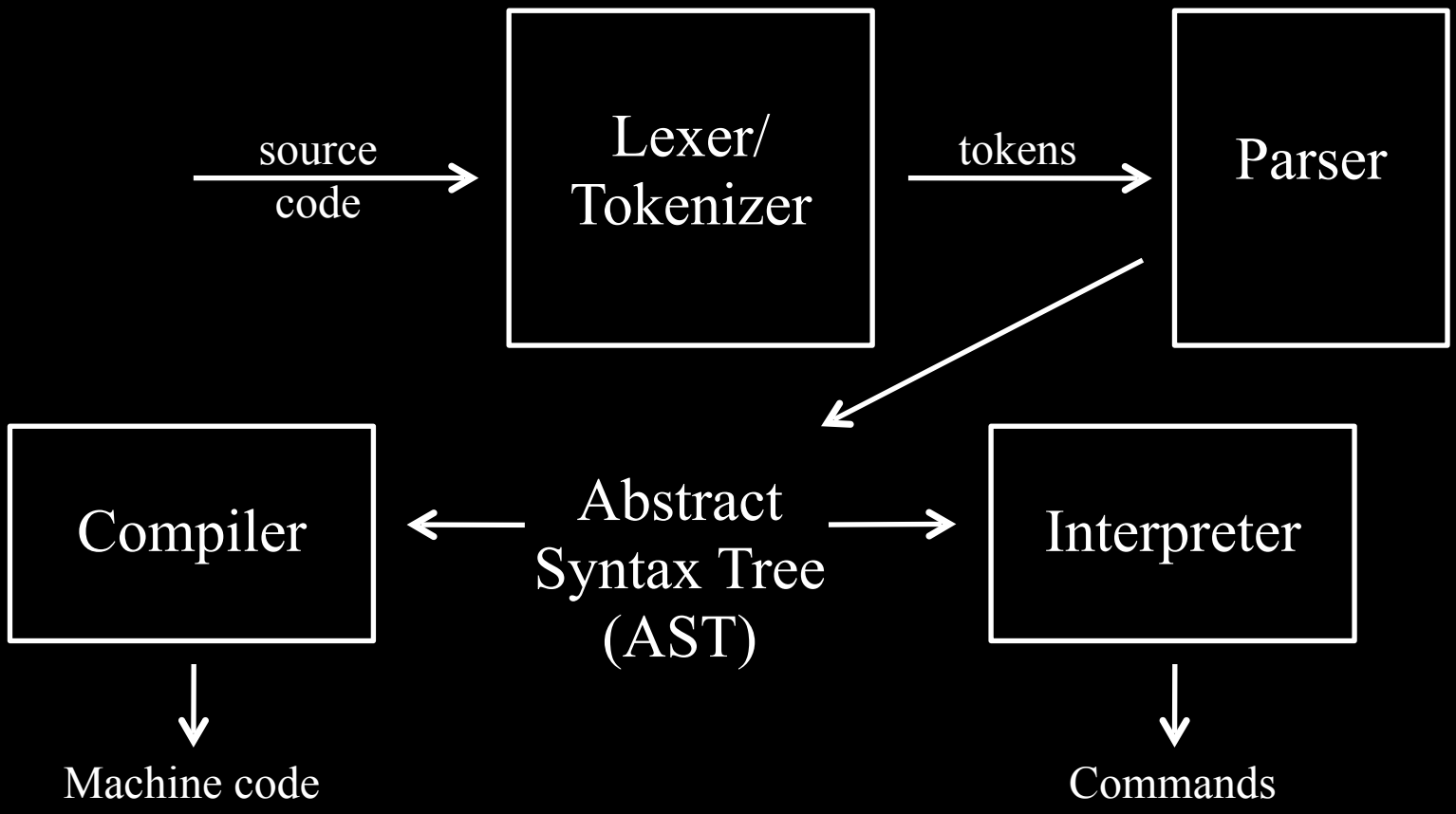
CS 252:
Advanced Programming Language Principles

Virtual Machines and JITs

Prof. Tom Austin
San José State University

A Review of Compilers



Virtual Machines (VM)

- Code is compiled to *bytecode*
  - low-level
  - platform independent
- The VM interprets bytecode

Lab: Scheme VM

In today's lab, you will implement:

- a compiler for Scheme
- a stack-based VM

Input program

```
(println (+ 2 3 4))
(println (- 13 (* 2 4)))
(println (- 10 4 3))
```

Sample VM Operations

- **PUSH1** – adds argument to stack
- **PRINT** – pops & prints top of stack
- **ADD**
  - pops top two elements
  - adds them together
  - places result on stack
- **SUB** – subtraction
- **MUL** – multiplication
- **SWAP1** – swaps top two arguments on the stack

Bytecode Representation

```
PUSH1 2
PUSH1 3
ADD
PUSH1 4
ADD
PRINT
PUSH1 13
PUSH1 2
PUSH1 4
MUL
SWAP1
SUB
PRINT
PUSH1 10
PUSH1 4
SWAP1
SUB
PUSH1 3
SWAP1
SUB
PRINT
```

Lab, part 1: Run compiler/VM

- Run:
  ```
  $ node compiler.js test1.scm
  ```
- Inspect `test.byco`
- Run:
  ```
  $ node vm.js test1.byco
  ```

Node Buffer class

- Fixed-length sequences of bytes
- Good for low-level data management
- https://nodejs.org/api/buffer.html

Buffer static methods

- Key static methods:
  - `Buffer.alloc(n)`: creates a buffer of `n` bytes
  - `Buffer.from(d)`: creates a buffer to fit the data `d`
- Key instance methods
  - `buff.writeUInt8(b,p)`: writes byte `b` to position `p` of buffer `buff`.
  - `buff.readUInt8(p)`: reads byte from position `p` of buffer `buff`.

Lab, part 2: Add Opcodes to VM

- Add the MUL, SUB, and SWAP1 opcodes to `op-codes.js`.
- Review the details from https://ethervm.io/ and https://ethereum.org/en/developers/docs/evm/opcodes/.
- Test with `test2.byco`.

Lab, part 3: Add Compiler support for '*' and '-'

- Once you have MUL, SUB, and SWAP1, implement multiplication and subtraction.
- Modify the `writeBytecode` function in compiler.js to add in this support.
- Test with `test2.scm`.

Compiler or Interpreter?

- Compilers
  - –efficient code
- Interpreters
  - –runtime flexibility
- Can we get the best of both?

Just-in-time compilers (JITs)

- interpret code
- "hot" sections are compiled *at run time*

JIT tradeoffs

+ Speed of compiled code
+ Flexibility of interpreter
− Overhead of both approaches
− Complex implementation

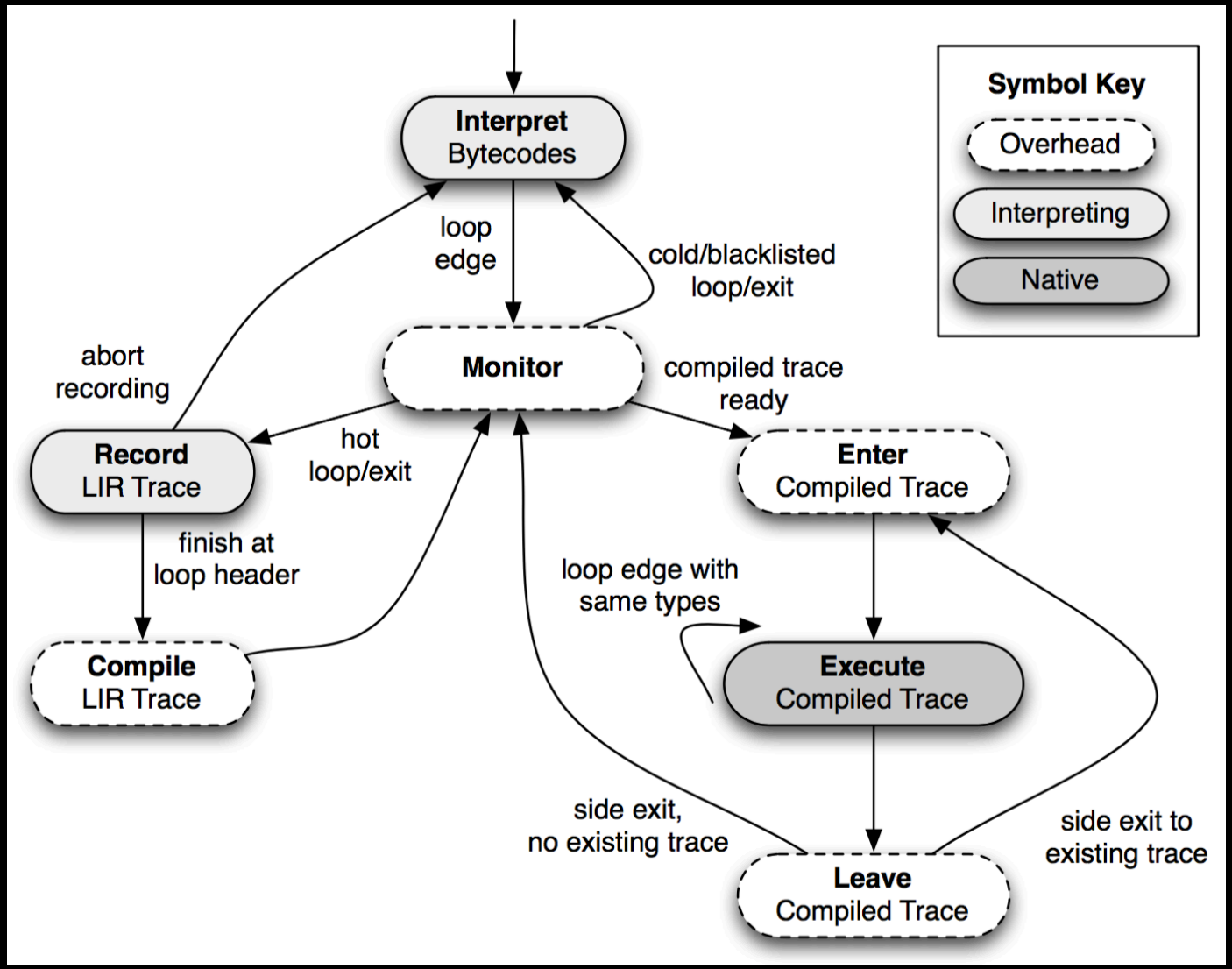
Dynamic recompilation

- JIT pursues aggressive optimizations
  - make assumptions about code
  - guard conditions verify assumptions
- Unexpected cases interpreted
- Can outperform static compilation

Types of JITs

- Method based
  - Compiles methods
- Trace based
  - Compiles loops
  - Gal et al. 2009 http://www.stanford.edu/class/cs343/resources/tracemonkey.pdf

Trace-based JIT design (Gal et al. 2009)



How can a language designer make use of a JIT?

1. Become an expert in JITs
   - study the latest techniques
   - build large code bases to test
   - profile your code execution
2. Use someone else's JIT-ed VM

Lab, part 4 – Support for Variables

- Add support for MSTORE and MLOAD to VirtualMachine class.
- We simplify them so that they only load a byte.
- The VirtualMachine class has a "memory" array.

- The scheme `define` keyword assigns values to variables:

```
(define x 3) ; sets x to 3.
```

- Test out your solution against `store.scm`.

Lab, part 5 – Support Conditional Expressions (EXTRA CREDIT)

- Update `compiler.js` to support booleans and 'if' expressions.
- You need VM support for 'JUMP' and 'JUMPI'.
- 'JUMPDEST' is already implemented, and may prove useful.