Bytecodes and bytecode interpretation

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

- key aspect

  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

Abstract machines

- Perhaps a better name than virtual machine?

abstract: adj. existing in thought or as an idea not having a physical or concrete existence

Examples:

- Landin's SECD machine for Lambda Calculus
- Warren Abstract Machine for Prolog
- ..and of course, the Turing machine

Details, details

- However, some details, while irrelevant to the semantics of the guest language, are pragmatically essential
  - Examples: instruction encodings, for binary distribution and inter-operability
- A “concrete abstract machine”? The term *virtual machine* has stuck.

virtual: adj. not physically existing as such but made by software to appear to do so

Expression machine encodings

3 bits → opcode

5 bits → operand

Opcode	OP (7:5)	ARG (4:0)	Other bytes
LIT	0	unused	4 bytes for literal
ADDop	1	unused	
SUBop	2	unused	
MULop	3	unused	
DIVop	4	unused	
GET	5	var (0-25)	
PUT	6	var (0-25)	
END	7	unused	

read →

write →

Compiling expressions to bytecode

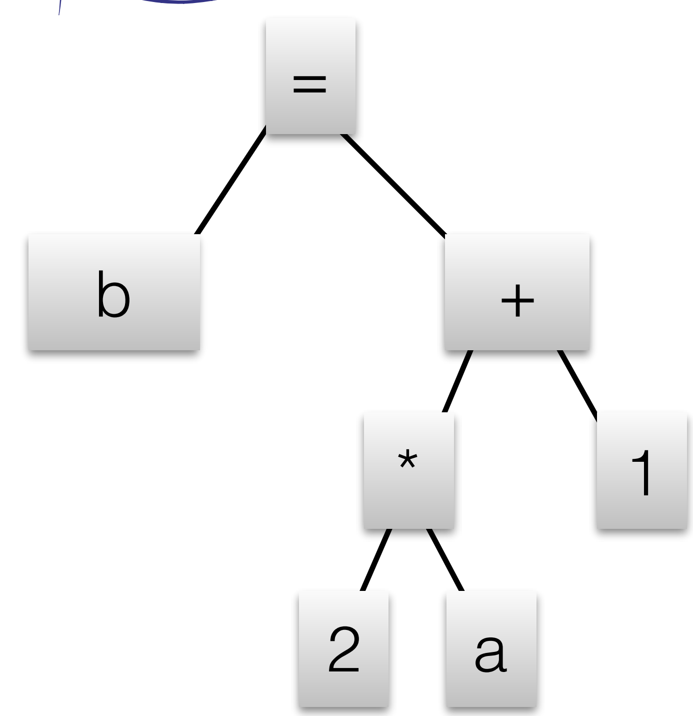
- Reverse Polish Notation (*a OP b* becomes *a b OP)*

Example: b = 2*a + 1 becomes 2 a * 1 + *b=*
where *b=* is a single assignment operation. An alternative would be to push the address of b and then do a store op:
2 a * 1 + &b ←
(exposes addresses as a type). Another would be to use a symbolic form:
2 a * 1 + ‘b ←

This form is easily generated by a mostly left-to-right, depth-first walk of the AST.

- Transformation to stack machine ops is then simple:

```
LIT 2     // push 2 onto stack
GET a     // push a onto stack
MULop     // multiply top two stack items, replace with result
LIT 1     // push 1
ADDop     // add top two stack items, replace with result
PUT b     // pop top of stack into b
```



Compiling expressions to bytecode

```
Output emit(Exp *e, Output p) {
  switch (e->tag) {
  case  CONST: return emitWord(e->u.val,
                                 emitByte(o(LIT), p));
  case    VAR: return emitByte(o(GET) | e->u.var, p);
  case    ADD: return
                  emitByte(o(ADDop),
                    emit(e->u.exp.r, emit(e->u.exp.l, p)));
  /* ditto SUB, MUL, DIV */
  case ASSIGN: return emitByte(o(PUT) | e->u.assgn.var,
                                 emit(e->u.assgn.rhs));
  }
}
```

where emitByte, emitWord add a byte, word, resp., to the output and return the output

Expression bytecode interpreter

```
#define o(OP) (OP<<5)
#define L5(OP) (OP&0x1F)
#define H3(OP) (OP&0xE0)


while (1) {
  switch (H3(*pc)) {
  case o(LIT)  : *++sp= ((((pc[1]<<8)+pc[2])<<8)+pc[3])<<8)+pc[4];
                 pc += 5;
                 continue;
  case o(ADDop): sp[-1] += sp[0]; --sp; ++pc; continue;
  case o(SUBop): sp[-1] -= sp[0]; --sp; ++pc; continue;
  case o(MULop): sp[-1] *= sp[0]; --sp; ++pc; continue;
  case o(DIVop): sp[-1] /= sp[0]; --sp; ++pc; continue;
  case o(GET)  : *++sp= vars[L5(*pc)];  ++pc; continue;
  case o(PUT)  : vars[L5(*pc)]= *sp--;  ++pc; continue;
  case o(END)  : goto out;
  }
}
out:
```

Expression bytecode interpreter — more like how it *should* look

```
while (1) {
  switch (H3(*pc)) {
  case o(LIT)  : push(get4(pc+1));
                 inc_pc(5);
                 continue;
  case o(ADDop): push(pop()+pop());    inc_pc(); continue;
  …
  case o(GET)  : push(vars[L5(*pc)]);  inc_pc(); continue;
  case o(PUT)  : vars[L5(*pc)]= pop(); inc_pc(); continue;
  case o(END)  : goto out;
  }
}
out:
```

Adding simple control flow to the language

- Let’s consider what happens if we add simple conditionals and loops:
  if x == y then z else w
  while $\boxed{x < y}$ do x = x + 1 end
  (Comparisons only appear in control expressions.)

Implementing simple control flow

- Add branch bytecodes, e.g., BEQ, BLT, etc., which pop-and-compare top two stack items and then branch by a given displacement if test succeeds.
- Also need unconditional branch, B (no pops).
- Note that the PC cannot be accessed by arithmetic opcodes.

protect wasm

Compiled examples

if x == y then z else w

```
   GET x
   GET y
   BEQ j
   GET w
   B k
j: GET z
k:
```

Compiled examples

while x < y do x = x + 1 end

```
s:GET x
  GET y
  BGE e
  GET x
  LIT 1
  ADD
  PUT x
  B s
e:
```

Implementing branches

- BEQ implementation; branch displacement is encoded into the instruction (re-encoding required; this code uses 12-bit displacements).

```
case o(BEQ):
  op1= *--sp; op2= *--sp;
  if (op1==op2) pc += (L4(*pc)<<8)+pc[1]+2;
  else pc += 2;
  continue;
```

- Unconditional branch:

```
case o(B):
  pc += (L4(*pc)<<8)+pc[1]+2;
  continue;
```

Adding functions

- Let's use upper case for function names, to keep functions separate for values:

```
F(x,y) {x*x + y*y}
M(a,b) {if a > b then a else b}
T(x,y,z) {
  if y < x then T(T(x-1,y,z),T(y-1,z,x),T(z-1,x,y))
  else z}
```

CALL bytecode

Can involve more static information

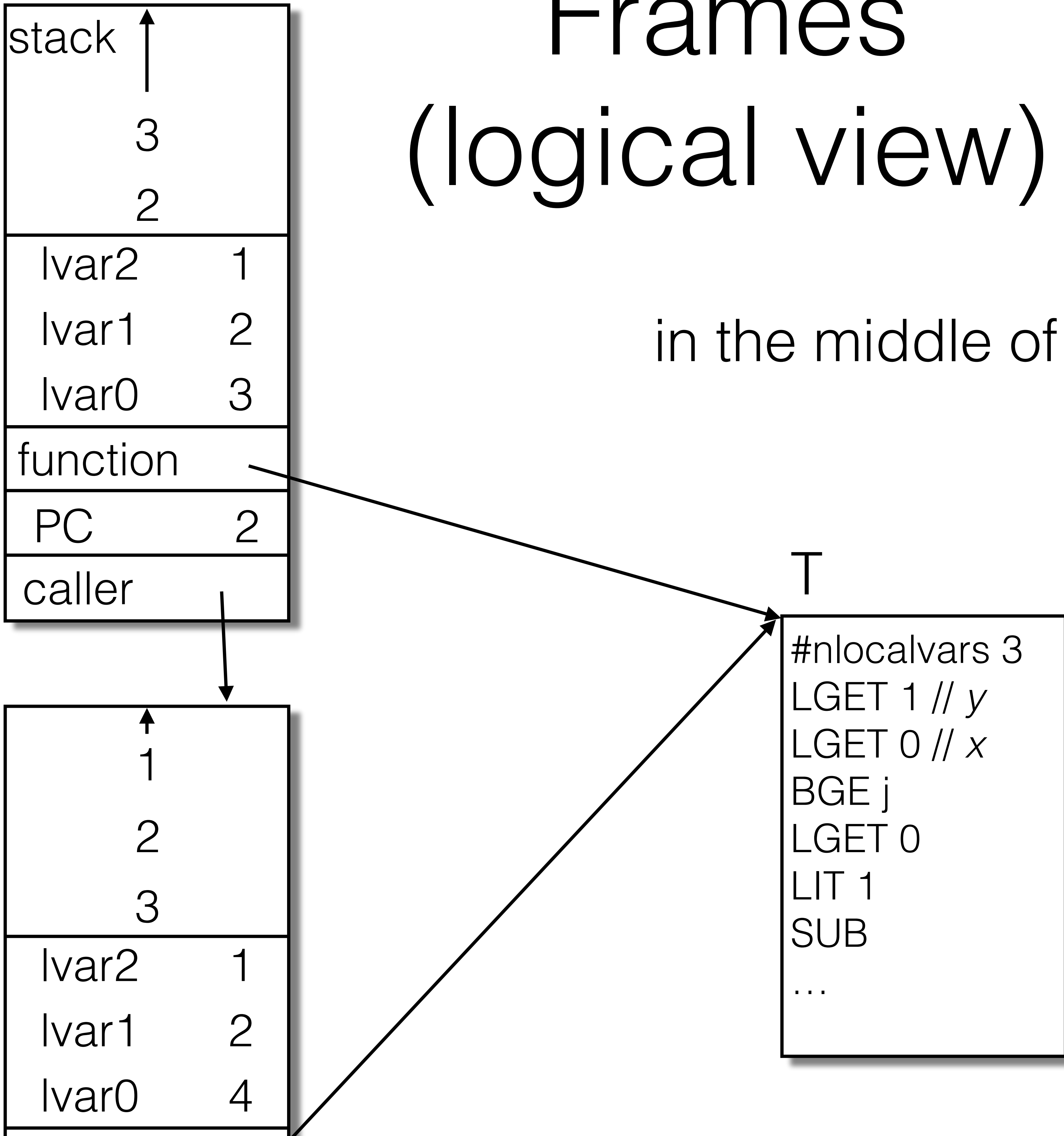
- An abstract representation would not reify the PC or stack, e.g.:
  CALL$\boxed{\textbf{n}}$ F (*n* part of opcode) calls F passing the top *n* items from the stack as arguments; they are replaced with the result.

- The called function runs in a separate frame with its own stack. The return link is implicit.

- The frame needs to store the arguments; let's used an indexed subarray, and LGET and LPUT to access them.

Frames (logical view)

in the middle of a call to T(4,2,1)



Function representation in bytecodes

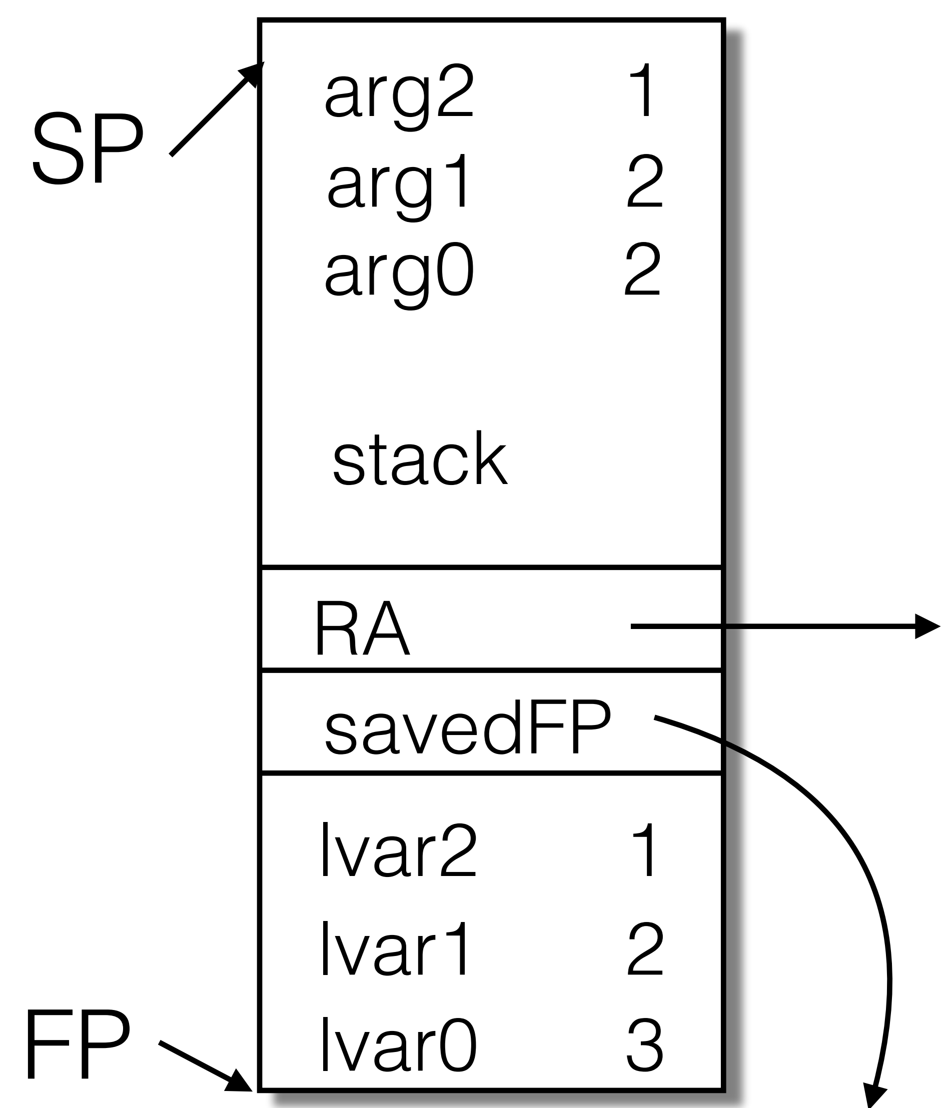
- How do we represent a function in bytecodes?
- Depends on the external VM interface; it can
  - Read a source string, and execute it, and/or
  - Ingest a binary file of bytecode.

The former does not constrain us at all; but for the latter we need to adopt a file format.

If functions become values, or can be reflected upon, then there are more decisions and constraints.

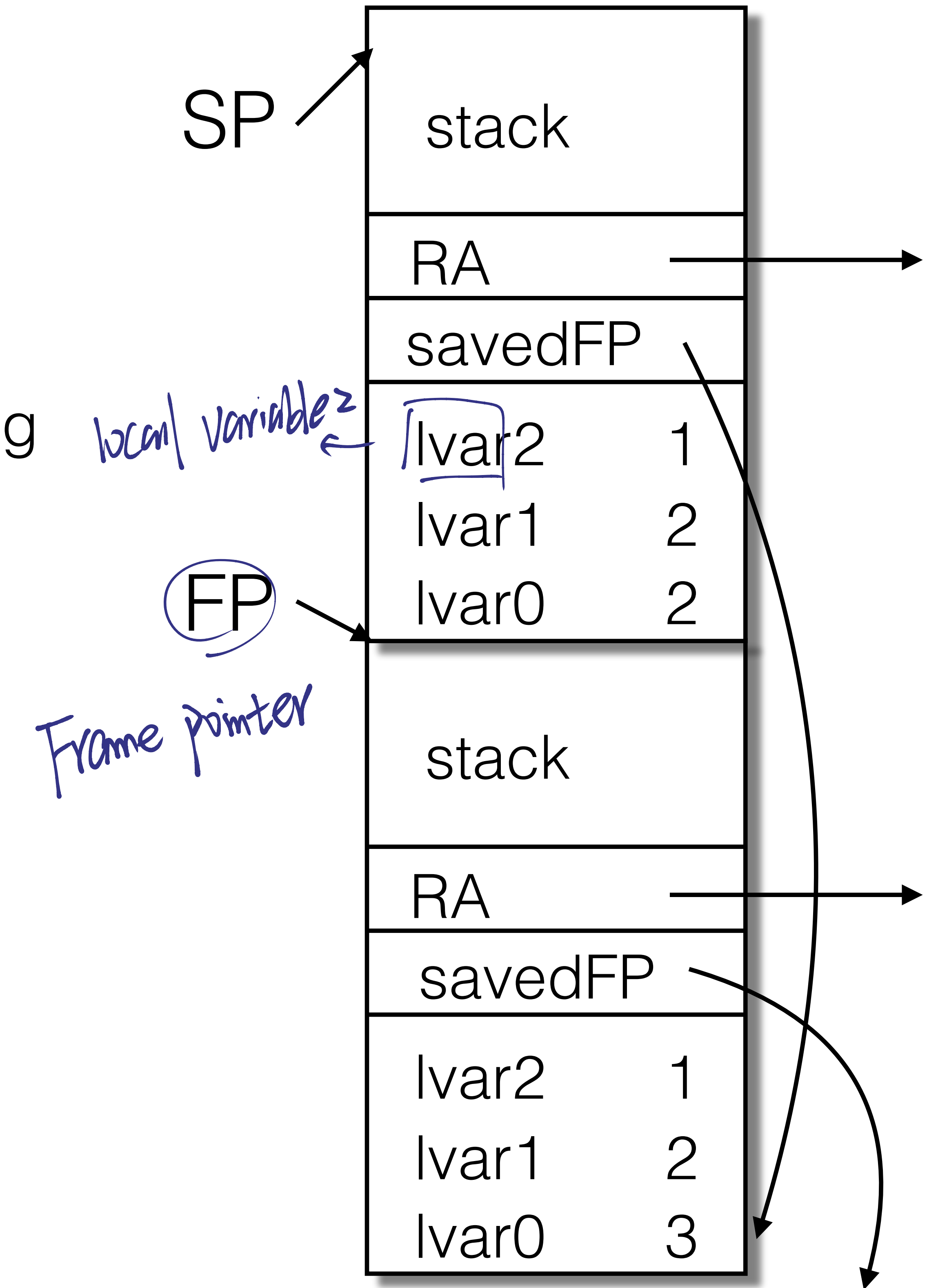
A simple CALL and RET implementation

- Add a frame pointer, pointing into the stack
- Call: Push FP, ret address
- Args are FP[0], FP[1], …



A simple CALL and RET implementation

- Add a frame pointer, pointing into the stack
- Call: Push FP, ret address
- Args are FP[0], FP[1], …



A simple CALL and RET implementation

```
case o(CALL1):
  nArgs = 1;
  push(fp); // save fp
  fp = sp - nArgs - 1; // new fp
  push(pc+2); //ret addr (Q1. Type of *sp?)
  pc = entry_PC_of_fun(pc[1]);
  continue;
case o(LGET): // lvar index in low order bits
  push(fp[L(*pc)]); pc++; continue;
case o(RET): // Q2. What's on the stack here?
  fp[0]= pop(); // ret val
  pc= pop(); // ret addr
  newsp= fp; fp= pop(); sp= newsp;
```

Method dispatch

- In object-oriented languages (and some others) there is an involved process in resolving a method call in order to determine the code to invoke.

- Typically:

  1. Find the *method identifier* (name, or sometimes signature).

  2. Find the *receiver* (`self`, `this`).

  3. Find the receiver's class.

  4. Search the methods of the class for one matching the (name, signature).

  5. If not found, move to the superclass, repeat 4. (Dynamic languages) If there is no superclass, the target method is a distinguished error handling method.

  6. Invoke the target method.

Serialization

- Enables distribution of bytecode-compiled programs
  - Example: GNU Emacs (.el, .elc)
- Can introduce new states that cannot be reached from the source language
  - Example from simple expression language
  - Must defend against these, or at least define behavior

Lab assignment #3

- Write a bytecode interpreter for Feeny

Performance

- [after the lab]
- Source to bytecodes to native instructions
- Cycles per bytecode
- Branch prediction

Performance

- Walk through same expression, machine code
- Hard-to-predict jumps
  - In effect, the behavior of the original program is being masked by the interpreter
- See [RSS15] for a recent take on this

Other properties

- Denser encoding
- Easily serialized
- Harder to decompile(?)