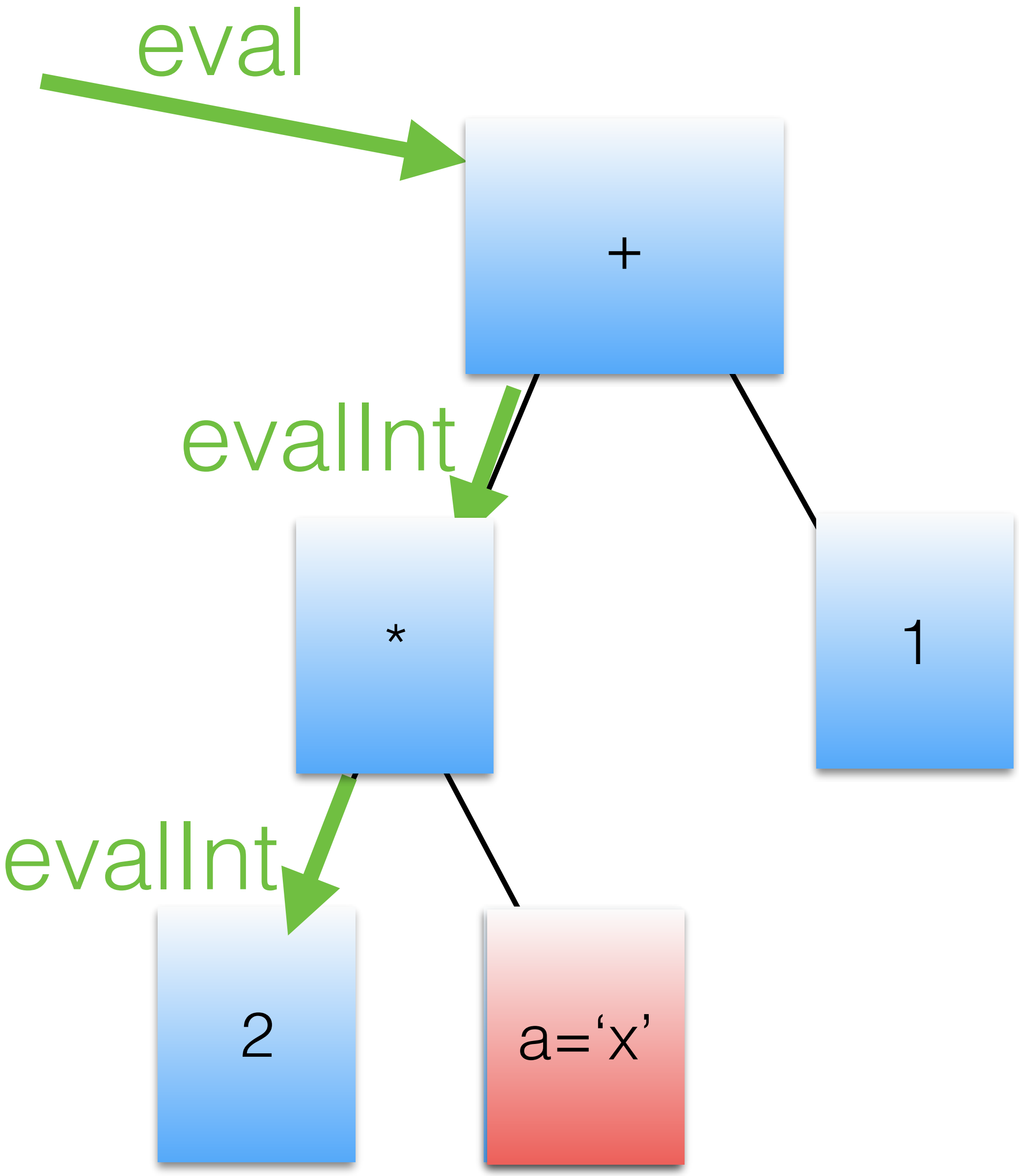
Evolution of an expression
3. Mismatch



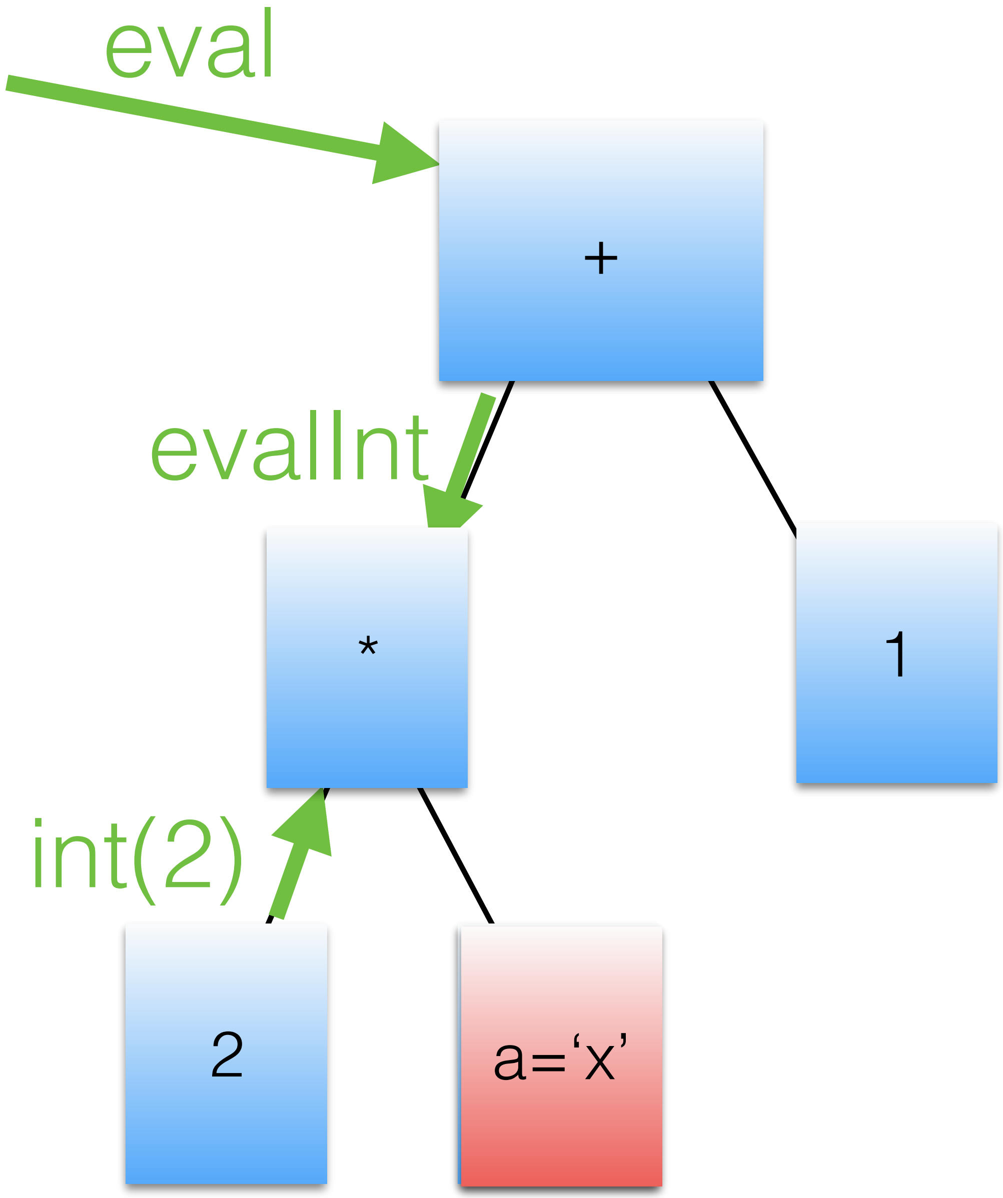
Evolution of an expression
3. Mismatch



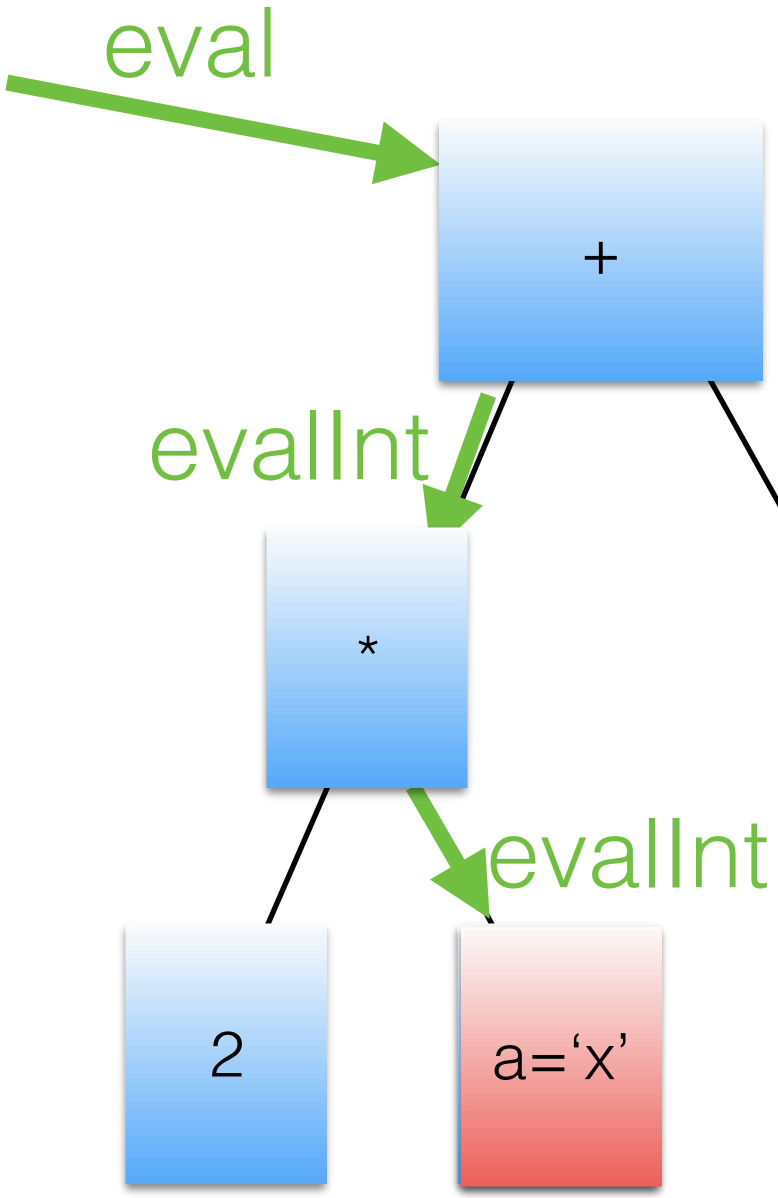
Evolution of an expression
3. Mismatch



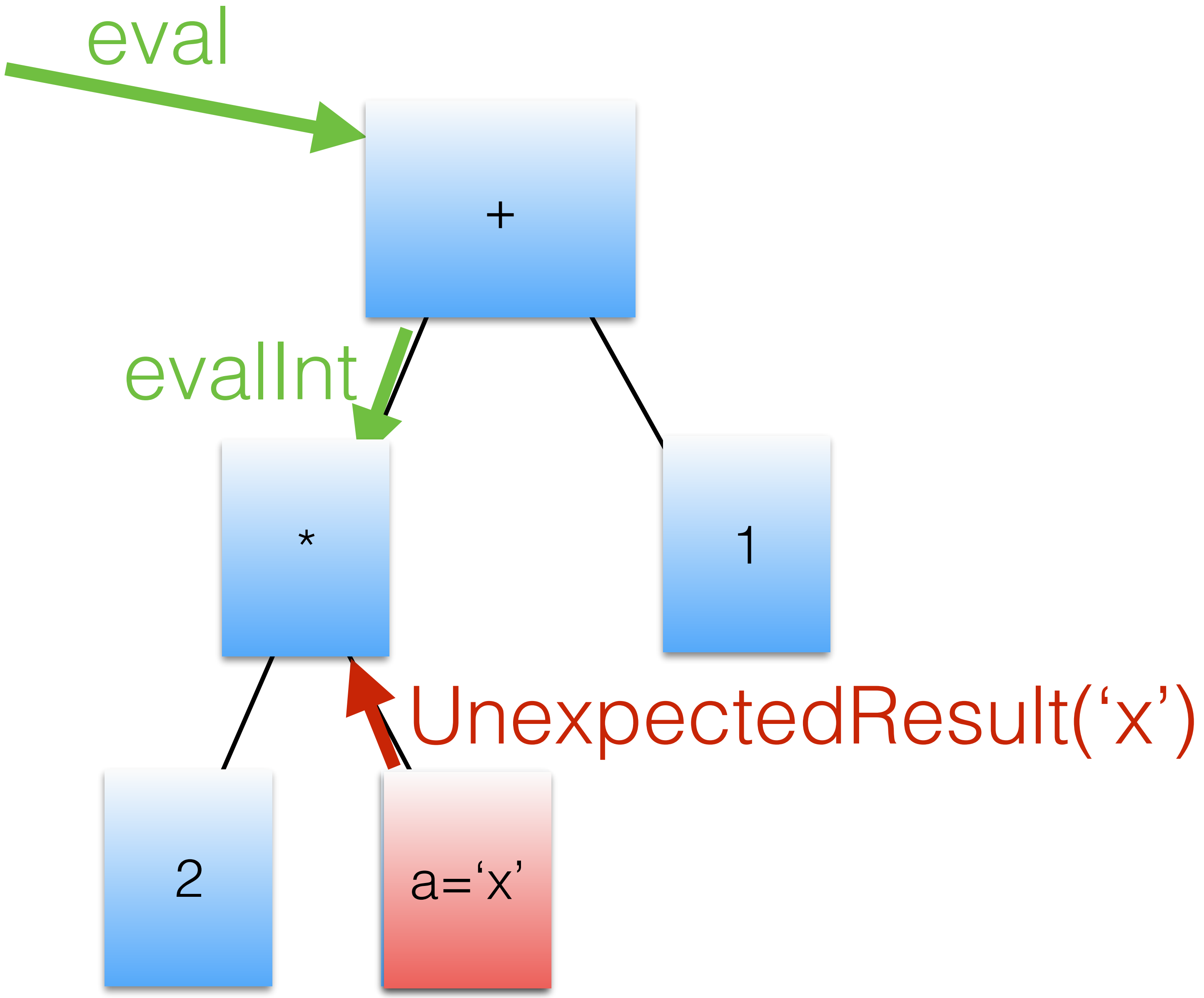
Evolution of an expression
3. Mismatch



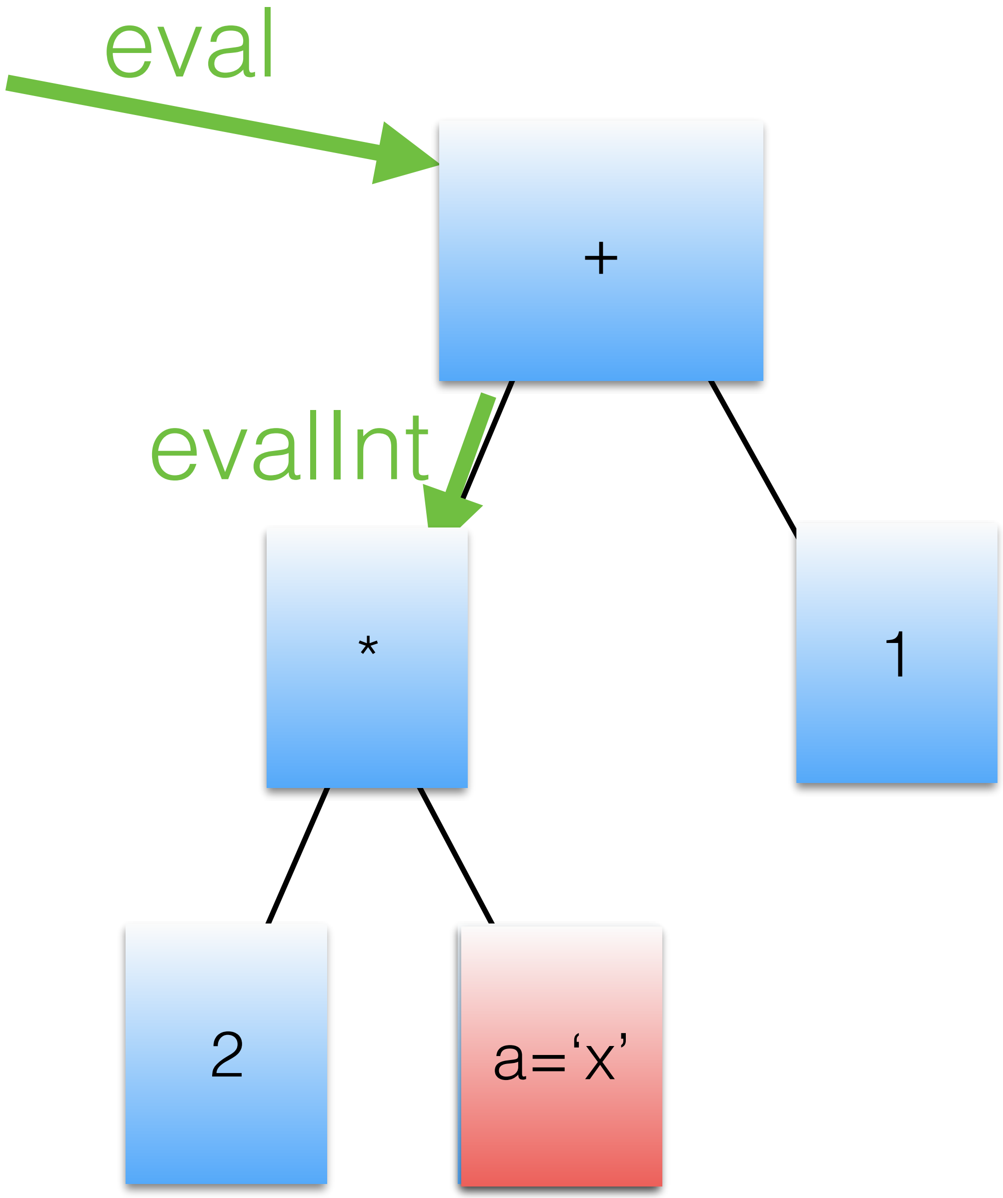
Evolution of an expression
3. Mismatch



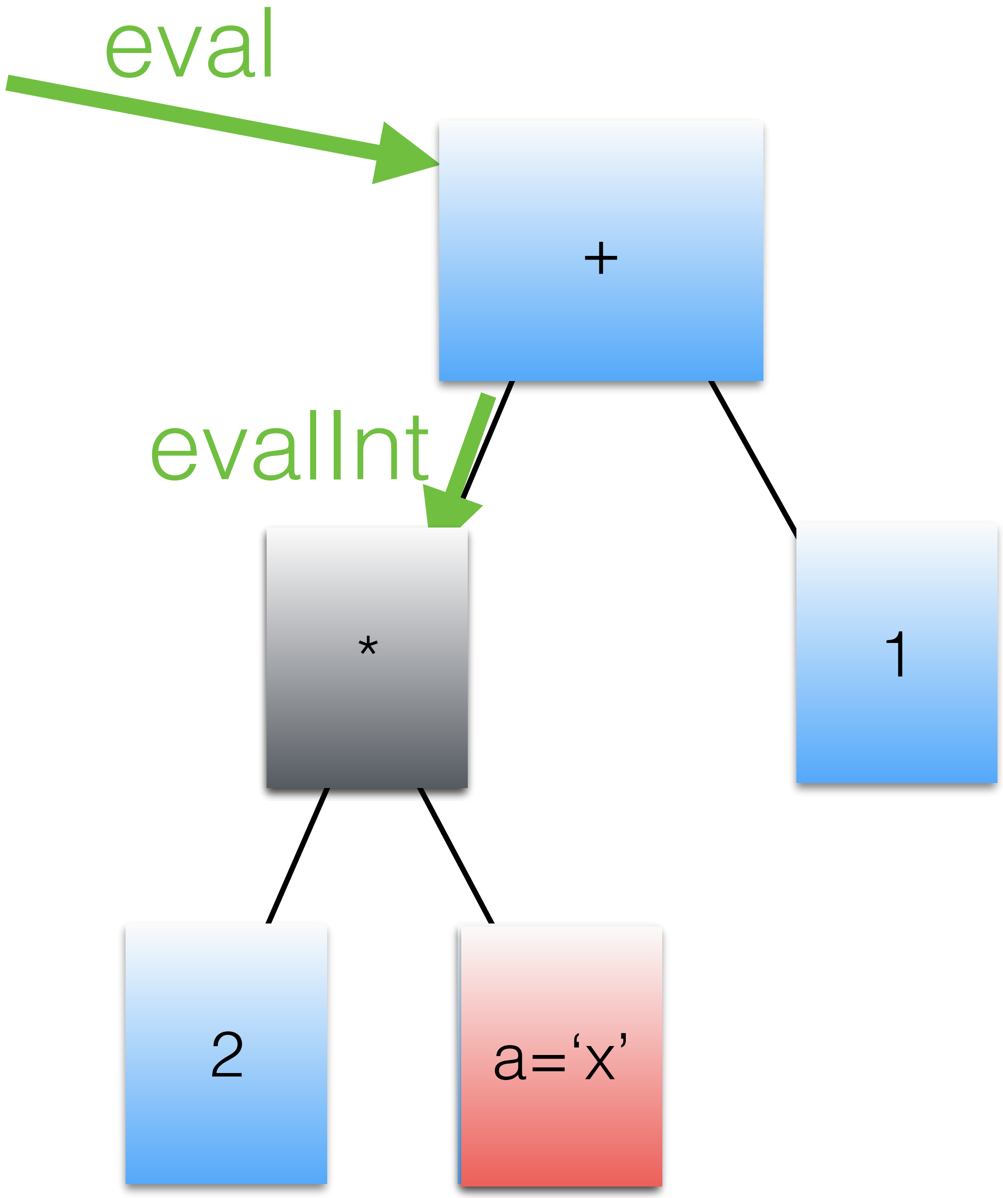
Evolution of an expression
3. Mismatch



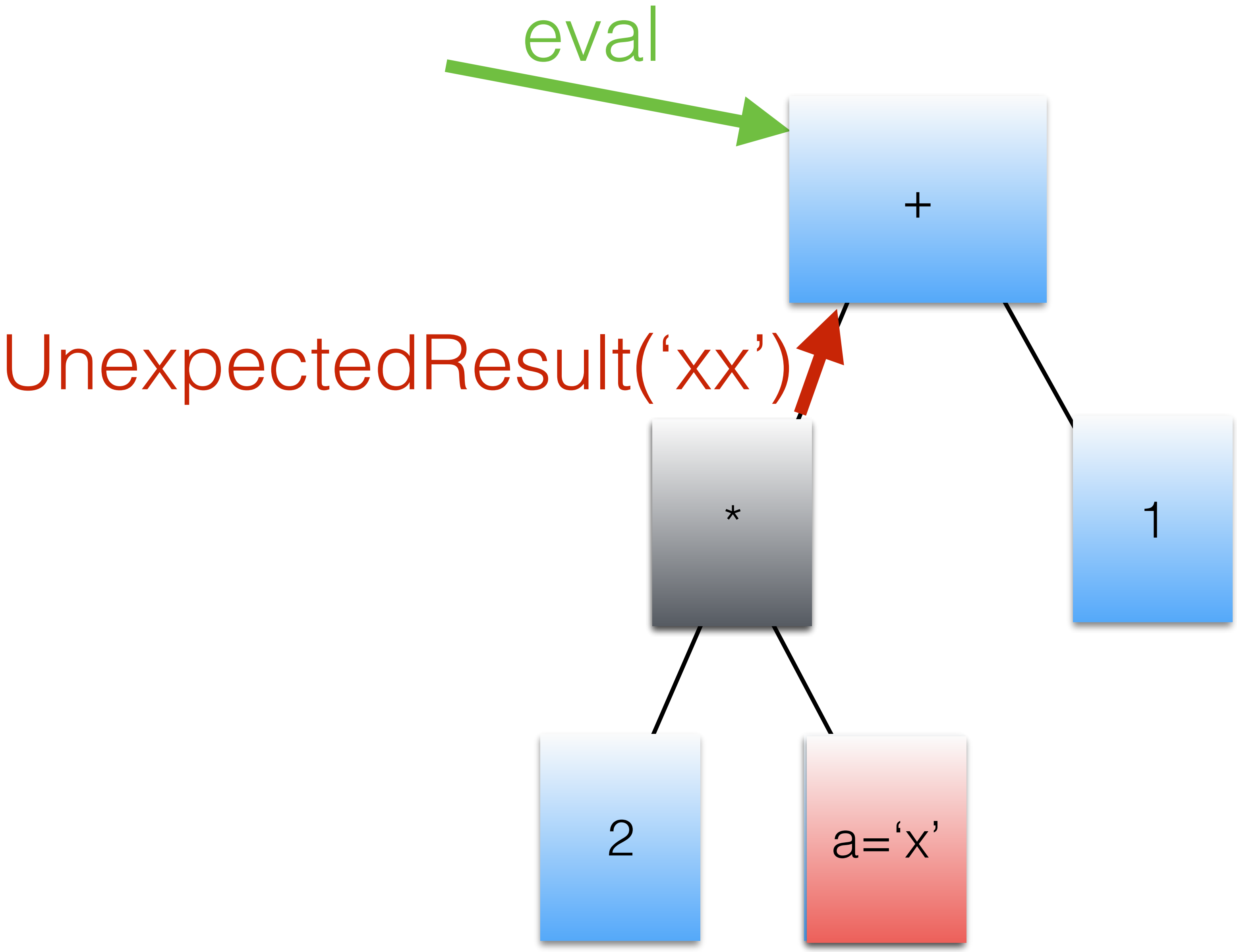
Evolution of an expression
3. Mismatch



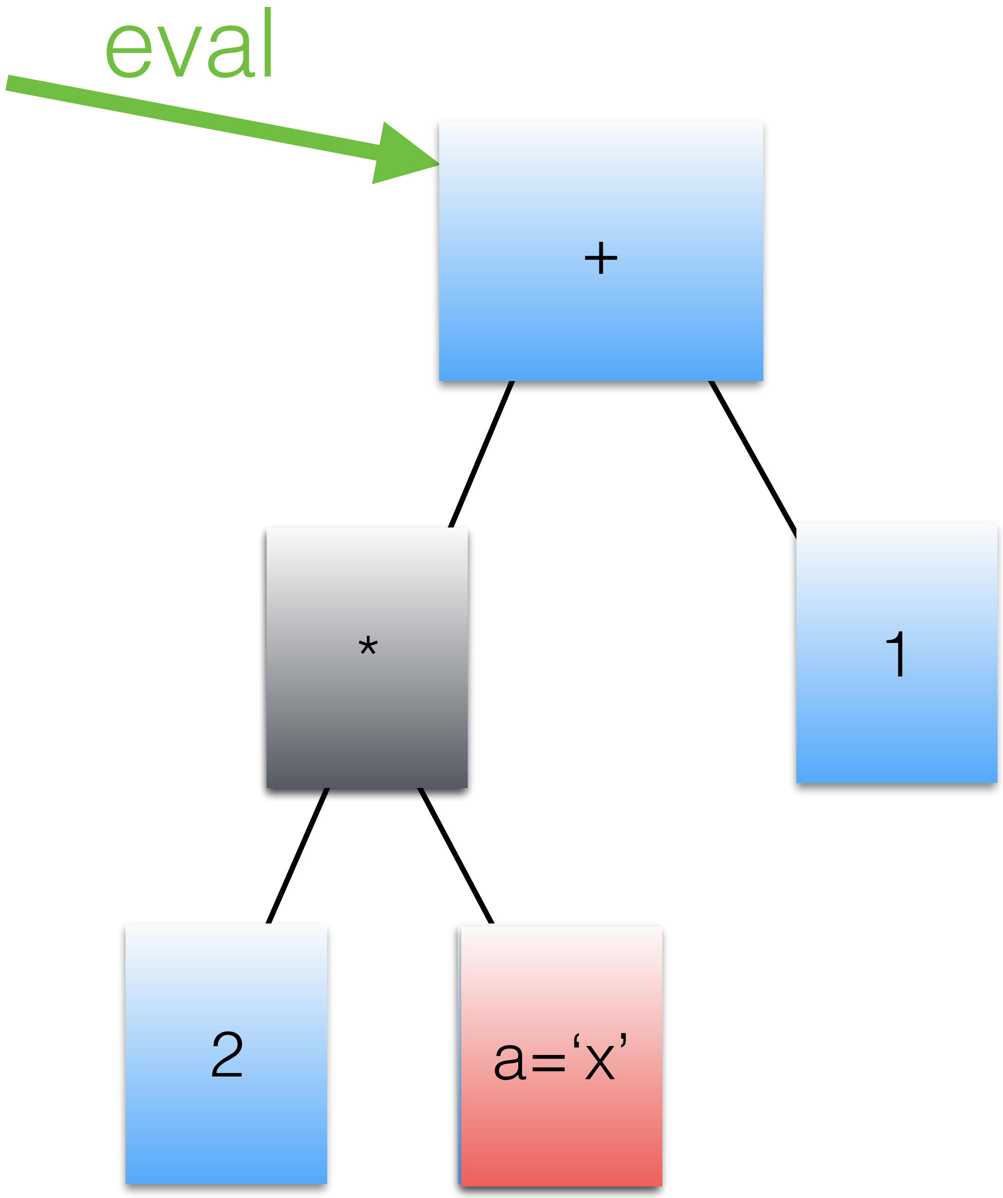
Evolution of an expression
3. Mismatch



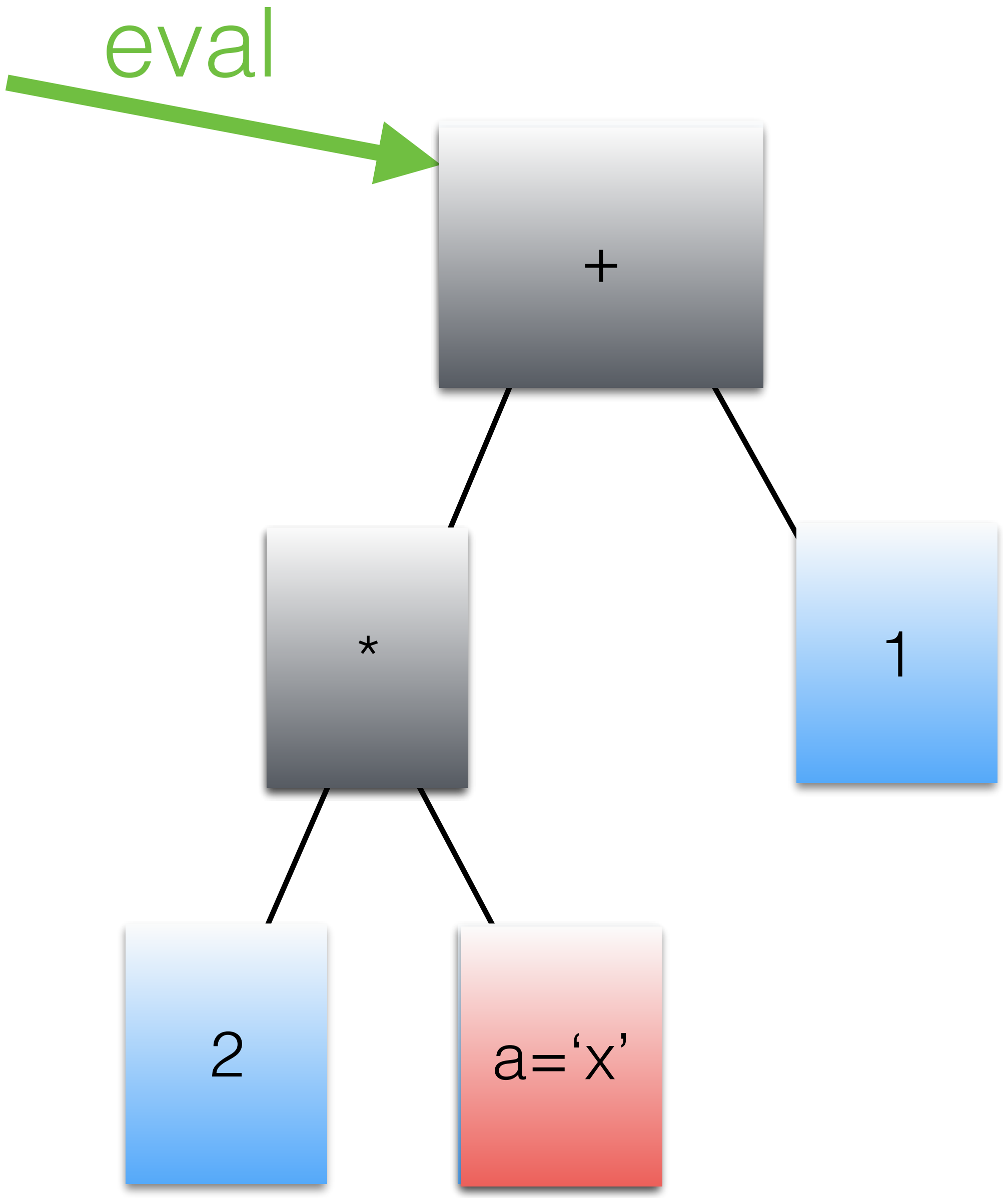
Evolution of an expression
3. Mismatch



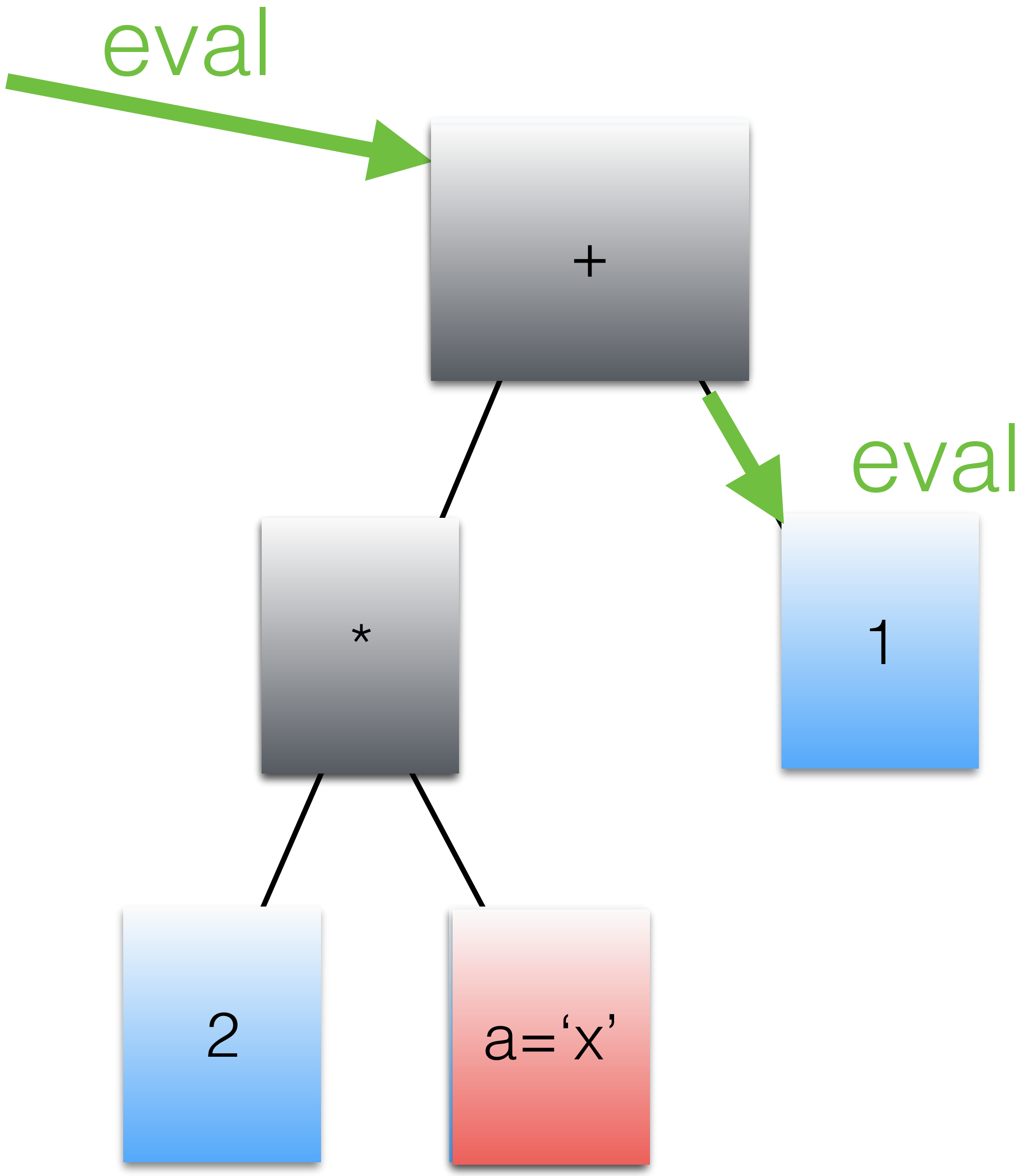
Evolution of an expression
3. Mismatch



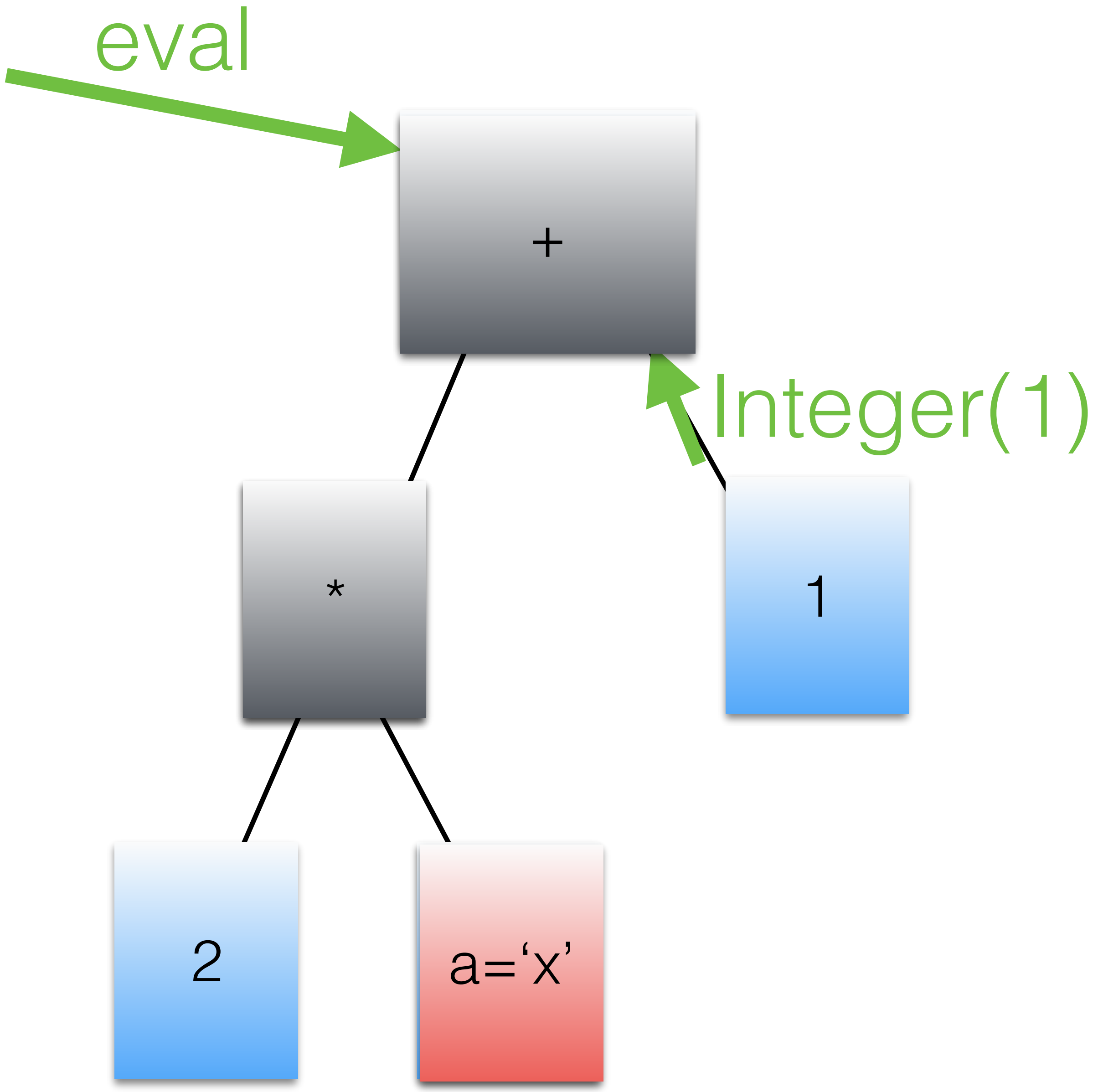
Evolution of an expression
3. Mismatch



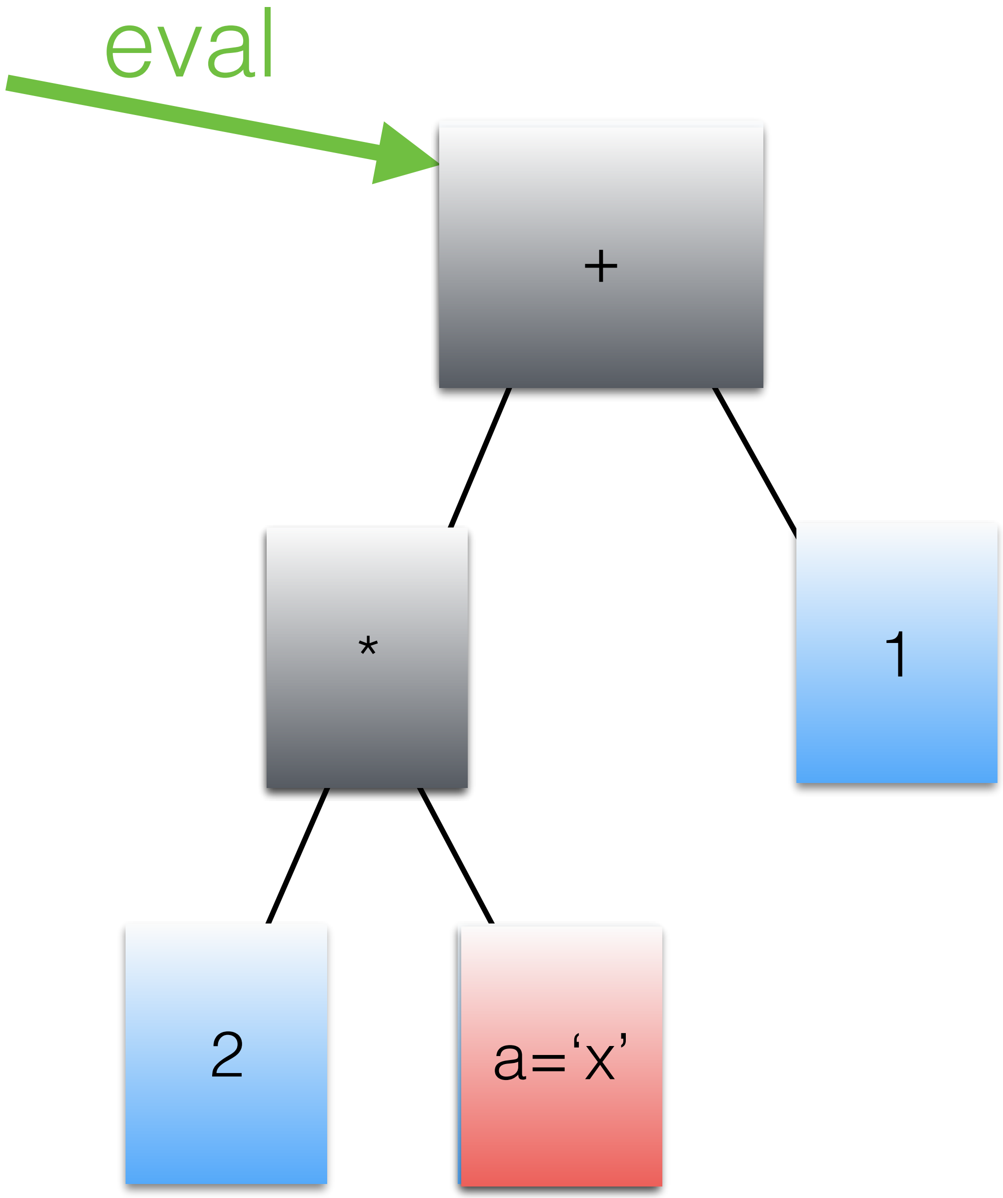
Evolution of an expression
3. Mismatch



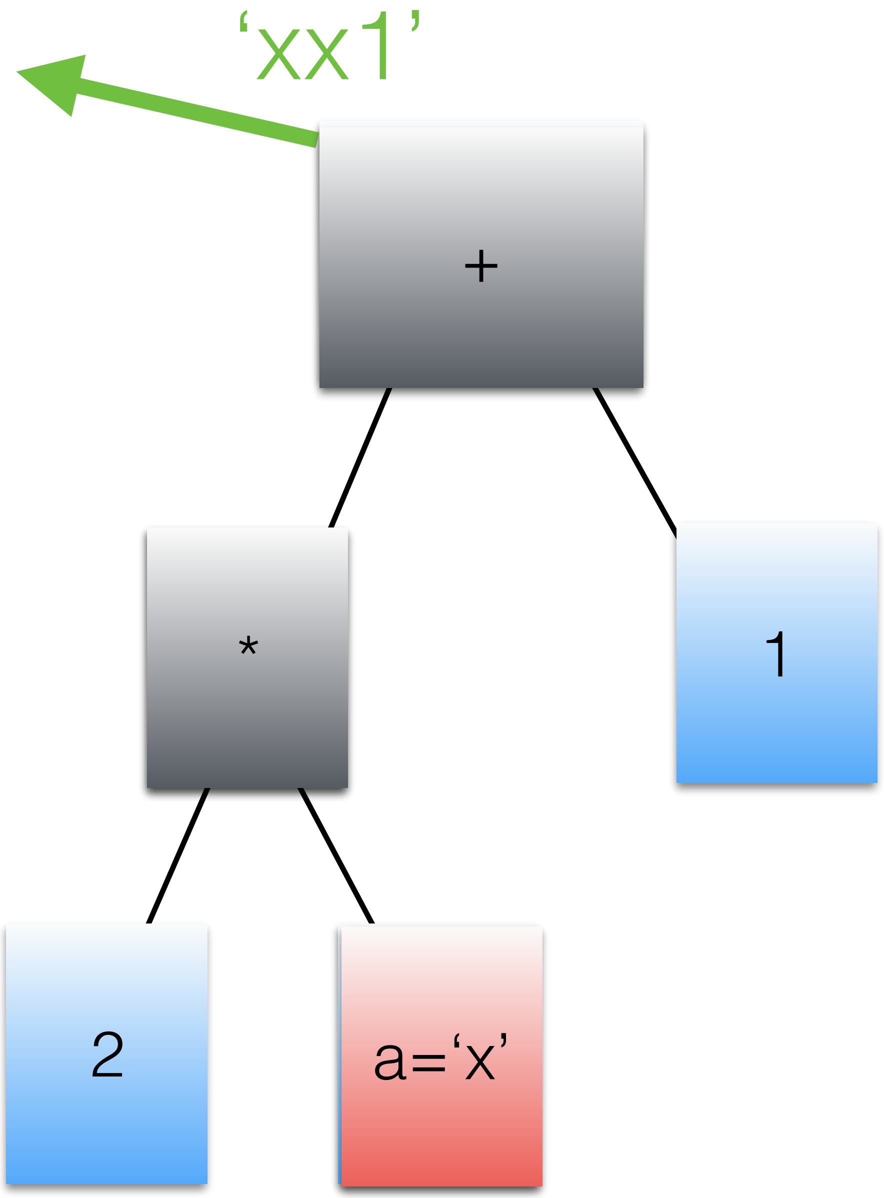
Evolution of an expression
3. Mismatch



Evolution of an expression
3. Mismatch



Evolution of an expression
3. Mismatch



In pseudo-code

(this is not what you actually write)

In pseudo-code

(this is not what you actually write)

```
abstract class Node {
   Node parent; // every node in the AST has a single parent, so that it can replace itself
…}
```

In pseudo-code

(this is not what you actually write)

```
abstract class Node {
   Node parent; // every node in the AST has a single parent, so that it can replace itself
…}

abstract class BinaryNode extends ExprNode {
   ExprNode leftChild, rightChild;
…}
```

In pseudo-code

(this is not what you actually write)

```
abstract class Node {
   Node parent; // every node in the AST has a single parent, so that it can replace itself
…}

abstract class BinaryNode extends ExprNode {
   ExprNode leftChild, rightChild;
…}

abstract class AddNode extends BinaryNode {
  Object eval();
  int evalInt() throws UnexpectedResult;
  String evalString() throws UnexpectedResult;
…}
```

In pseudo-code

(this is not what you actually write)

```
abstract class Node {
   Node parent; // every node in the AST has a single parent, so that it can replace itself
…}

abstract class BinaryNode extends ExprNode {
   ExprNode leftChild, rightChild;
…}

abstract class AddNode extends BinaryNode {
  Object eval();
  int evalInt() throws UnexpectedResult;
  String evalString() throws UnexpectedResult;
…}

class UninitializedAddNode extends AddNode {
  Object eval(); // evaluate and replace with the appropriate specializer
```

In pseudo-code

(this is not what you actually write)

```
abstract class Node {
   Node parent; // every node in the AST has a single parent, so that it can replace itself
…}

abstract class BinaryNode extends ExprNode {
   ExprNode leftChild, rightChild;
…}

abstract class AddNode extends BinaryNode {
  Object eval();
  int evalInt() throws UnexpectedResult;
  String evalString() throws UnexpectedResult;
…}

class UninitializedAddNode extends AddNode {
  Object eval(); // evaluate and replace with the appropriate specializer

class IntAddNode extends AddNode {
  int evalInt() throws UnexpectedResult; // assumes ints, replace with Generic otherwise
```

In pseudo-code

(this is not what you actually write)

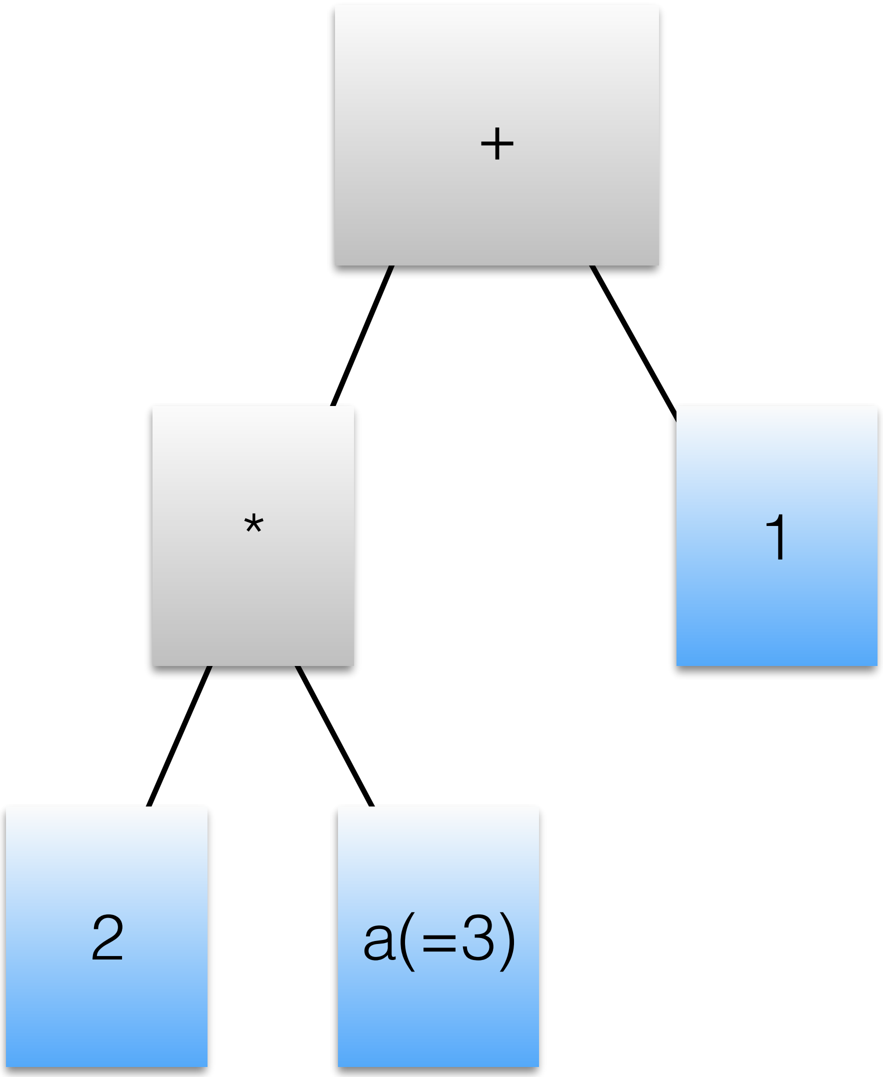
```
abstract class Node {
   Node parent; // every node in the AST has a single parent, so that it can replace itself
…}

abstract class BinaryNode extends ExprNode {
   ExprNode leftChild, rightChild;
…}

abstract class AddNode extends BinaryNode {
  Object eval();
  int evalInt() throws UnexpectedResult;
  String evalString() throws UnexpectedResult;
…}

class UninitializedAddNode extends AddNode {
  Object eval(); // evaluate and replace with the appropriate specializer

class IntAddNode extends AddNode {
  int evalInt() throws UnexpectedResult; // assumes ints, replace with Generic otherwise

class StringConcatNode extends AddNode {
  String evalString() throws UnexpectedResult; // assumes Strings, replace with Generic otherwise
```

In pseudo-code

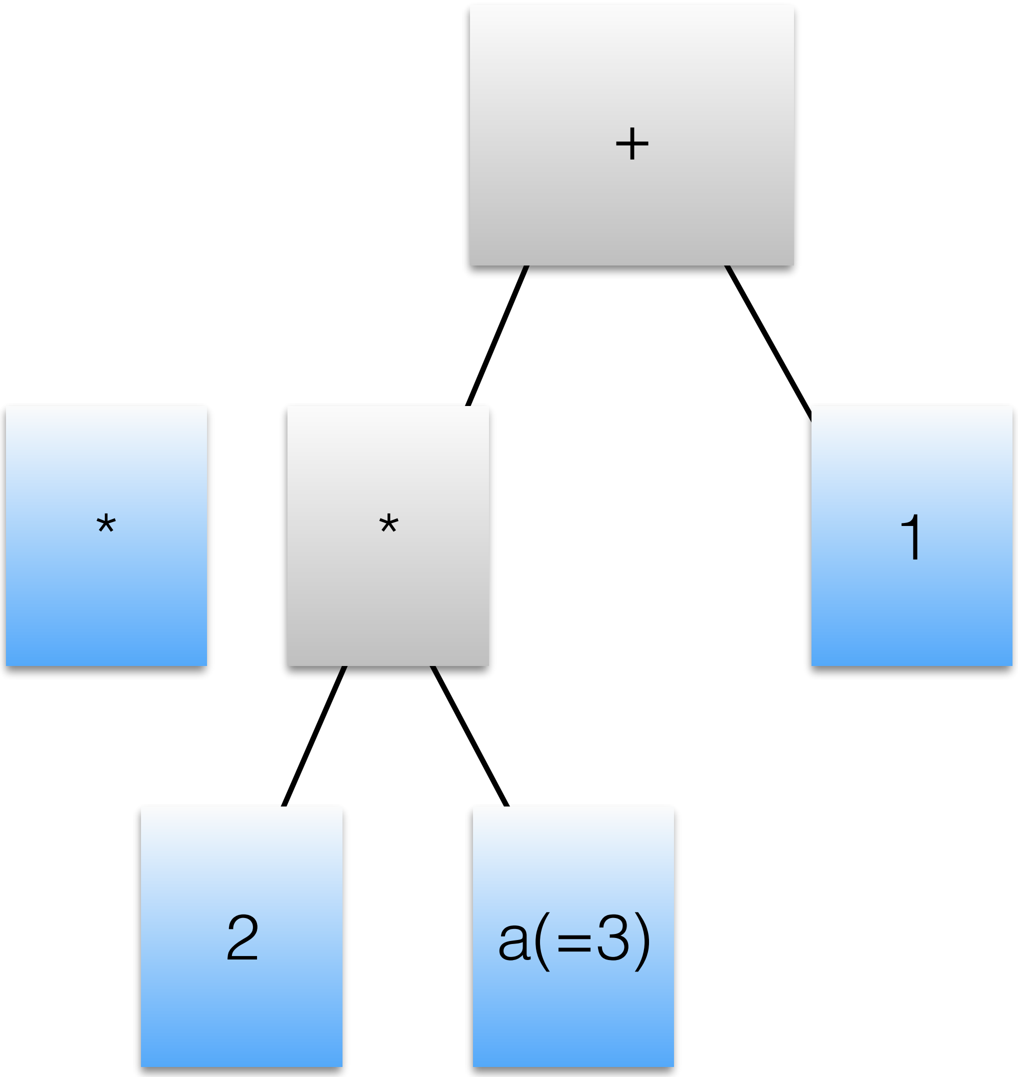
(this is not what you actually write)

```
abstract class Node {
   Node parent; // every node in the AST has a single parent, so that it can replace itself
…}

abstract class BinaryNode extends ExprNode {
   ExprNode leftChild, rightChild;
…}

abstract class AddNode extends BinaryNode {
  Object eval();
  int evalInt() throws UnexpectedResult;
  String evalString() throws UnexpectedResult;
…}

class UninitializedAddNode extends AddNode {
  Object eval(); // evaluate and replace with the appropriate specializer

class IntAddNode extends AddNode {
  int evalInt() throws UnexpectedResult; // assumes ints, replace with Generic otherwise

class StringConcatNode extends AddNode {
  String evalString() throws UnexpectedResult; // assumes Strings, replace with Generic otherwise

class GenericPlusNode extends AddNode {
  Object eval(); // implement user-defined +
```

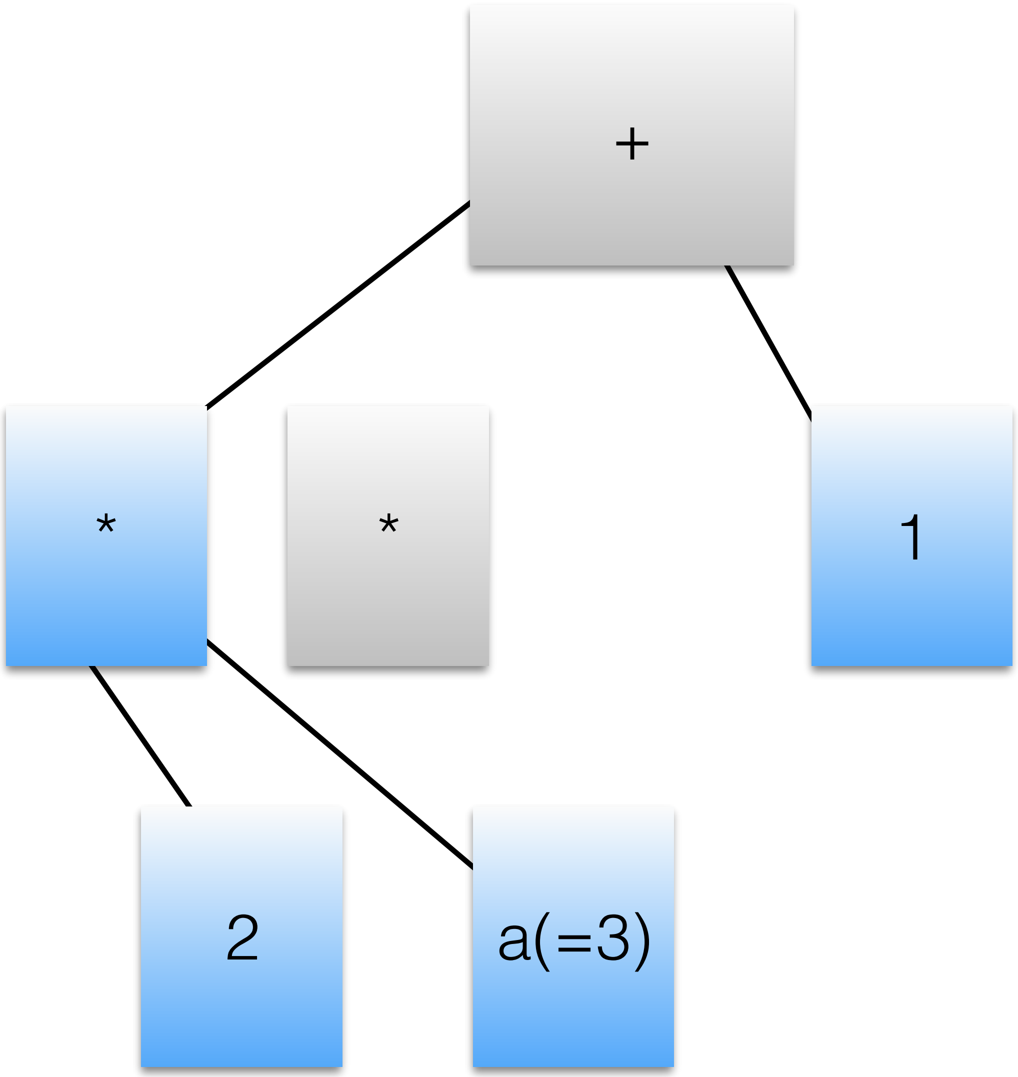
Node replacement



Node replacement



Node replacement



More pseudo-code

```
class UninitializedAddNode extends AddNode {
  Object eval() {
     Object a = leftChild.eval();
     Object b = rightChild.eval();
     if (a instanceof Integer && b instanceof Integer)
        return this.replaceWithIntAddNode(a, b);
     else if (a instanceof String && b instanceof String)
        return this.replaceWithStringConcatNode(a, b);
     else
        return this.replaceWithGenericPlusNode(a, b);
  }
  …
}
```

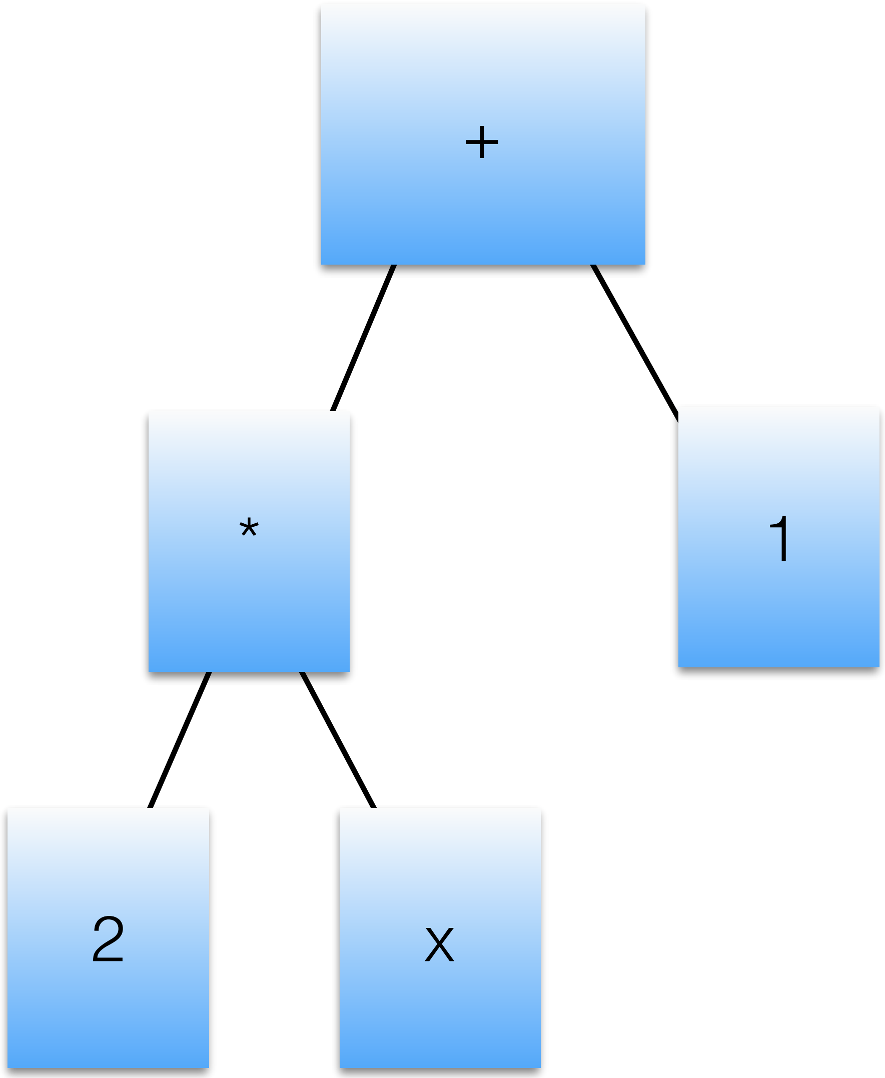
More pseudo-code

```
class IntAddNode extends AddNode {
    int evalInt() throws UnexpectedResult {
      int a;
      try {
        a = leftChild.evalInt();
      } catch {UnexpectedResult ex) {
        throw rewrite(ex.result, rightChild.eval());
      }
      int b;
      try {
        b = rightChild.evalInt();
      } catch {UnexpectedResult ex) {
        throw rewrite(a, ex.result);
      }
      return a+b;
   }
…
```

Compiling the specialized ASTs

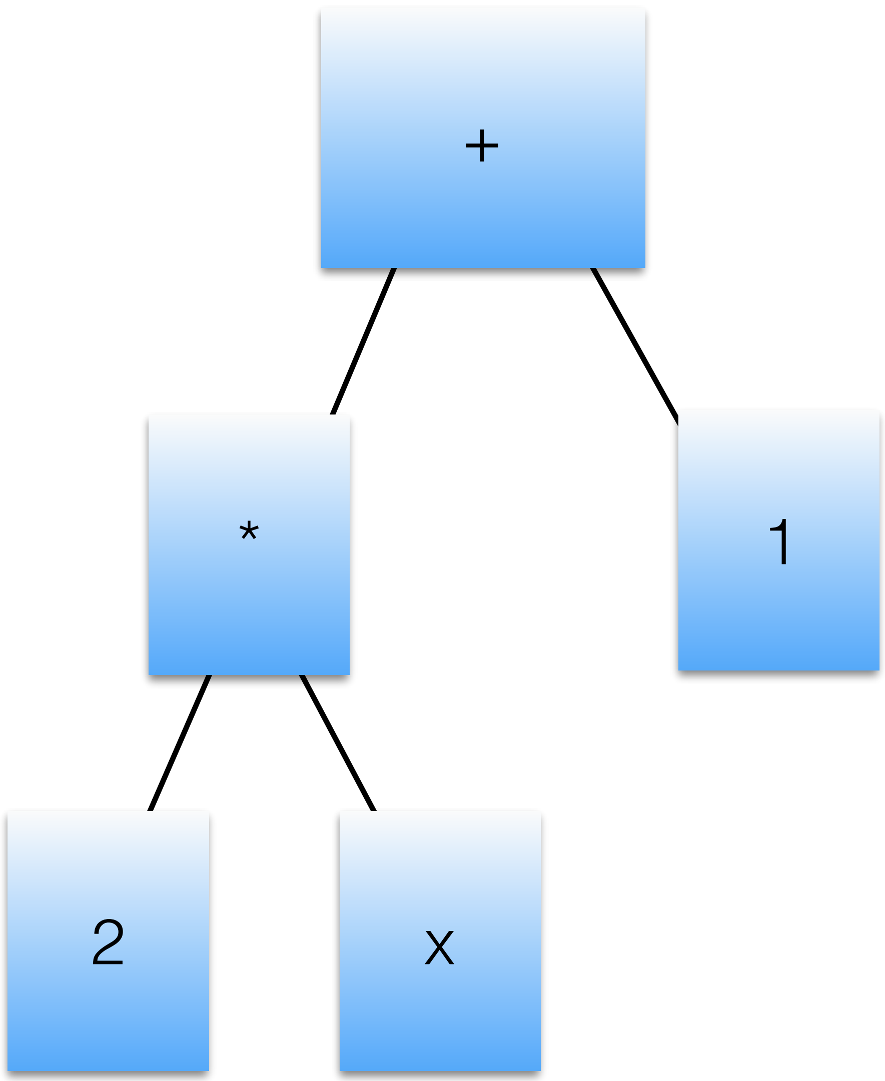
- ASTs are also decorated with counters to do self-profiling
- When a hot AST node is found, the compiler is invoked
- It walks the AST, and uses the specialization and profile information to guide inlining.
- Unexpected types result in deoptimization

Compiling the specialized AST



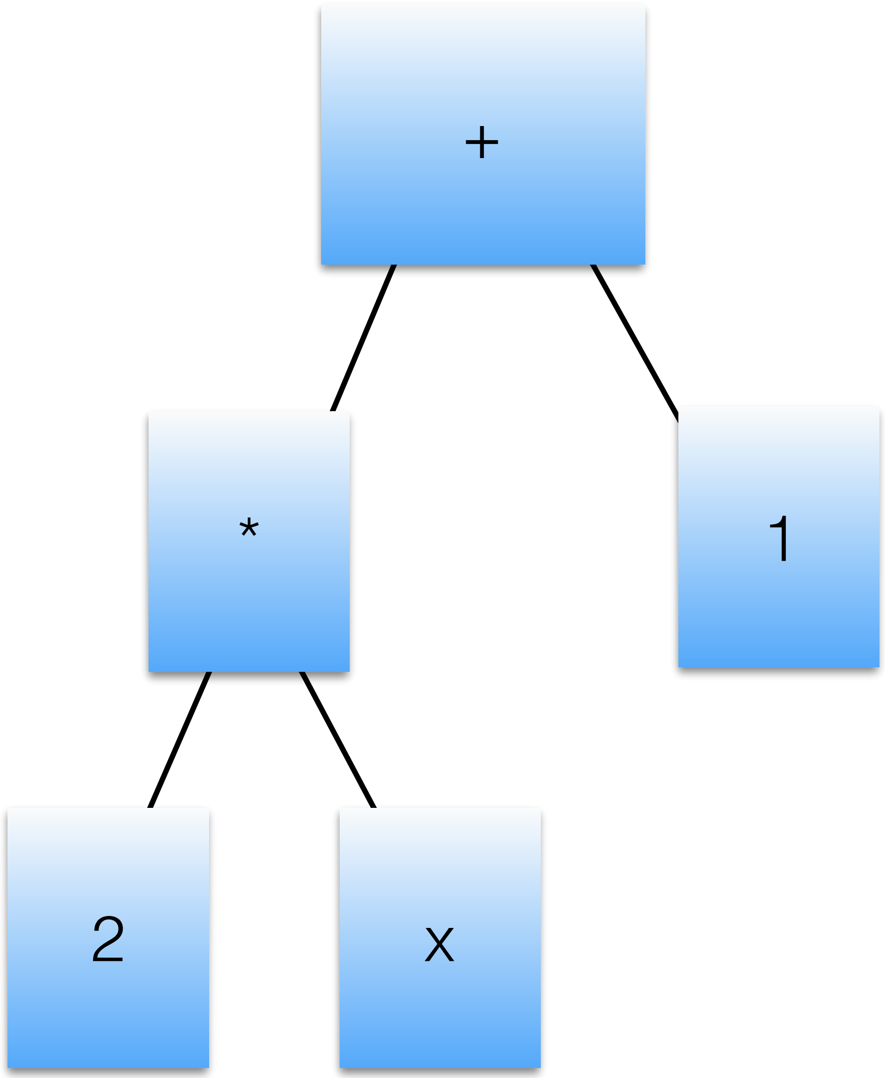
```
class IntAddNode extends AddNode {
  int evalInt() throws UnexpectedResult {
    int a;
    try {
      a = leftChild.evalInt();
    } catch {UnexpectedResult ex) {
      throw rewrite(ex.result, rightChild.eval());
    }
    int b;
    try {
      b = rightChild.evalInt();
    } catch {UnexpectedResult ex) {
      throw rewrite(a, ex.result);
    }
    return a+b;
  }
…
```

Compiling the specialized AST



```
class IntAddNode extends AddNode {
  int evalInt() throws UnexpectedResult {
    int a;
    try {
      a = leftChild.evalInt();
    } catch {UnexpectedResult ex) {
      throw rewrite(ex.result, rightChild.eval());
    }
    int b;
    try {
      b = rightChild.evalInt();
    } catch {UnexpectedResult ex) {
      throw rewrite(a, ex.result);
    }
    return a+b;
  }
…
```

Compiling the specialized AST



```
class IntAddNode extends AddNode {
    int evalInt() throws UnexpectedResult {
      int a;
      try {
        a = {
           a = 2;
           b = var['x']
           return a*b
        }
        b = 1


      }
      return a+b;
   }
…
```

Implementation language desiderata

Implementation language desiderata

- Must handle machine types

Implementation language desiderata

- Must handle machine types
- Exception paths for unexpected types

Implementation language desiderata

- Must handle machine types
- Exception paths for unexpected types
- Efficient code generation

Implementation language desiderata

- Must handle machine types
- Exception paths for unexpected types
- Efficient code generation
- Code processing/generation to handle boiler-plate

Implementation language desiderata

- Must handle machine types
- Exception paths for unexpected types
- Efficient code generation
- Code processing/generation to handle boiler-plate

Java with compiler directives, annotations and an annotation processor ⇒ the **Truffle DSL**

5. Wrap-up

The Future

- Multi-lingual support is ~~coming~~ here
- Further proliferation as VMs get easier to build
- New languages? New DSLs?

Acknowledgments

- Thanks to the following for comments on drafts and discussions:
  Laurie Tratt, Carl Friedrich Bolz, Peter Kessler, Michael Van De Vanter, Adam Welc, Laurent Daynès, Christian Wimmer.