CSc 553
Principles of Compilation

10. Interpreters and JIT Compilers

Saumya Debray

The University of Arizona

Tucson, AZ 85721

Overview

Interpreters

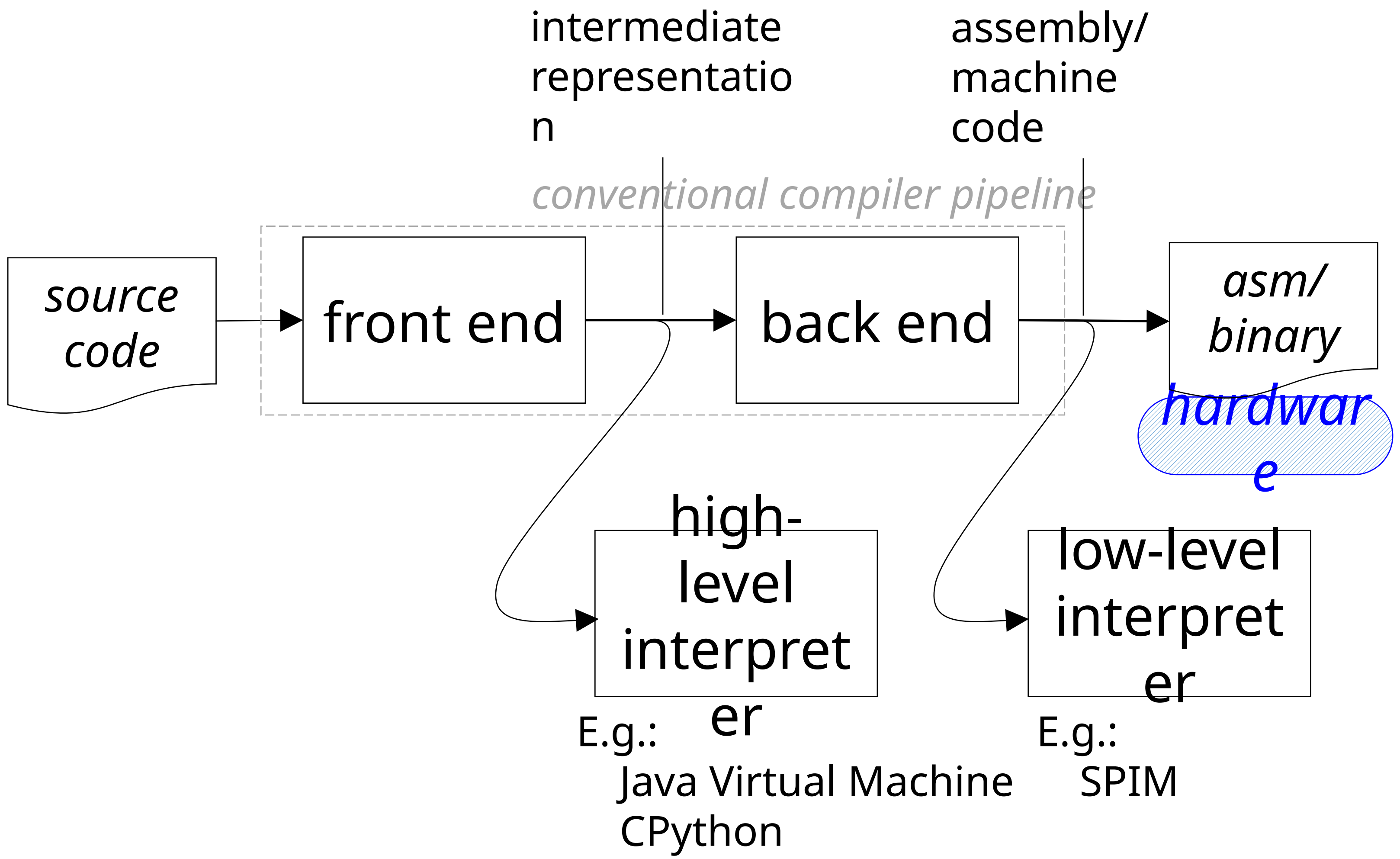
An *interpreter* is a program that executes another program.

- implements a *virtual machine*, which may be different from the underlying hardware platform.
- The virtual machine is often at a higher level of semantic abstraction than the native hardware.
  - “higher level of semantic abstraction” ≡ operations are closer to the source language
  - This can help portability, but affects performance.

Some common language interpreters

JavaScript

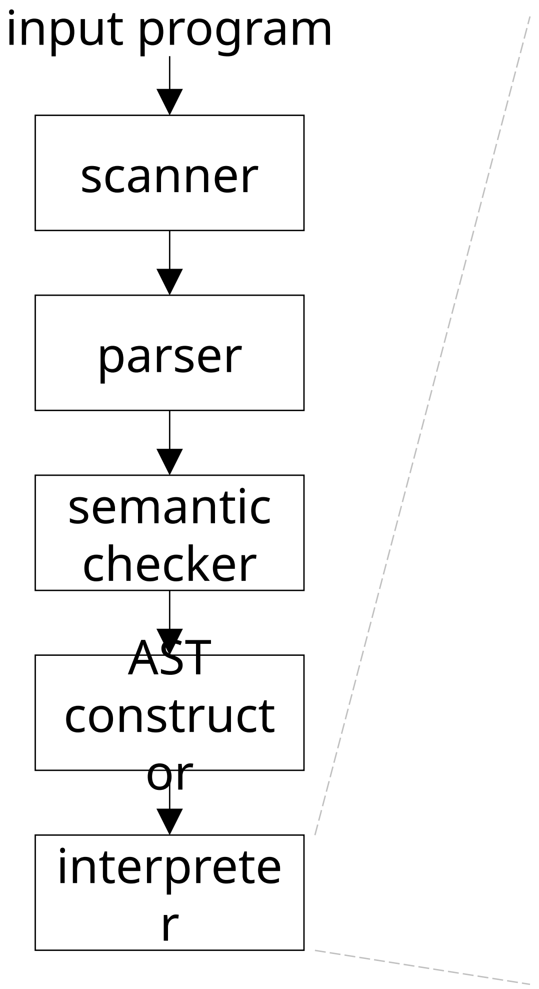
Interpretation vs. Compilation



Interpreter Operation

```
ip = start of program    /* ip ≡ instruction pointer */
while ( not done ) {
    op = current operation at ip
    execute code for op on current operands
    advance ip to next operation
}
```

Example: High-level C-- interpreter



```
int  interp(ast_node *ast,  environment
*env) {
    switch(ast->ntype) {
        case ASSG:
            rhs_val = interp(ast->child1, env);
            update(env, ast->child0, rhs_val);
            return rhs_val;
        case PLUS:
            val0 = interp(ast->child0, env);
            val1 = interp(ast->child1, env);
            return val0 + val1;
    …
        case ID:
            return lookup(env, ast->name);
        case INTCON:
            return ast->value;
    }
    …
}
```

Interpretation: "executing" the AST instead of generating code from it

Interpreters vs. Compilers

Interpreter operation

implemented in software
∴ can be independent of the underlying hardware/OS

```
ip = start of program
while ( not done ) {
    op = current operation at ip
    execute code for op
    advance ip to next operation
}
```

implemented in software
∴ uses additional instructions, incurs a performance overhead

Interpreters vs. Compilers: Portability

Interpreted code is typically more portable than compiled binaries

- easier to make them independent of the underlying hardware and/or OS
- this is especially useful when the underlying hardware/OS is not known ahead of time, e.g.: web applications (Java, JavaScript)
  - Java was originally designed for the digital cable television industry
  - modern web browsers all have built-in support for JavaScript

Interpreters vs. Compilers: Performance

Interpretation incurs a performance overhead relative to compiled code

- some languages (e.g., Python, JavaScript) have significant language-related overheads,* e.g.:

  object allocation and deallocation, name lookups, dynamic type checking, garbage collection, …

- for such languages, the additional % overhead due to interpretation may not be large
  - the benefits of portability and programming convenience then become important

* See: M. Ismail and G. E. Suh, "Quantitative Overhead Analysis for Python", IEEE International Symposium on Workload Characterization, 2018, pages 36-47.

EXERCISE

On your first day at your new job at Acme Widgets Co., your boss puts you in charge of implementing a new in-house programming language. One of your decisions is whether to write an interpreter or a compiler.

How will you figure out which option is better?

- what are the advantages of interpretation compared to compilation?
- what are the disadvantages of interpretation compared to compilation?

Interpreters: Design Considerations

Interpreter Operation

```
ip = start of program     /* ip ≡ instruction pointer */
while ( not done ) {
    op = current operation at ip
    execute code for op on current operands
    advance ip to next operation
}
```

where? *where?*

interpreter design considerations

Interpreter Design Issues

- Encoding the operation

  I.e., getting from the opcode to the code for that operation ("dispatch"):
  - byte code (e.g., JVM)
  - direct threaded code

- Representing operands
  - *register machines*: operations are performed on a fixed set of global locations ("registers")   (e.g.: SPIM);
  - *stack machines*: operations are performed on the top of a stack of operands (e.g.: JVM)

Dispatch

Byte Code

- Operations encoded as small integers
- The interpreter uses the opcode to index into a table of code addresses:

```
while ( TRUE ) {
    byte op = *ip;
    switch ( op ) {
        case ADD: … perform addition; break;
        case SUB: … perform subtraction; break;
         …
    }
}
```

- Compact and portable, but can be slow

Digression: switch statements

Source code

```
switch ( e ) {
    case 12: A;
    case 13: B;
    …
    case 26: W;
}
```

normalize offsets

JmpTbl

0	addr(A)
1	addr(B)
2	addr(C)
	⋮
14	addr(W)

Machine code

```
t1 = evaluate(e)
t1 = t1 - 12
t2 = (t1 < 0 || t1 > 14)
if  t2  goto L_default
t3 = &JmpTbl
t1 = t1 * 8 /* 64-bit */
t3 = t3 + t1
jmp *t3
```

Digression: switch statements

Source code
switch (*e*) {
case 12: A;
case 13: B;
…
case 26: W;
}

JmpTbl

0	addr(A)
1	addr(B)
2	addr(C)
⋮	⋮
14	addr(W)

Machine code	Cost
t1 = evaluate(*e*)	
t1 = t1 – 12	1
t2 = (t1 < 0 || t1 > 14)	1
	1-3
if t2 goto $L_{default}$	1-2
t3 = &JmpTbl	1
t1 = t1 * 8 /* 64-bit */	1
t3 = t3 + t1	6-10
jmp *t3	

TOTAL: 12-20 cycles

Cost of switch-based dispatch

- switch statements are expensive
  - incur a cost of ∼12–20 cycles
- switch-based dispatch instructions incur significant cost:*
  - on a set of benchmark programs, they accounted for < 15% of all executed instructions
  - but accounted for > 50% of the execution time

* M. Anton Ertl an David Gregg. The structure and performance of efficient interpreters. *The Journal of Instruction-Level Parallelism*, Nov. 2003.

Direct Threaded Code

- Indexing through a jump table (as with byte code) is expensive.
- *Idea*: Use the address of the code for an operation as the opcode for that operation.
  - *The opcode may no longer fit in a byte.*

```
while ( TRUE ) {
  addr op = *ip;
  goto *op; /* jump to code for the operation */
}
```

  - More efficient, but the interpreted code is less portable.

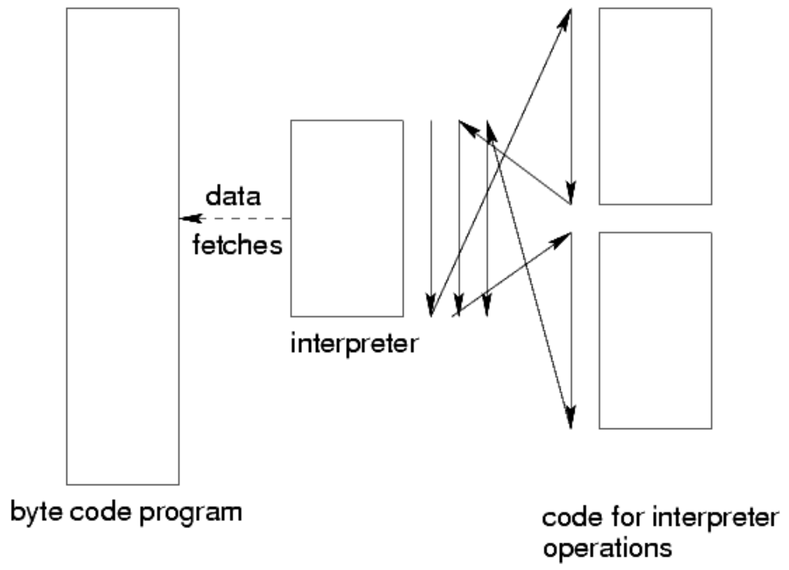
  [James R. Bell. Threaded Code. *Communications of the ACM*, vol. 16 no. 6, June 1973, pp. 370–372]

Byte Code vs. Threaded Code

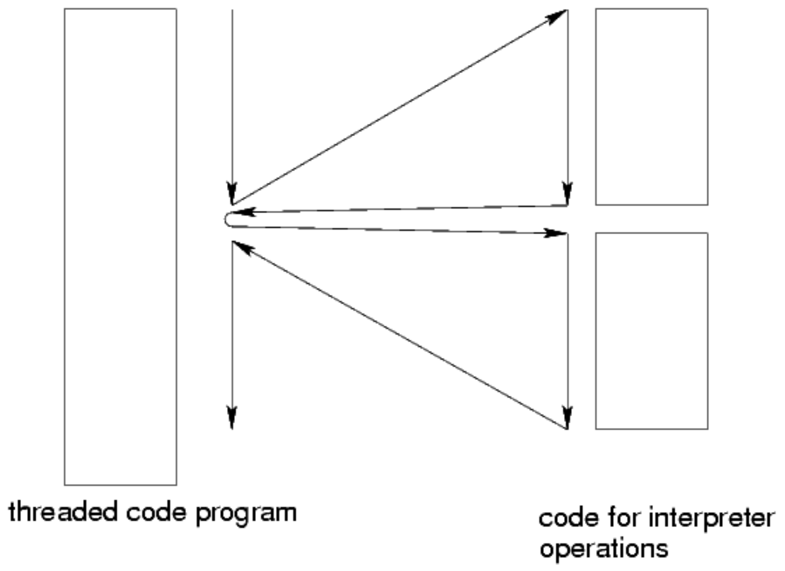
Control flow behavior:

[based on: James R. Bell. Threaded Code. *Communications of the ACM*, vol. 16 no. 6, June 1973, pp. 370–372]

byte code:



threaded code:



Example

Interpreter memory

address	operation
10000	add: …
11000	sub: …
12000	mul: …
13000	div: …
	…

Program ops

add
mul
sub
mul
add
add

byte code

17
23
18
23
17
17

Op Table

index	contents
17	10000
18	11000
23	12000
27	13000

direct threaded

10000
12000
11000
12000
10000
10000

EXERCISE

Interpreter memory	
add	10 000
sub	20 000
mul	30 000
store	40 000
call	50 000
return	60 000

byte-code op table

Index	Contents
0	10 000
1	30 000
2	40 000
3	60 000
4	50 000
5	20 000

program ops

add
mul
sub
add
store
add
return

⇨

byte code

?

EXERCISE

Interpreter memory	
add	10 000
sub	20 000
mul	30 000
store	40 000
call	50 000
return	60 000

byte-code op table

Index	Contents
0	10 000
1	30 000
2	40 000
3	60 000
4	50 000
5	20 000

byte code

1
0
0
1
1
5
2
3

$\Rightarrow$

program ops

?

EXERCISE

Interpreter memory	
add	10 000
sub	20 000
mul	30 000
store	40 000
call	50 000
return	60 000

program ops

add
mul
sub
add
store
add
retur
n

⇨

direct threaded code

?

EXERCISE

Interpreter memory	
add	10 000
sub	20 000
mul	30 000
store	40 000
call	50 000
return	60 000

direct-threaded code

20 000
10 000
40 000
20 000
10 000
10 000
40 000

$\Rightarrow$

program ops

?

Operands

Handling Operands 1: Stack Machines

- Used by Pascal interpreters (’70s and ’80s); resurrected in the Java Virtual Machine.
- *Basic idea*:
  - operands and results are maintained on a stack;
  - operations pop operands off this stack, push the result back on the stack.
- *Advantages*:
  - simplicity
  - compact code.
- *Disadvantages*:
  - code optimization (e.g., utilizing hardware registers effectively) not easy.

Stack Machine Code

- The code for an operation ‘*op* $x_1, \ldots, x_n$’ is:

```
push x_n
…
push x_1
op
```

- *Example*: JVM code for ‘x = 1 – y*2’:

```
iconst 1      /* push the integer constant 1 */
iload y       /* push the value of the integer variable y */
iconst 2
imul          /* after this, stack contains: ‹1, y*2› */
isub
istore x      /* pop stack top, store to integer variable x */
```

Generating Stack Machine Code

Essentially just a post-order traversal of the syntax tree:

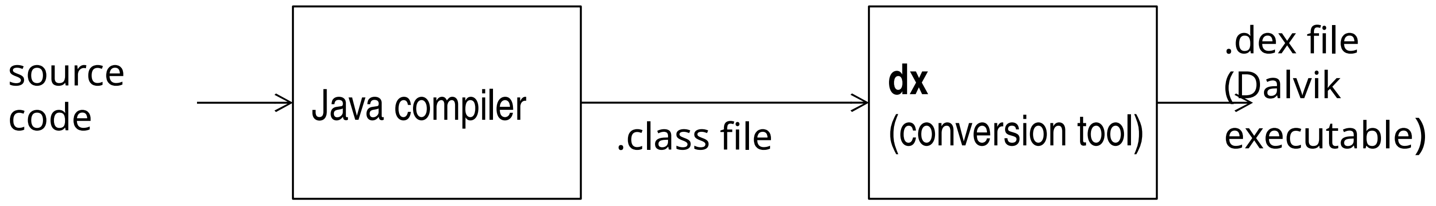
```
void gencode(struct syntaxTreeNode  ⋆tnode )
{
    if ( IsLeaf( tnode ) ) {  ...  }
    else {
      n = tnode→n_operands;
      for (i = n; i > 0; i--) {
        gencode( tnode→operand[i] );        /* traverse children first
  */
     }  /* for */
     gen_instr( opcode_table[tnode→op] );   /* code for the node
  */
   }  /* if [else] */
}
```

Handling Operands 2: Register Machines

- *Basic idea*:
  - Have a fixed set of “virtual machine registers;”
  - Some of these get mapped to hardware registers.
- *Advantages*:
  - potentially better optimization opportunities.
- *Disadvantages*:
  - code is less compact;
  - interpreter becomes more complex (e.g., to decode VM register names).

Register Machine Example: Dalvik VM

Dalvik VM: runs Java on Android phones



- designed for processors constrained in speed, memory
  - relies on underlying Linux kernel for thread support
- register machine
  - depending on the instruction, can access 16, 256, or 64K registers
  - Example:

add-int $\quad$ dstReg_{8bit} $\quad$ $\mathrm{srcRegA}_{8bit}$ $\quad$ $\mathrm{srcRegB}_{8bit}$

Stack Machines vs. Register Machines

- Stack machines are simpler, have smaller byte code files
- Register machines execute fewer VM instructions (e.g., no need to push/pop values)
- Dalvik VM (relative to JVM):
  - executes 47% fewer VM instructions
  - code is 25% larger
    - but instruction fetch cost is only 1.07% more real machine loads per VM instruction executed
    - space savings in the constant pool offset the code size increase

Just-in-Time Compilers (JIT)

Just-in-Time Compilers (JITs)

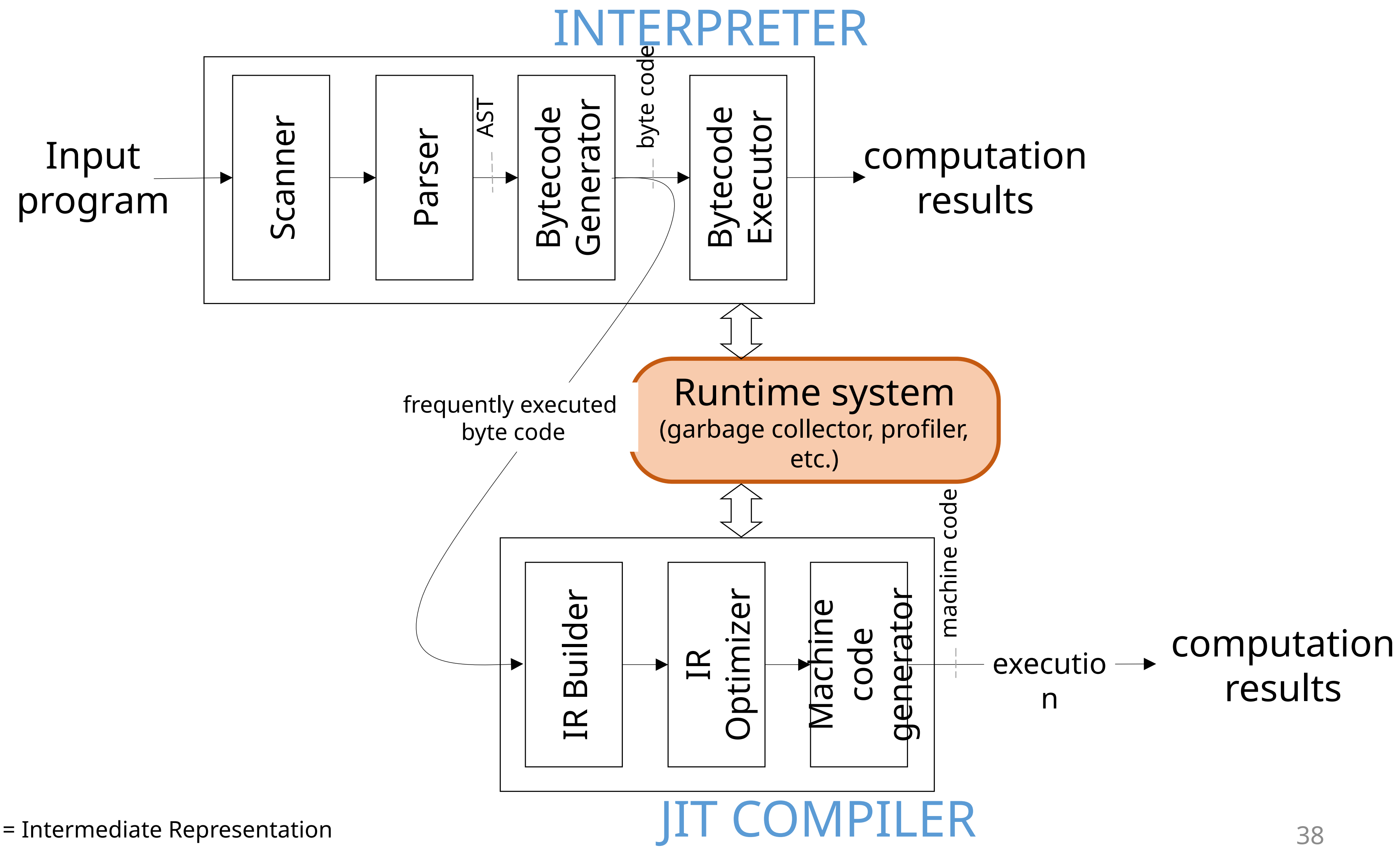
- *Basic idea*: compile byte code to native code during execution.
- *Advantages*:
  - original (interpreted) program remains portable, compact;
  - the executing program runs faster.
- *Disadvantages*:
  - more complex runtime system;
  - performance may degrade if runtime compilation cost exceeds savings.

Improving JIT Effectiveness

- *Reducing Costs*:
  - incur compilation cost only when justifiable;
  - invoke JIT compiler on a per-method basis, at the point when a method is invoked.

- *Improving benefits*:
  - some systems monitor the executing code;
  - methods that are executed repeatedly get optimized further.

Interpreter + JIT Compiler: structure



IR = Intermediate Representation

Method Dispatch: vtables

- vtables ("virtual tables") are a common implementation mechanism for virtual methods in OO languages.
- The implementation of each class contains a vtable for its methods:
  - each virtual method f in the class has an entry in the vtable that gives f's address.
  - each instance of an object gets a pointer to the corresponding vtable.
  - to invoke a (virtual) method, get its address from the vtable.

VMs with JITs

- Each method has vtable entries for:
  - byte code address;
  - native code address.
- Initially, the *native code address* field points to the JIT compiler.
- When a method is first invoked, this automatically calls the JIT compiler.
- The JIT compiler:
  - generates native code from the byte code for the method;
  - patches the *native code address* of the method to point to the newly generated native code (so subsequent calls go directly to native code);
  - jumps to the native code.

JITs: Deciding what to Compile

- For a JIT to improve performance, the benefit of compiling to native code must offset the cost of doing so.

  E.g., JIT compiling infrequently called straight-line code methods can lead to a slowdown!

- We want to JIT-compile only those methods that contain frequently executed ("hot") code:
  - methods that are called a large number of times; or
  - methods containing loops with large iteration counts.

JITs: Deciding what to Compile

- Identifying frequently called methods:
  - count the number of times each method is invoked;
  - if the count exceeds a threshold, invoke JIT compiler.
  - (In practice, start the count at the threshold value and count down to 0: this is slightly more efficient.)
- Identifying hot loops:
  - modify the interpreter to “snoop” the loop iteration count when it finds a loop, using simple bytecode pattern matching.
  - use the iteration count to adjust the amount by which the invocation count for the method is decremented on each call.

Typical JIT Optimizations

Choose optimizations that are cheap to perform and likely to improve performance, e.g.:

- inlining frequently called methods:

  consider both code size and call frequency in inlining decision.
- exception check elimination:

  eliminate redundant null pointer checks, array bounds checks.
- common subexpression elimination:

  avoid address recomputation, reduce the overhead of array and instance variable access.
- simple, fast register allocation.

Summary

Summary

- Interpreters are common in modern computing environments
- Design issues (⇒ portability/performance tradeoffs):
  - operation encoding (dispatch)
  - operands (stack vs. registers)
- JIT compilers
  - create optimized code on the fly
  - usually based on identifying “hot” code at runtime
  - typically use cheap + effective optimizations