CSc 520

Principles of Programming Languages

3 : Interpreters

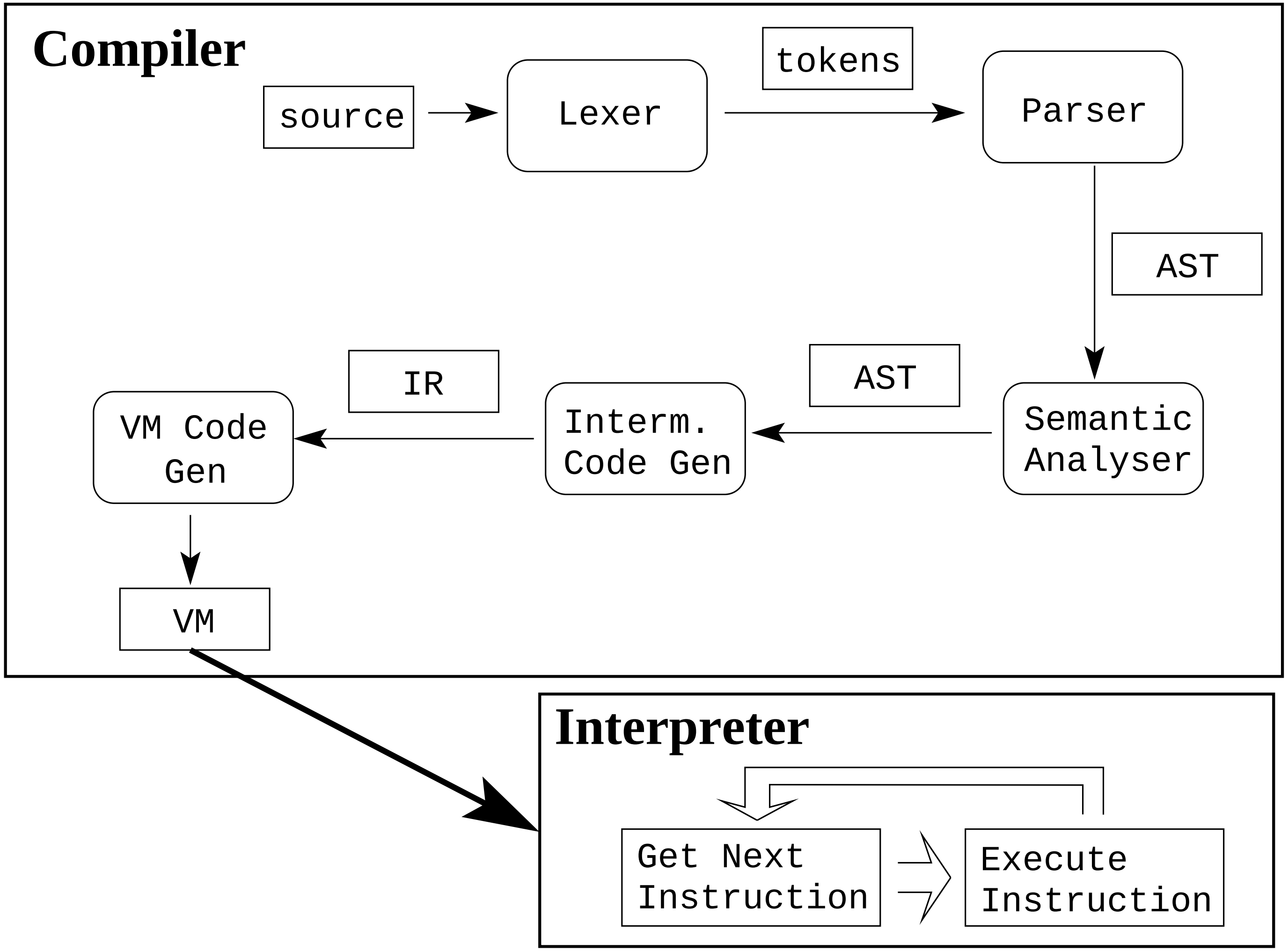
Christian Collberg

collberg+520@gmail.com

Department of Computer Science

University of Arizona

Compiler Phases



Interpretation

- An interpreter is like a CPU, only in software.
- The compiler generates *virtual machine* (VM) code rather than native machine code.
- The interpreter executes VM instructions rather than native machine code.

Interpreters are

slow Often 10–100 times slower than executing machine code directly.

portable The virtual machine code is not tied to any particular architecture.

Interpretation. . .

Interpreters are

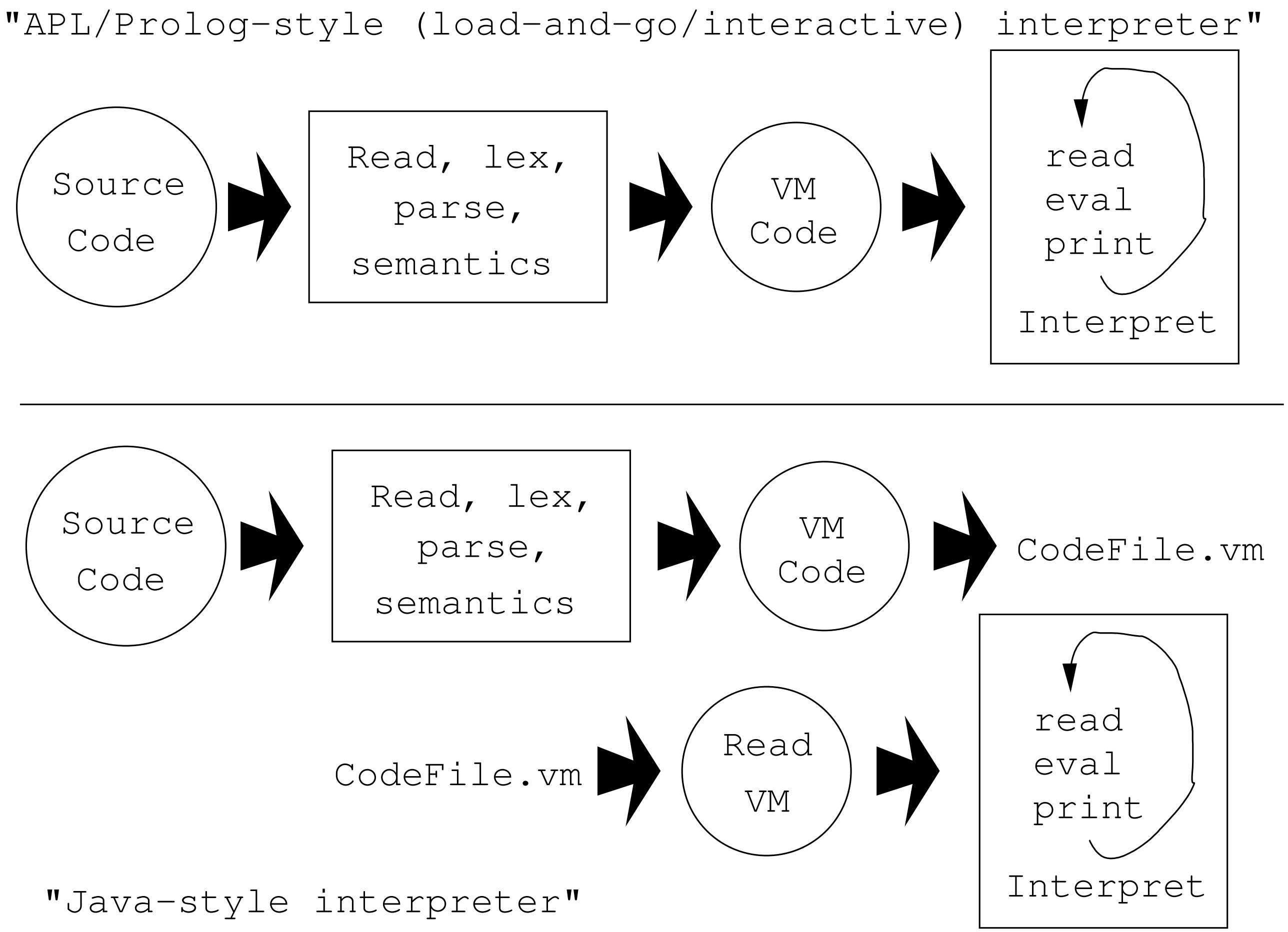
slow Often 10–100 times slower than executing machine code directly.

portable The virtual machine code is not tied to any particular architecture.

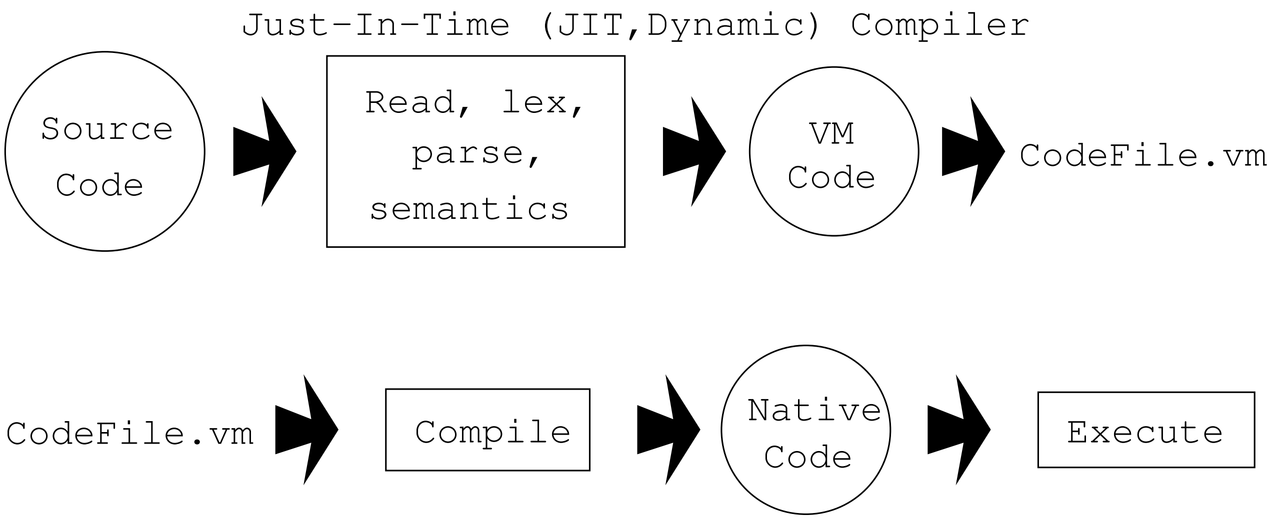
Interpreters work well with

very high-level, dynamic languages (APL,Prolog,ICON) where a lot is unknown at compile-time (array bounds, etc).

Kinds of Interpreters



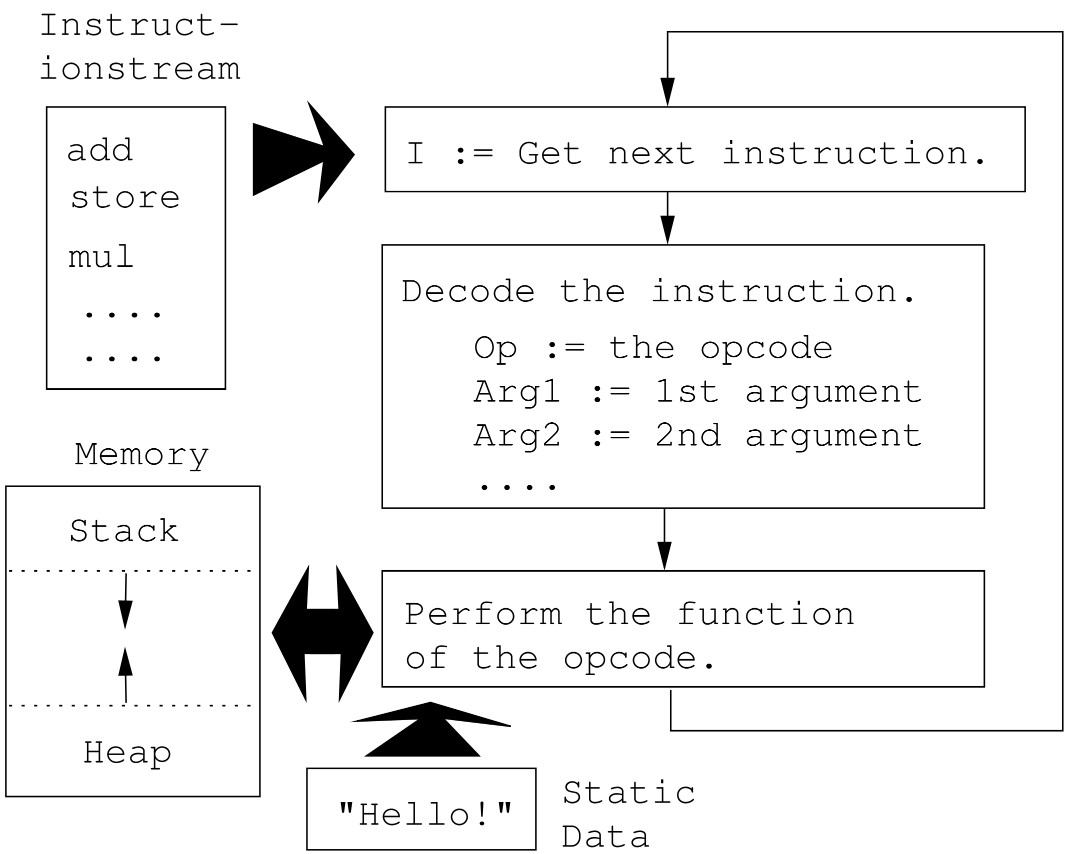
Kinds of Interpreters. . .



Actions in an Interpreter

- Internally, an interpreter consists of
  1. The interpreter *engine*, which executes the VM instructions.
  2. *Memory* for storing user data. Often separated as a heap and a stack.
  3. A stream of VM instructions.

Actions in an Interpreter. . .



Stack-Based Instruction Sets

- Many virtual machine instruction sets (e.g. Java bytecode, Forth) are *stack based*.

  add pop the two top elements off the stack, add them together, and push the result on the stack.

  push X push the value of variable X.

  pusha X push the address of variable X.

  store pop a value V, and an address A off the stack. Store V at memory address A.

Stack-Based Instruction Sets. . .

- Here's an example of a small program and the corresponding stack code:

Source Code	VM Code
`VAR X,Y,Z : INTEGER;`	`pusha X`
`BEGIN`	`push Y`
`  X := Y + Z;`	`push Z`
`END;`	`add`
	`store`

Register-Based Instruction Sets

- Stack codes are *compact*. If we don't worry about code size, we can use any intermediate code (tuples, trees). Example: RISC-like VM code with ∞ number of virtual registers $R_1, \cdots$:

add R_1, R_2, R_3 Add VM registers R_2 and R_3 and store in VM register R_1.

load R_1, X R_1 := value of variable X.

loada R_1, X R_1 := address of variable X.

store R_1, R_2 Store value R_2 at address R_1.

Register-Based Instruction Sets...

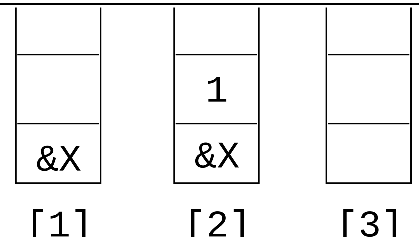
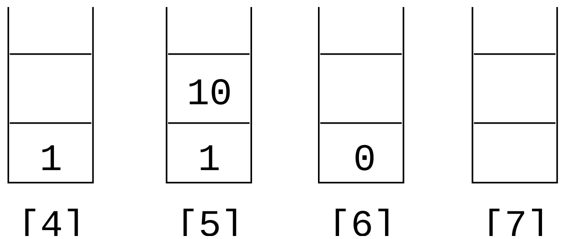
- Here's an example of a small program and the corresponding register code:

Source Code	VM Code
`VAR X,Y,Z : INTEGER;`	`load` R_1, Y
`BEGIN`	`load` R_2, Z
`  X := Y + Z;`	`add` R_3, R_1, R_2
`END;`	`loada` R_4, X
	`store` R_4, R_3

Stack Machine Example I

Source Code	VM Code
`VAR X,Y,Z : INTEGER;`	`[1] pusha X`
`BEGIN`	`[2] push 1`
`X := 1;`	`[3] store`
`WHILE X < 10 DO`	`[4] push X`
	`[5] push 10`
	`[6] GE`
	`[7] BrTrue 14`
`X := Y + Z;`	`[8] pusha X`
	`[9] push Y`
	`[10] push Z`
	`[11] add`
`ENDDO`	`[12] store`
`END;`	`[13] jump 4`

Stack Machine Example (a)

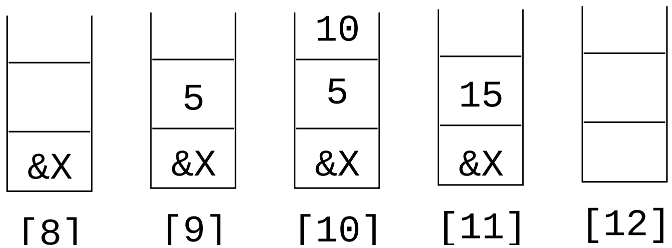
VM Code	Stack	Memory
[1] pusha X [2] push 1 [3] store		X 1 Y 5 Z 10
[4] push X [5] push 10 [6] GE [7] BrTrue 14		X 1 Y 5 Z 10

Stack Machine Example (b)

VM Code

```
[8]  pusha X
[9]  push Y
[10] push Z
[11] add
[12] store
```

Stack



Memory

X	15
Y	5
Z	10

```
[13] jump 4
```

Switch Threading

Switch Threading

- Instructions are stored as an array of integer tokens. A switch selects the right code for each instruction.

```
typedef enum {add,load,store,···} Inst;
void engine () {
   static Inst prog[] = {load,add,···};
   Inst *pc = &prog;
   int Stack[100]; int sp = 0;
   for (;;)
     switch (*pc++) {
       case add:  Stack[sp-1]=Stack[sp-1]+Stack[sp];
                  sp--; break;
   }}}
```

Switch Threading in Java

- Let's look at a simple Java switch interpreter.
- We have a stack of integers `stack` and a stack pointer `sp`.
- There's an array of bytecodes `prog` and a program counter `pc`.
- There is a small memory area `memory`, an array of 256 integers, numbered 0–255. The `LOAD`, `STORE`, `ALOAD`, and `ASTORE` instructions access these memory cells.

Bytecode semantics

mnemonic	opcode	stack-pre	stack-post	side-effects
ADD	0	[A,B]	[A+B]	
SUB	1	[A,B]	[A-B]	
MUL	2	[A,B]	[A*B]	
DIV	3	[A,B]	[A-B]	
LOAD X	4	[]	[Memory[X]]	
STORE X	5	[A]	[]	Memory[X] = A
PUSHB X	6	[]	[X]	
PRINT	7	[A]	[]	Print A
PRINTLN	8	[]	[]	Print a newline
EXIT	9	[]	[]	The interpreter exits
PUSHW X	11	[]	[X]	

Bytecode semantics. . .

mnemonic	opcode	stack-pre	stack-post	side-effects
BEQ L	12	[A,B]	[]	if A=B then PC+=L
BNE L	13	[A,B]	[]	if A!=B then PC+=L
BLT L	14	[A,B]	[]	if A<B then PC+=L
BGT L	15	[A,B]	[]	if A>B then PC+=L
BLE L	16	[A,B]	[]	if A<=B then PC+=L
BGE L	17	[A,B]	[]	if A>=B then PC+=L
BRA L	18	[]	[]	PC+=L
ALOAD	19	[X]	[Memory[X]]	
ASTORE	20	[A,X]	[]	Memory[X] = A
SWAP	21	[A,B]	[B,A]	

Example programs

This program prints a newline character and then exits:

```
PRINTLN
EXIT
```

Or, in binary: $\langle 8, 9 \rangle$

This program prints the number 10, then a newline character, and then exits:

```
PUSHB 10
PRINT
PRINTLN
EXIT
```

Or, in binary: $\langle 6, 10, 7, 8, 9 \rangle$

Example programs...

This program pushes two values on the stack, then performs an `ADD` instruction which pops these two values off the stack, adds them, and pushes the result. `PRINT` then pops this value off the stack and prints it:

```
PUSHB 10
PUSHB 20
ADD
PRINT
PRINTLN
EXIT
```

Or, in binary: $\langle 6, 10, 6, 20, 0, 7, 8, 9 \rangle$

Example program...

This program uses the `LOAD` and `STORE` instructions to store a value in memory cell number 7:

```
PUSHB 10
STORE 7
PUSHB 10
LOAD 7
MUL
PRINT
PRINTLN
EXIT
```

Or, in binary: $\langle 6, 10, 5, 7, 6, 10, 4, 7, 2, 7, 8, 9 \rangle$

Example programs...

```
# Print the numbers 1 through 9.
# i = 1; while (i < 10) do {print i; println; i++;}
PUSHB 1       # mem[1] = 1;
STORE 1
LOAD 1        # if mem[1] < 10 goto exit
PUSHB 10
BGE
LOAD 1        # print mem[i] value
PRINT
PRINTLN
PUSHB 1       # mem[1]++
LOAD 1
ADD
STORE 1
BRA           # goto top of loop
EXIT
```

Bytecode Description

`ADD`: Pop the two top integers A and B off the stack, then push $A + B$.

`SUB`: As above, but push $A - B$.

`MUL`: As above, but push $A * B$.

`DIV`: As above, but push A/B.

`PUSHB` X: Push X, a signed, byte-size, value, on the stack.

`PUSHW` X: Push X, a signed, word-size, value, on the stack.

`PRINT`: Pop the top integer off the stack and print it.

`PRINTLN`: Print a newline character.

`EXIT`: Exit the interpreter.

Bytecode Description...

`LOAD` X: Push the contents of memory cell number X on the stack.

`STORE` X: Pop the top integer off the stack and store this value in memory cell number X.

`ALOAD`: Pop the address of memory cell number X off the stack and push the value of X.

`ASTORE`: Pop the address of memory cell number X and the value V off the stack and store the V in X.

`SWAP`: Exchange the two top elements on the stack.

Bytecode Description. . .

`BEQ` L: Pop the two top integers A and B off the stack, if $A == B$ then continue with instruction $\texttt{PC} + L$, where PC is address of the instruction *following* this one. Otherwise, continue with the next instruction.

`BNE` L: As above, but branch if $A \neq B$.

`BLT` L: As above, but branch if $A < B$.

`BGT` L: As above, but branch if $A > B$.

`BLE` L: As above, but branch if $A \leq B$.

`BGE` L: As above, but branch if $A \geq B$.

`BRA` L: Continue with instruction $\texttt{PC} + L$, where PC is the address of the instruction *following* this one.

Switch Threading in Java

```
public class Interpreter {
   static final byte ADD     = 0;
   static final byte SUB     = 1;
   static final byte MUL     = 2;
   static final byte DIV     = 3;
   static final byte LOAD    = 4;
   static final byte STORE   = 5;
   static final byte PUSHB   = 6;
   static final byte PRINT   = 7;
   static final byte PRINTLN= 8;
   static final byte EXIT    = 9;
   static final byte PUSHW   = 11;
```

```
static final byte BEQ    = 12;
static final byte BNE    = 13;
static final byte BLT    = 14;
static final byte BGT    = 15;
static final byte BLE    = 16;
static final byte BGE    = 17;
static final byte BRA    = 18;
static final byte ALOAD  = 19;
static final byte ASTORE = 20;
static final byte SWAP   = 21;
```

```
static void interpret (byte[] prog)
       throws Exception {
    int[] stack = new int[100];
    int[] memory = new int[256];
    int pc = 0;
    int sp = 0;
    while (true) {
    switch (prog[pc]) {
       case ADD    : {
           stack[sp-2]+=stack[sp-1]; sp--;
           pc++; break;
         }
       /* Same for SUB, MUL, DIV. */
```

```
case LOAD   : {
  stack[sp] = memory[(int)prog[pc+1]];
  sp++; pc+=2; break;}

case STORE  : {
  memory[prog[pc+1]] = stack[sp-1];
  sp-=1; pc+=2; break;}

case ALOAD   : {
  stack[sp-1] = memory[stack[sp-1]];
  pc++; break;}

 case ASTORE  : {
   memory[stack[sp-1]] = stack[sp-2];
   sp-=2; pc++; break;}
```

```
case SWAP : {
   int tmp = stack[sp-1];
   stack[sp-1] = stack[sp-2];
   stack[sp-2]=tmp;
   pc++; break; }

 case PUSHB  : {
   stack[sp] = (int)prog[pc+1];
   sp++; pc+=2; break; }
 /* Similar for PUSHW. */

 case PRINT  : {
   System.out.print(stack[--sp]);
   pc++; break; }
```

```
        case PRINTLN: {
          System.out.println(); pc++; break; }

        case EXIT : {return;}

        case BEQ  : {/*Same for BNE,BLT,...*/
          pc+= (stack[sp-2]==stack[sp-1])?
                    2+(int)prog[pc+1]:2;
          sp-=2; break; }

        case BRA  : {
           pc+= 2+(int)prog[pc+1]; break; }

        default : throw new Exception("Illegal")
}}}}
```

Switch Threading...

- Switch (case) statements are implemented as indirect jumps through an array of label addresses (a *jump-table*). Every switch does 1 range check, 1 table lookup, and 1 jump.

```
switch (e) {
 case 1:  S1; break;
 case 3:  S2; break;
 default:  S3;
}
```

$\Rightarrow$

```
JumpTab = {0,&Lab1,&Lab3,&Lab2};
if ((e < 1) || (e > 3)) goto Lab3;
goto *JumpTab[e];
Lab1:  S1; goto Lab4;
Lab2:  S2; goto Lab4;
Lab3:  S3;
Lab4:
```

Faster Operator Dispatch

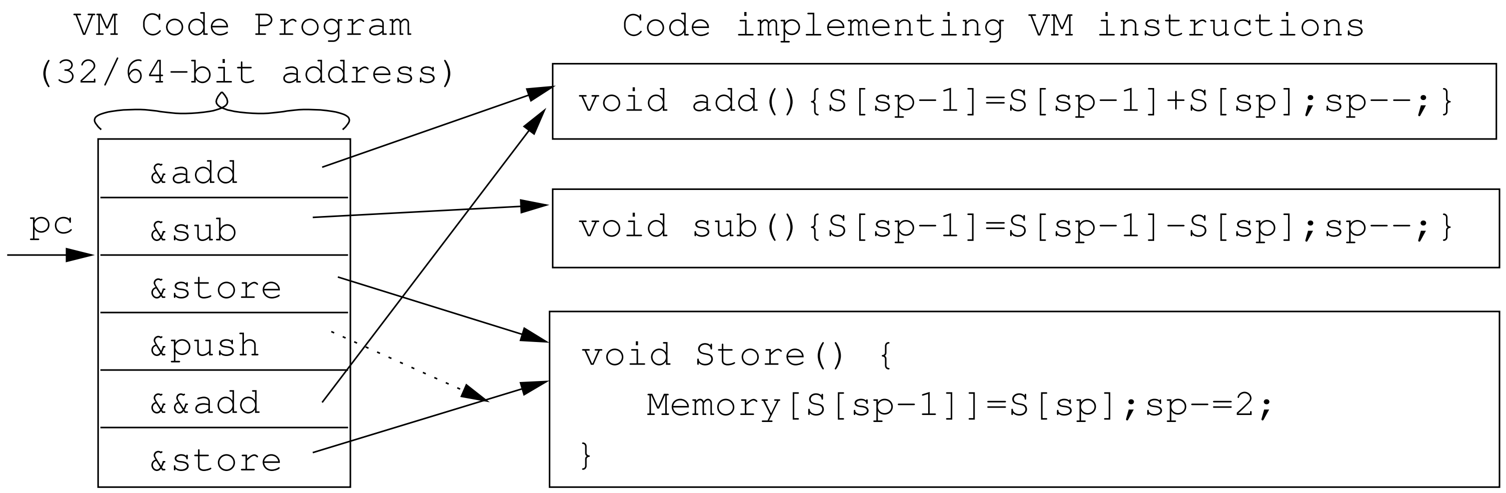
Direct Call Threading

- Every instruction is a separate function.
- The program `prog` is an array of pointers to these functions.
- I.e. the `add` instruction is represented as the address of the `add` function.
- `pc` is a pointer to the current instruction in `prog`.
- `(*pc++)()` jumps to the function that `pc` points to, then increments `pc` to point to the next instruction.
- Hard to implement in Java.

Direct Call Threading. . .

```
typedef void (* Inst)();
Inst prog[] = {&load,&add,···};

Inst *pc = &prog;
int Stack[100]; int sp = 0;

void add(); {
   Stack[sp-1]=Stack[sp-1]+Stack[sp];
   sp--;}

void engine () {
   for (;;) (*pc++)()
}
```

Direct Call Threading. . .



Direct Call Threading...

- In direct call threading all instructions are in their own functions.
- This means that VM registers (such as `pc`, `sp`) must be in global variables.
- So, every time we access `pc` or `sp` we have to load them from global memory. $\Rightarrow$ Slow.
- With the switch method `pc` and `sp` are local variables. Most compilers will keep them in registers. $\Rightarrow$ Faster.
- Also, a direct call threaded program will be large since each instruction is represented as a 32/64-bit address.
- Also, overhead from call/return sequence.

Direct Threading

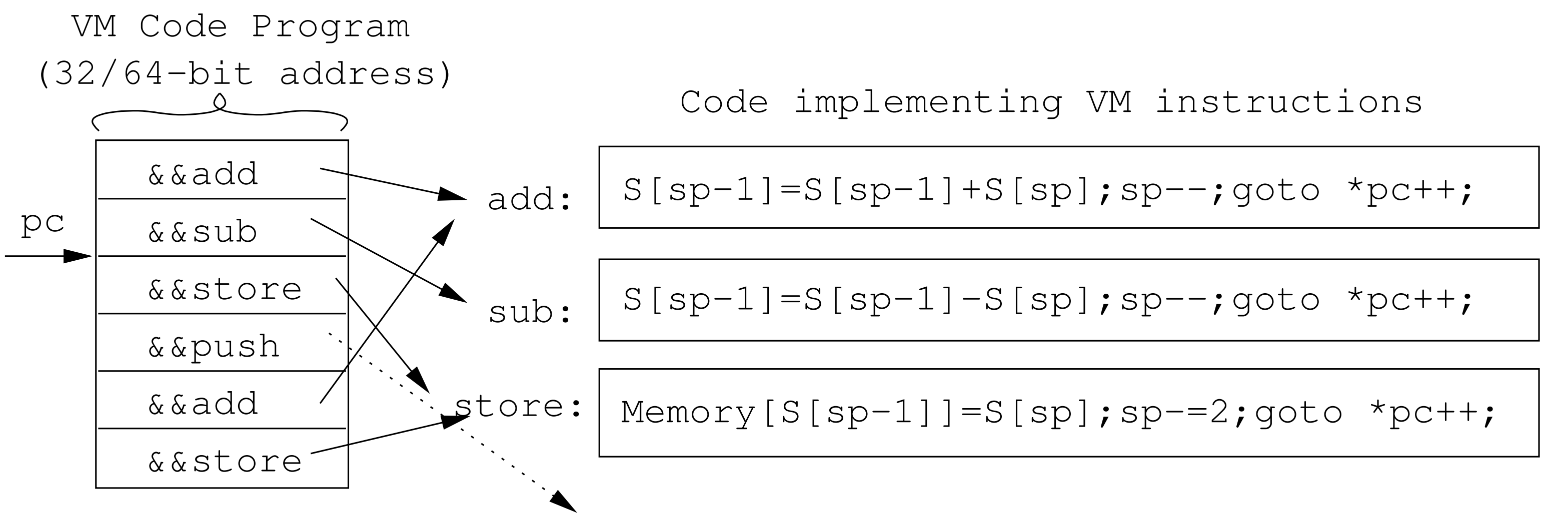
- Each instruction is represented by the address (label) of the code that implements it.
- At the end of each piece of code is an indirect jump `goto *pc++` to the next instruction.
- `"&&"` takes the address of a label. `goto *V` jumps to the label whose address is stored in variable V. This is a `gcc` extensions to C.

Direct Threading. . .

```
typedef void *Inst
static Inst prog[]={&&add,&&sub,···};

void engine() {
   Inst *pc = &prog;
   int Stack[100]; int sp=0;
   goto **pc++;

   add:  Stack[sp-1]+=Stack[sp]; sp--; goto **pc++;

   sub:  Stack[sp-1]-=Stack[sp]; sp--; goto **pc++;
}
```

Direct Threading. . .

- Direct threading is the most efficient method for instruction dispatch.



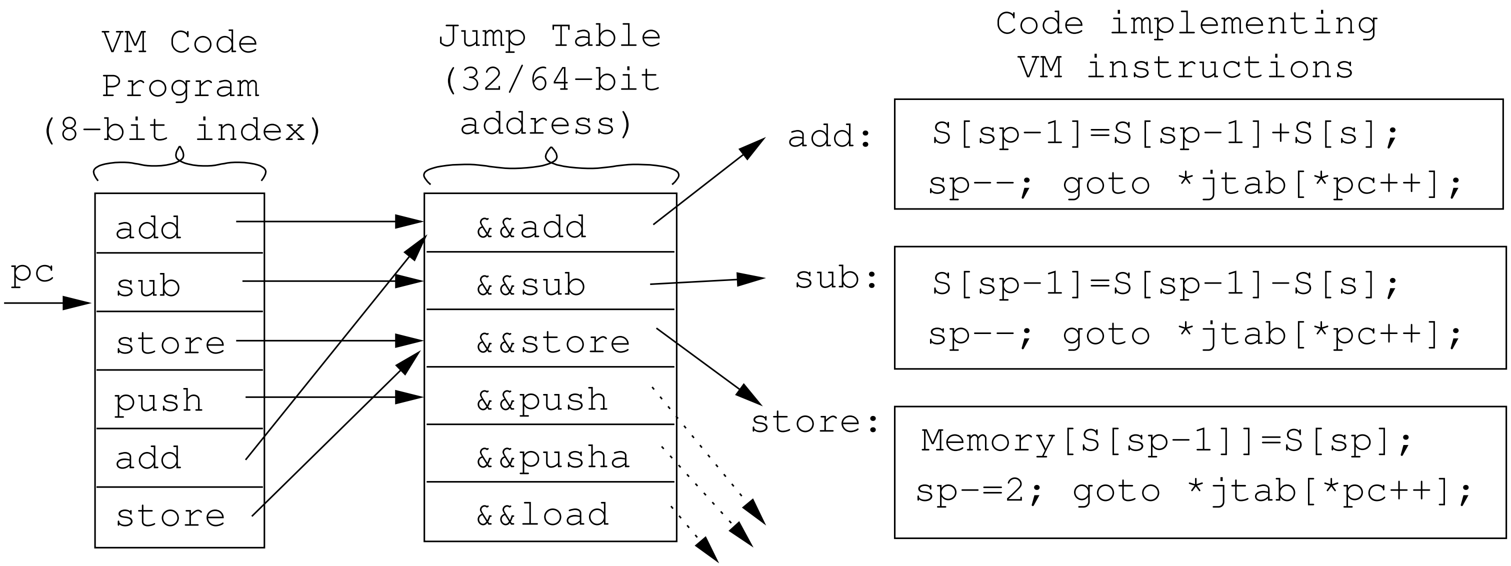
Indirect Threading

- Unfortunately, a direct threaded program will be large since each instruction is an address (32 or 64 bits).
- At the cost of an extra indirection, we can use byte-code instructions instead.
- `prog` is an array of bytes.
- `jtab` is an array of addresses of instructions.
- `goto *jtab[*pc++]` finds the current instruction (what `pc` points to), uses this to index `jtab` to get the address of the instruction, jumps to this code, and finally increments `pc`.

Indirect Threading. . .

```
typedef enum {add,load,···} Inst;
typedef void *Addr;
static Inst prog[]={add,sub,···};

void engine() {
   static Addr jtab[]= {&&add,&&load,···};
   Inst *pc = &prog;
   int Stack[100]; int sp=0;
   goto *jtab[*pc++];

   add:  Stack[sp-1]+=Stack[sp]; sp--;
         goto *jtab[*pc++];
}
```

Indirect Threading. . .



Other Optimizations

Minimizing Stack Accesses

- To reduce the cost of stack manipulation we can keep one or more of the *Top-Of-Stack* elements in registers.
- In the example below, `TOS` holds the top stack element. `Stack[sp]` holds the element second to the top, etc.

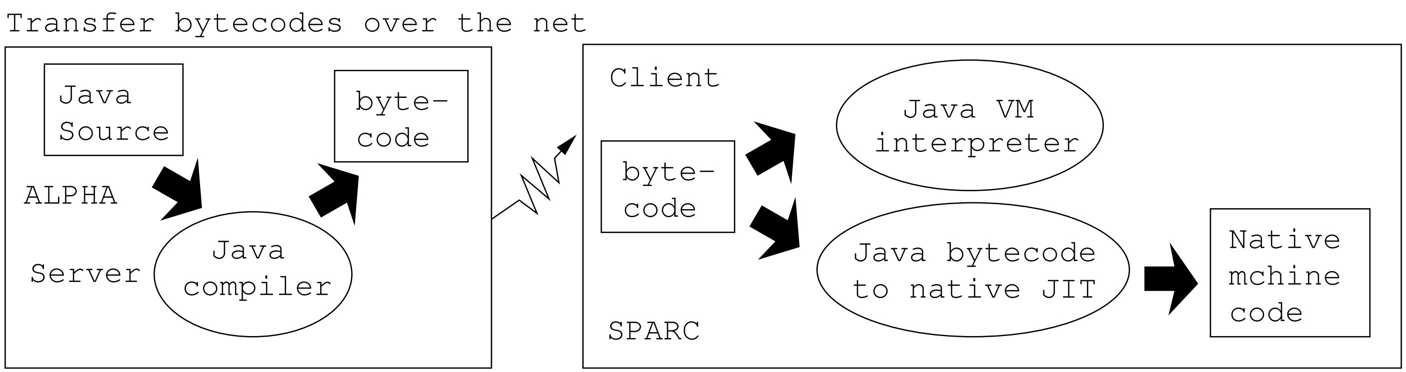
```
void engine() {
  static Inst prog[]={&&add,&&store,···};
  Inst *pc = &prog; int sp; register int TOS;
  goto *pc++;
  add:   TOS+=Stack[sp]; sp--; goto *pc++;
  store: Memory[Stack[sp]]=TOS; TOS=Stack[sp-1]
         sp-=2; goto *pc++;
}
```

Instruction Sets Revisited

- We can (sometimes) speed up the interpreter by being clever when we design the VM instruction set:
  1. Combine often used code sequences into one instruction. E.g. `muladd` a, b, c, d for $a := b * c + d$. This will reduce the number of instructions executed, but will make the VM engine larger.
  2. Reduce the total number of instructions, by making them simple and RISC-like. This will increase the number of instructions executed, but will make the VM engine smaller.
- A small VM engine may fit better in the cache than a large one, and hence yield better overall performance.

Just-In-Time Compilation

- Used to be called *Dynamic Compilation* before the marketing department got their hands on it. Also a verb, *jitting*.
- The VM code is compiled to native code just prior to execution. Gives machine independence (the bytecode can be sent over the net) and speed.
- When? When a class/module is loaded? The first time a method/procedure is called? The 2nd time it's called?



Readings and References

- Louden, pp. 4–5.
- M. Anton Ertl, *Stack Caching for Interpreters*, ACM Programming Language Design and Implementation (PLDI'95), 1995, pp. 315–318.

  `http://www.complang.tuwien.ac.at/papers/ertl94sc.ps.Z`
- Todd Proebsting, *Optimizing an ANSI C Interpreter with Superoperators*, ACM Principles of Programming Languages (POPL'96), January 1996, pp. 322–332.

  `http://www.acm.org/pubs/articles/proceedings/plan/199448/p322-proebs`
- P. Klint, Interpretation Techniques, Software — Practice & Experience, 11(9) 1981, 963–973.

Summary

- Direct threading is the most efficient dispatch method. It cannot be implemented in ANSI C. Gnu C's "labels as values" do the trick.
- Indirect threading is almost as fast as direct threading. It may sometimes even be faster, since the interpreted program is smaller and may hence fits better in the cache.
- Call threading is the slowest method. There is overhead from the jump, save/restore of registers, the return, as well as the fact that VM registers have to be global.

Summary…

- Switch threading is slow but has the advantage to work in all languages with a case statement.
- The interpretation overhead consists of *dispatch overhead* (the cost of fetching, decoding, and starting the next instruction) and *argument access overhead*.
- You can get rid of some of the argument access overhead by *caching* the top k elements of the stack in registers. See Ertl's article.
- Jitting is difficult on machines with separate data and code caches. We must generate code into the data cache, then do a *cache flush*, then jump into the new code. Without the flush we'd be loading the old data into the code cache!