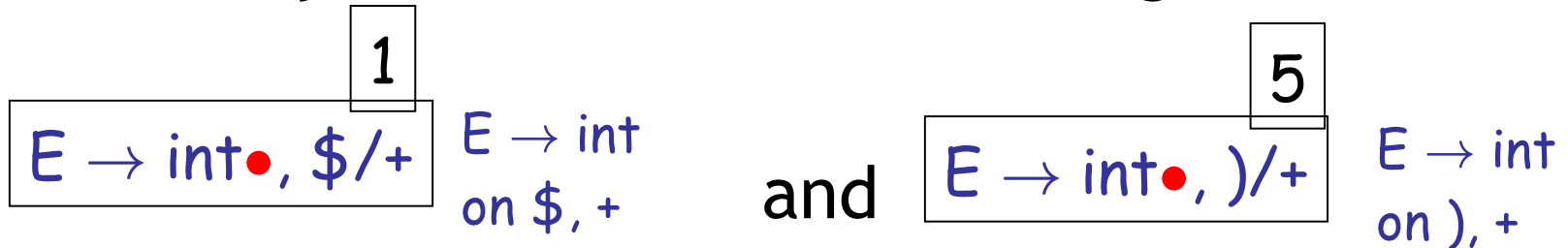


**GOLD HILL**  
EST. — 1859  
ELEV. — 8463  
POP. — 118  
TOTAL 10440

**Code  
Generation  
Super  
Lectures**

# LR(1) Parsing Tables are Big

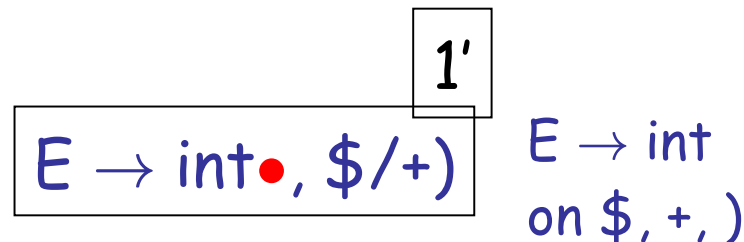
- But many states are similar, e.g.



- Idea: **merge** the DFA states whose items **differ only in the lookahead tokens**

- We say that such states have the same **core**

- We obtain



# LR(1) Parsing Tables are Big

- But many states are similar e.g.



- E
- Ident
- We

$E \rightarrow \text{int} \bullet, \$ / + )$

$E \rightarrow \text{int}$   
 $\text{on } \$, +, )$

# Huge One-Slide Summary

- **Assembly language** is untyped, unstructured, low-level and imperative. In a **load-store** architecture, **instructions** operate on **registers** (which are like global variables). The **stack pointer** is a special-purpose register.
- We can **generate code** by targeting a **stack machine** and using assembly instructions to implement the stack. The stack holds intermediate values, temporaries, and function arguments. The **accumulator** register (conceptually, the top of the stack) holds the result of the last computation. As an **invariant**, the stack is unchanged by intermediate calculations.
- We will maintain a **stack discipline** (or **calling convention**). Each function call is represented on the stack by an **activation record** (or **stack frame**). The activation record contains the **frame pointer**, the **parameters**, the **self** object pointer, the **return address**, and space for **temporaries**. The code you generate for function calls and function bodies must consistently agree on the calling convention.
- Our **object layout** choice must support using a subtype whenever a supertype is expected. Objects are **contiguous** blocks of memory that hold bookkeeping information (e.g., type tags, method pointers) as well as space for **fields**. **Subobjects** will **extend** (be bigger than in memory) their superobjects and will **share a common prefix**.
- A **dispatch table** (or **virtual function** table or **vtable**) is an array of pointers to methods. Each object points to its vtable, and members of a class **share** one vtable. This allows us to implement **dynamic dispatch**: method invocation is resolved by looking up the method address in the object's vtable at **runtime**.

# (Two Day) Lecture Outline

- Stack machines
  - e.g., Java Virtual Machine
- The COOL-ASM assembly language
  - It's MIPS/RISC + Java Bytecode
- A simple source language
- Stack-machine implementation of the simple language
- An optimization: stack-allocated variables
- Object Oriented Code Generation
  - Object Layout, Dynamic Dispatch

# Stack Machines

- A simple evaluation model
- No variables or registers
- A stack of values for intermediate results

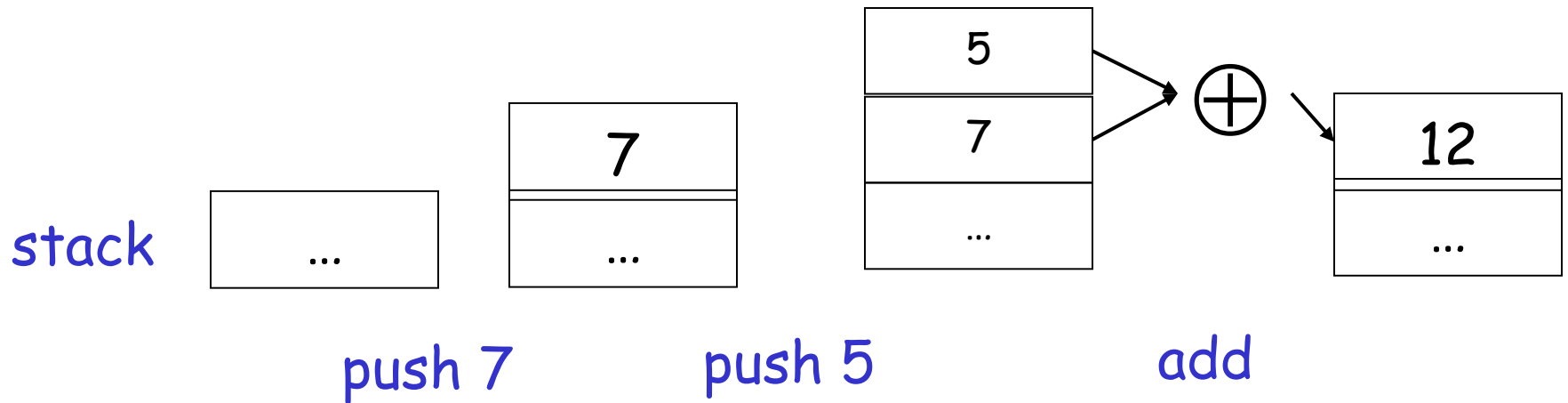


# Example

## Stack Machine Program

- Consider two instructions
  - `push i` - place the integer `i` on top of the stack
  - `add` - pop two elements, add them and put the result back on the stack
- A program to compute  $7 + 5$ :
  - `push 7`
  - `push 5`
  - `add`

# Stack Machine Example



- Each instruction:
  - Takes its operands from the top of the stack
  - Removes those operands from the stack
  - Computes the required operation on them
  - Pushes the result on the stack



# Why Use a Stack Machine ?

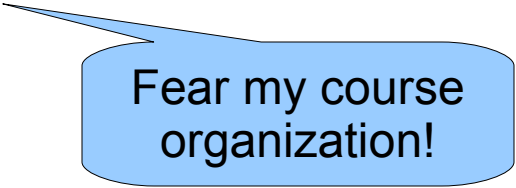
- Each operation takes operands from the same place and puts results in the same place
- This means a uniform compilation scheme
- And therefore a simpler compiler
  - This is the easiest way to do PA6
  - The reference compiler is more complicated

# Why Use a Stack Machine ?

- Location of the operands is implicit
  - Always on the top of the stack
- No need to specify operands explicitly
- No need to specify the location of the result
- Instruction “**add**” as opposed to “**add**  $r_1$ ,  $r_2$ ”
  - ⇒ Smaller encoding of instructions
  - ⇒ More compact programs (= faster: why?)
- This is one reason why Java Bytecodes use a stack evaluation model

# Optimizing the Stack Machine

- The add instruction does 3 memory operations
  - Two reads and one write to the stack
  - The top of the stack is frequently accessed
- Idea: keep the top of the stack in a register (called the **accumulator**)
  - This should remind you of **Fold**
  - Register accesses are faster
- The “**add**” instruction is now
$$\text{acc} \leftarrow \text{acc} + \text{top\_of\_stack}$$
  - Only one memory operation!



Fear my course organization!

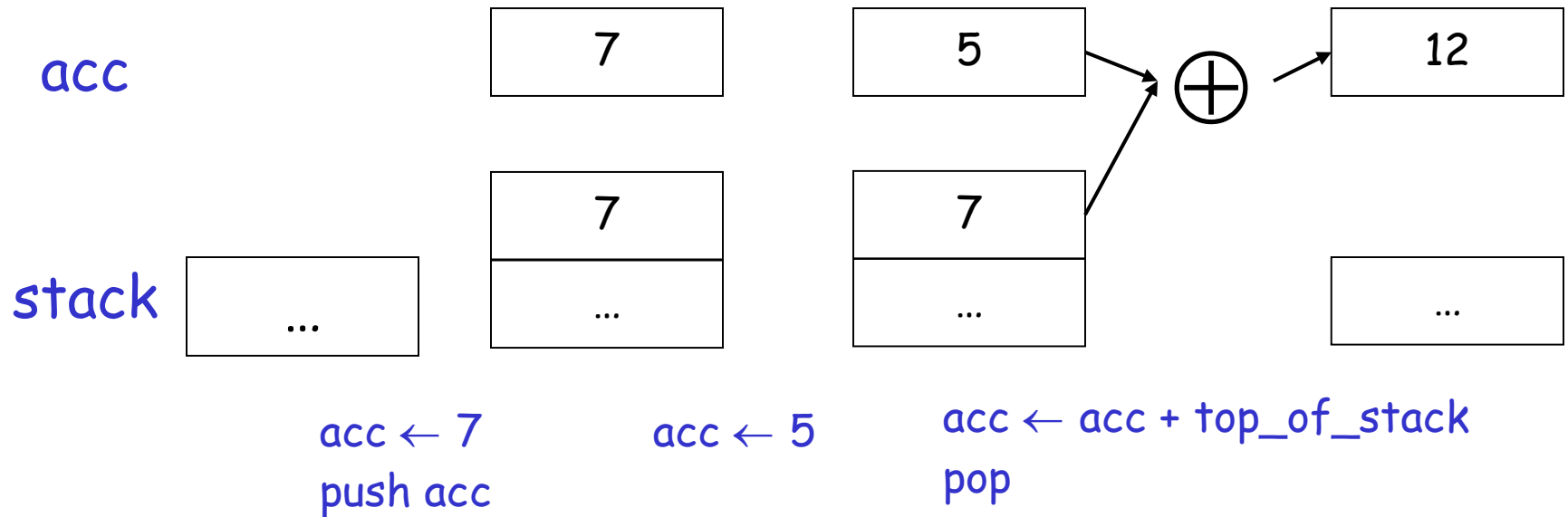
# Accumulator Invariants

- The result of computing an expression is always in the accumulator
- For an operation  $op(e_1, \dots, e_n)$  **push** the accumulator on the stack after computing each of  $e_1, \dots, e_{n-1}$ 
  - $e_n$ 's result is in the accumulator before **op**
  - After the operation **pop**  $n-1$  values
- After computing an expression the stack is as before

Example on next slide!

# Stack Machine with Accumulator: Example

- Compute  $7 + 5$  using an accumulator

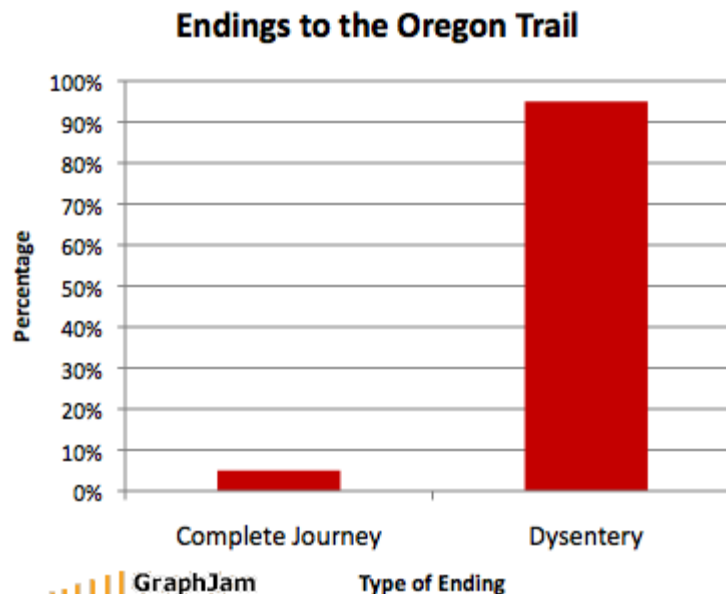


# A Bigger Example: $3 + (7 + 5)$

Code	Acc	Stack
<code>acc ← 3</code>	3	<init>
<code>push acc</code>	3	3, <init>
<code>acc ← 7</code>	7	3, <init>
<code>push acc</code>	7	7, 3, <init>
<code>acc ← 5</code>	5	7, 3, <init>
<code>acc ← acc + top_of_stack</code>	12	7, 3, <init>
<code>pop</code>	12	3, <init>
<code>acc ← acc + top_of_stack</code>	15	3, <init>
<code>pop</code>	15	<init>

# Notes

- It is **critical** that **the stack is preserved across the evaluation of a subexpression**
  - Stack before evaluating  $7 + 5$  is  $3, <init>$
  - Stack after evaluating  $7 + 5$  is  $3, <init>$
  - The first operand is on top of the stack



# From Stack Machines to RISC

- Our compiler will generate code for a stack machine with accumulator
- We want to run the resulting code on a processor
- We'll implement stack machine instructions using COOL-ASM instructions and registers
- Thus: Assembly Language





# Risky Business

- COOL-ASM is a RISC-style **assembly language**
  - An untyped, unsafe, low-level, fast programming language with few-to-no primitives.
- A **register** is a fast-access untyped global variable shared by the entire assembly program.
  - COOL-ASM: 8 general registers and 3 special ones (stack pointer, frame pointer, return address)
- An **instruction** is a primitive statement in assembly language that operates on registers.
  - COOL-ASM: add, jmp, ld, push, ...
- A **load-store** architecture: bring values in to registers from memory to operate on them.

# Drink Your Cool-Aid

- Sample COOL-ASM instructions:
  - See the CRM for all of them ...

add r2 <- r5 r2	; r2 = r5 + r2
li r5 <- 183	; r5 = 183
ld r2 <- r1[5]	; r2 = *(r1+5)
st r1[6] <- r7	; *(r1+6) = r7
my_label:	-- dashdash also a comment
push r1	; *sp = r1; sp --;
sub r1 <- r1 1	; r1 -- ;
bnz r1 my_label	; if (r1 != 0) goto my_label



# Simulating a Stack Machine...

- The **accumulator** is kept in register **r1**
  - This is just a convention. You could pick **r2**.
- The stack is kept in memory
- The stack **grows towards lower addresses**
  - Standard convention on the MIPS architecture
- The address of the next unused location on the stack is kept in register **sp**
  - The top of the stack is at address **sp + 1**
  - COOL-ASM “Word Size” = 1 = # of memory cells taken up by one integer/pointer/string

# Cool Assembly Example

- The stack-machine code for  $7 + 5$ :

`acc <- 7`

`push acc`

`acc <- 5`

`acc <- acc + top_of_stack`

`pop`

`li r1 7`

`sw sp[0] <- r1`

`sub sp <- sp 1`

`li r1 5`

`lw r2 <- sp[1]`

`add r1 <- r1 r2`

`add sp <- sp 1`

- We now generalize this to a simple language...

# Stack Instructions

- We have these COOL-ASM instructions:

**push rX**

```
st sp[0] <- rX
```

```
sub sp <- sp 1
```

**pop rX**

```
ld rX <- sp[1]
```

```
add sp <- sp 1
```

**; Note:**

**rX <- top**

```
ld rX <- sp[1]
```

# A Small Language

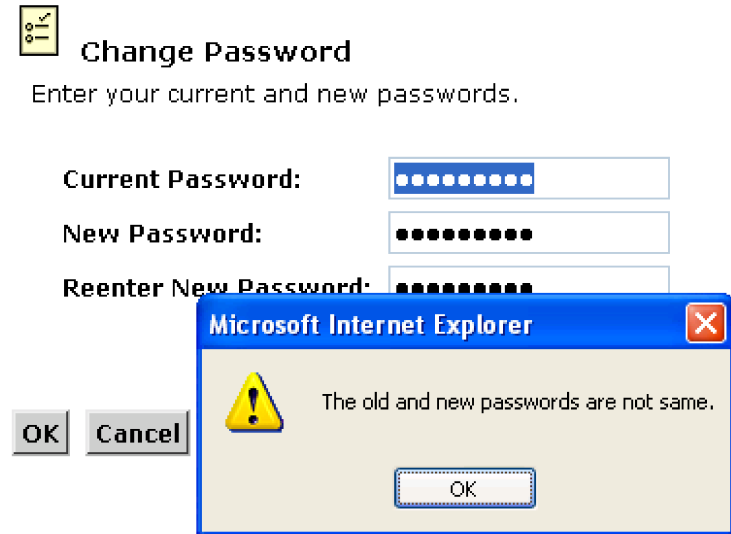
- A source language with integers and integer operations

$P \rightarrow D; P \mid D$

$D \rightarrow \text{def id(ARGS) = E;}$

$\text{ARGS} \rightarrow \text{id, ARGS} \mid \text{id}$

$E \rightarrow \text{int} \mid \text{id} \mid \text{if } E_1 = E_2 \text{ then } E_3 \text{ else } E_4$   
 $\mid E_1 + E_2 \mid E_1 - E_2 \mid \text{id}(E_1, \dots, E_n)$



# A Small Language (Cont.)

- The first function definition  $f$  is the “main” routine
- Running the program on input  $i$  means computing  $f(i)$
- Program for computing the Fibonacci numbers:

```
def fib(x) = if x = 1 then 0 else
             if x = 2 then 1 else
             fib(x - 1) + fib(x - 2)
```

# Code Generation Strategy

- For each expression  $e$  we generate COOL-ASM code that:
  - Computes the value of  $e$  in  $r1$  (accumulator)
  - Preserves  $sp$  and the contents of the stack
- We define a **code generation function**  $cgen(e)$  whose result is the code generated for  $e$



# Code Generation for Constants

- The code to evaluate a constant simply copies it into the accumulator:

**cgen(123) = li r1 123**

- Note that this also preserves the stack, as required



# Code Generation: Add

$\text{cgen}(e_1 + e_2) =$

$\text{cgen}(e_1)$

push r1

$\text{cgen}(e_2)$

*;; e2 now in r1*

pop t1

add r1 t1 r1

t1 is some  
unused  
“temporary”  
register

- Possible optimization: Put the result of  $e_1$  directly in register  $t1$  ?

# Code Generation Mistake

- Unsafe Optimization: put the result of  $e_1$  directly in  $t1$ ?

```
cgen( $e_1 + e_2$ ) =  
cgen( $e_1$ )  
mov t1 <- r1  
cgen( $e_2$ )  
add r1 <- t1 r1
```

- Try to generate code for :  $3 + (7 + 5)$



# Code Generation Notes

- The code for  $+$  is a template with “holes” for code for evaluating  $e_1$  and  $e_2$
- Stack-machine code generation is **recursive**
- Code for  $e_1 + e_2$  consists of code for  $e_1$  and  $e_2$  glued together
- Code generation can be written as a **recursive-descent tree walk** of the AST
  - At least for expressions

# Code Generation: Sub

- New instruction: `sub reg1 <- reg2 reg3`
  - Implements  $reg_1 \leftarrow reg_2 - reg_3$

**cgen(e<sub>1</sub> - e<sub>2</sub>) =**

`cgen(e1)`

`push r1`

`cgen(e2)`

`pop t1`

`sub r1 <- t1 r1`



# Code Generation: If

- We need flow control instructions
- New instruction: **beq reg<sub>1</sub> reg<sub>2</sub> label**
  - **Conditional Branch** to label if  $reg_1 = reg_2$
- New instruction: **jmp label**
  - **Unconditional Jump** to label

# Code Generation for If (Cont.)

```
cgen(if  $e_1 = e_2$  then  $e_3$  else  $e_4$ ) =  
  cgen( $e_1$ )  
  push r1  
  cgen( $e_2$ )  
  pop t1  
  beq r1 t1 true_branch ;; else fall through  
  cgen( $e_4$ )  
  jmp end_if  
true_branch:  
  cgen( $e_3$ )  
end_if:
```

# The Activation Record

- An **activation record** (or **stack frame**) stores calling context information on the stack during a function call.
- Code for function calls/definitions depends on the layout of the activation record
- A very simple AR suffices for this language:
  - The result is always in the accumulator
    - No need to store the result in the AR
  - The activation record holds actual parameters
    - For  $f(x_1, \dots, x_n)$  push  $x_1, \dots, x_n$  on the stack
    - These are the only variables in this language

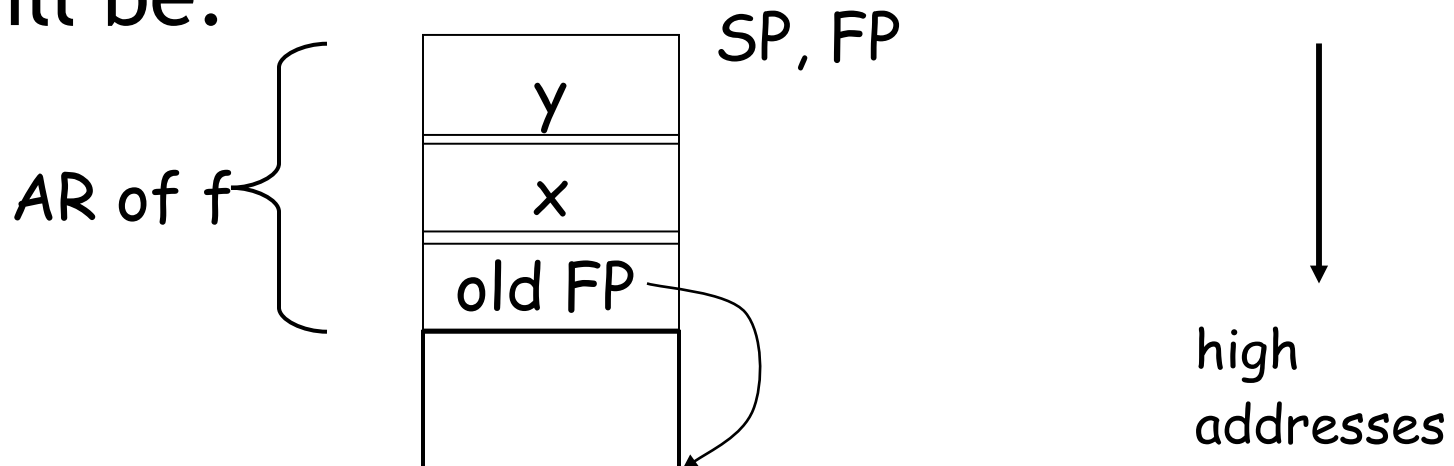


# Calling Convention

- This **calling convention** (or **stack discipline**) guarantees that on function exit **sp** is the same as it was on entry
  - No need to save **sp**
- We need the return address
- It's handy to have a pointer to start of the current activation
  - This pointer lives in register **fp** (frame pointer)
  - Reason for frame pointer will be clear shortly

# The Activation Record

- Summary: For this language, an AR with the caller's frame pointer, the actual parameters, and the return address suffices
- Picture: Consider a call to  $f(x,y)$ . The AR will be:



# Code Generation: Function Call

- The **calling sequence** is the instructions (of both caller and callee) to set up a function invocation
- New instruction: **call label**
  - Jump to label, save address of next instruction in **ra**
  - On other architectures the return address is stored on the stack by the “call” instruction
  - (This is also called “branch and link”.)

# Code Generation: Function Call

**cgen(f(e<sub>1</sub>,...,e<sub>n</sub>)) =**

push fp

cgen(e<sub>1</sub>)

push r1

...

cgen(e<sub>n</sub>)

push r1

call f\_entry

pop fp

- The caller saves its value of the frame pointer
- Then it saves the actual arguments in order
- The caller saves the return address in register **ra**
- The AR so far is **n+1** bytes long
- Caller restores **fp**

# Code Generation: Function Def

- New instruction: **return**
  - Jump to address in register **ra**

**cgen(def f(x<sub>1</sub>,...,x<sub>n</sub>) = e) =**

**f\_entry:**

**mov fp <- sp**

**push ra**

**cgen(e)**

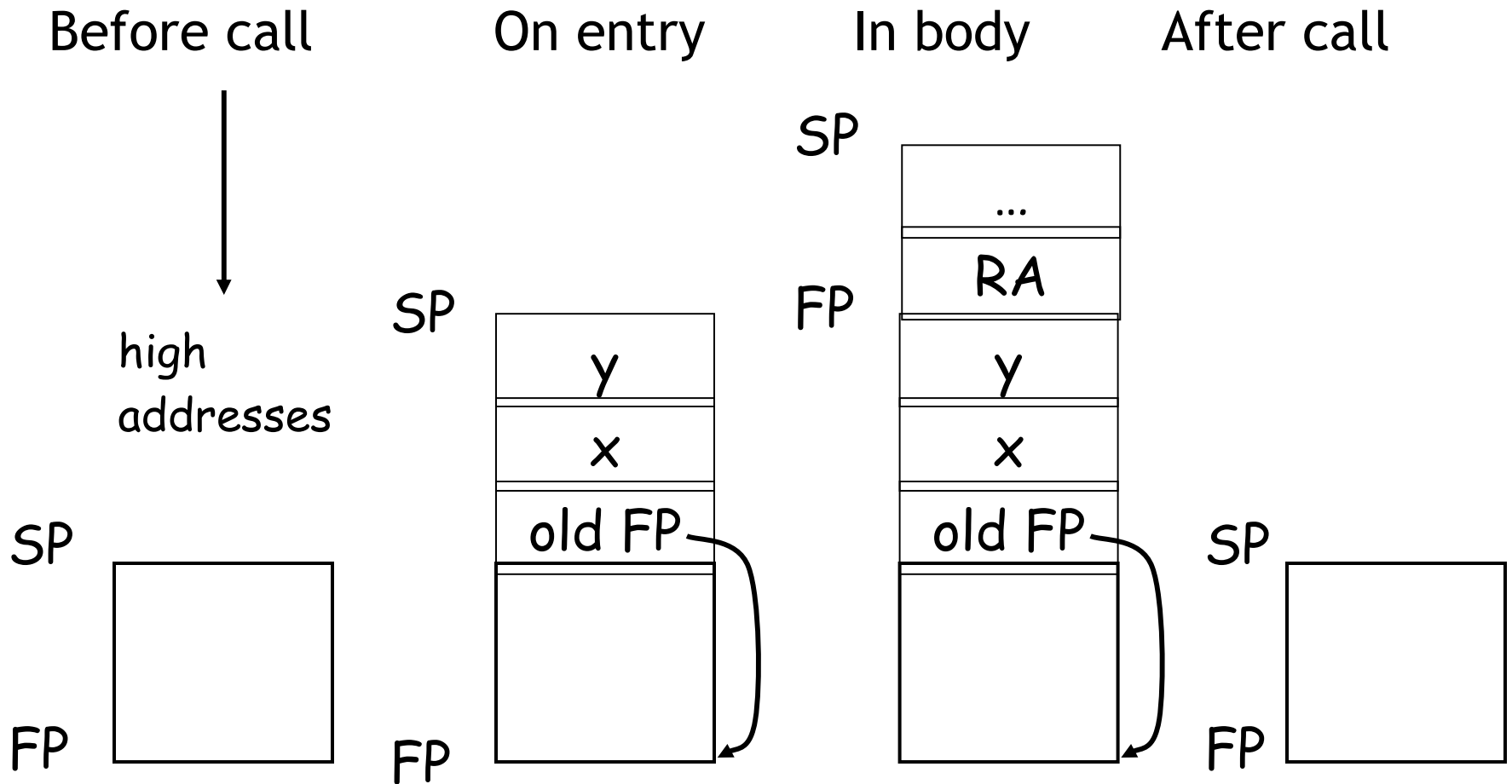
**ra <- top**

**add sp <- sp z**

**return**

- Note: The frame pointer points to the top, not bottom of the frame
- The callee pops the return address, the actual arguments and the saved value of the frame pointer
- **z = n + 2** (so far)

# Calling Sequence: $f(x,y)$



# Code Generation: Variables

- Variable references are the last construct
- The “variables” of a function are just its parameters
  - They are all in the AR
  - Pushed by the caller
- Problem: Because the stack grows when intermediate results are saved, the variables are not at a fixed offset from `sp`
  - Impress me: what are they a fixed offset from?

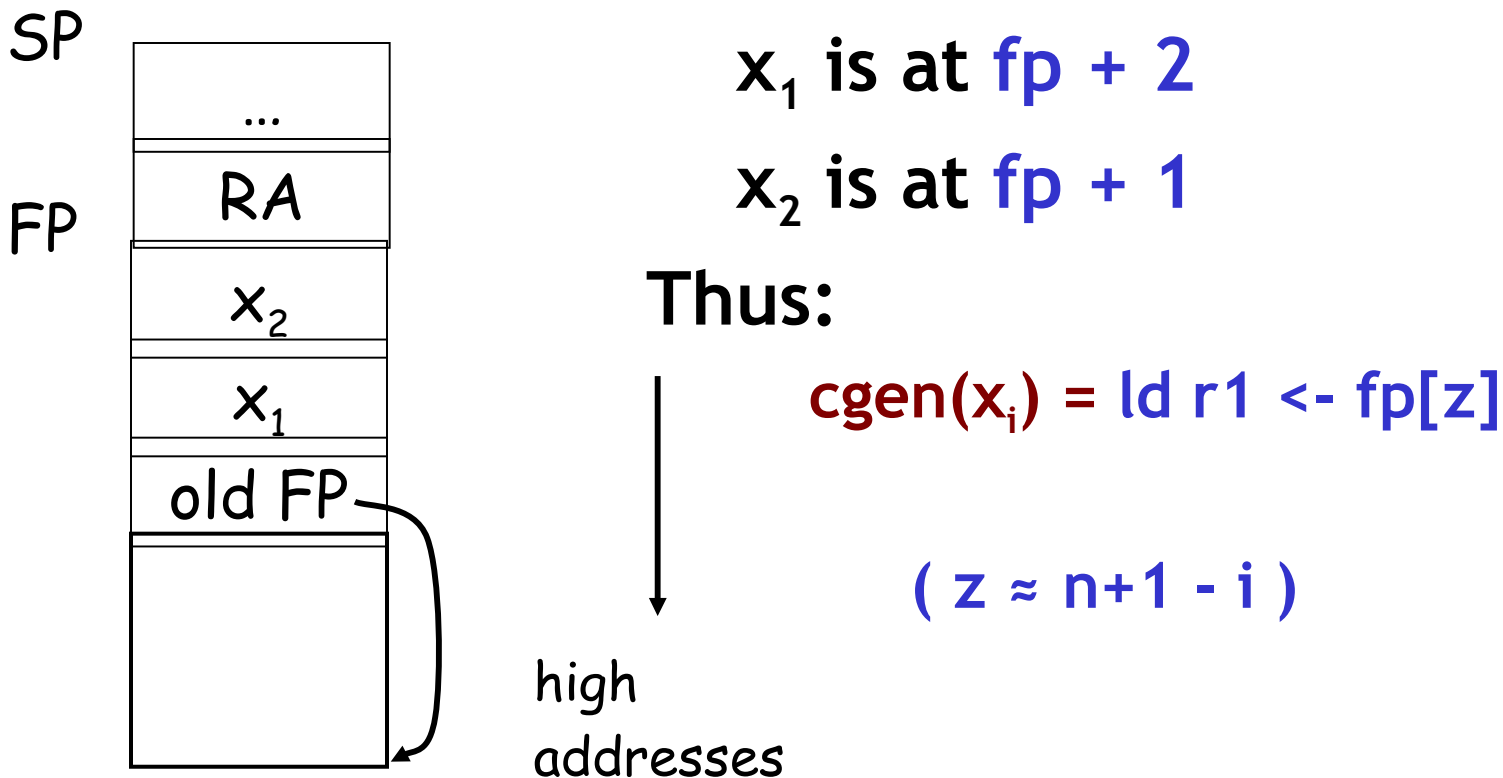
# Code Generation: Variables

- Solution: use the **frame pointer**
  - Always points to the return address on the stack (= the value of sp on function entry)
  - Since it does not move it can be used to find arguments stored on the stack
- Let  $x_i$  be the  $i^{\text{th}}$  ( $i = 1, \dots, n$ ) formal parameter of the function for which code is being generated



# Code Generation: Variables

- Example: For a function `def f(x1,x2) = e` the activation and frame pointer are set up as follows:



# Summary

- The activation record must be designed together with the code generator
- Code generation can be done by recursive traversal of the AST
- We recommend you use a stack machine for your Cool compiler (it's simple)

# More Information

- use cool `--asm hello-worl.cl` for examples
- Production compilers do different things
  - Emphasis is on keeping values (esp. current stack frame) in registers
  - Intermediate results are laid out in the AR, not pushed and popped from the stack

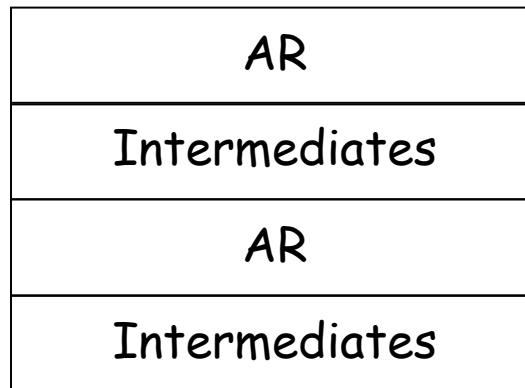


# Optimization: Allocating Temporaries in the Activation Record



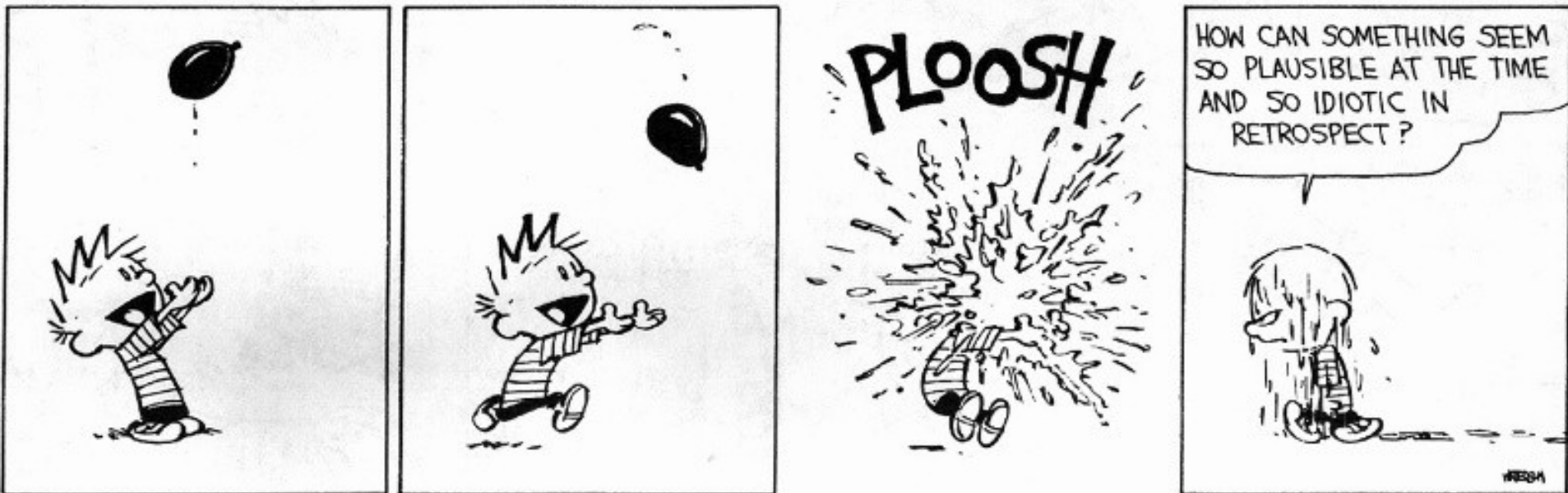
# Review

- The stack machine code layout we've described so far has activation records and intermediate results interleaved on the stack



# Stack Machine Implications

- Advantage: Very simple code generation
- Disadvantage: **Very slow code**
  - Storing and loading temporaries requires a store/load and **sp** adjustment



# A Better Way

- Idea: Keep temporaries in the AR
- Work: The code generator must assign space in the AR for each temporary



# Example

```
def fib(x) = if x = 1 then 0 else  
            if x = 2 then 1 else  
            fib(x - 1) + fib(x - 2)
```

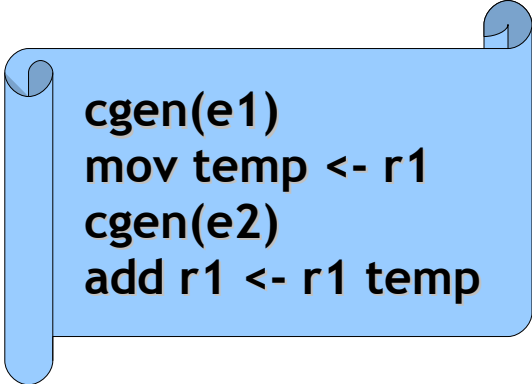
- We must determine:
  - What intermediate values are placed on the stack?
  - How many slots are needed in the AR to hold these values?



# How Many Temporaries?

- Let  $NT(e)$  = # of temps needed to eval  $e$

- Example:  $NT(e_1 + e_2)$



```
cgen(e1)
mov temp <- r1
cgen(e2)
add r1 <- r1 temp
```

- Needs at least as many temporaries as  $NT(e_1)$
  - Needs at least as many temporaries as  $NT(e_2) + 1$
- Space used for temporaries in  $e_1$  can be reused for temporaries in  $e_2$

# The NumTemps Equations

$$\text{NT}(e_1 + e_2) = \max(\text{NT}(e_1), 1 + \text{NT}(e_2))$$

$$\text{NT}(e_1 - e_2) = \max(\text{NT}(e_1), 1 + \text{NT}(e_2))$$

$$\text{NT}(\text{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4)$$

$$= \max(\text{NT}(e_1), 1 + \text{NT}(e_2), \text{NT}(e_3), \text{NT}(e_4))$$

$$\text{NT}(\text{id}(e_1, \dots, e_n)) = \max(\text{NT}(e_1), \dots, \text{NT}(e_n))$$

$$\text{NT}(\text{int}) = 0$$

$$\text{NT}(\text{id}) = 0$$

Is this bottom-up or top-down? (you tell me)

What is  $\text{NT}(\dots\text{code for fib}\dots)$ ?

# The Revised AR

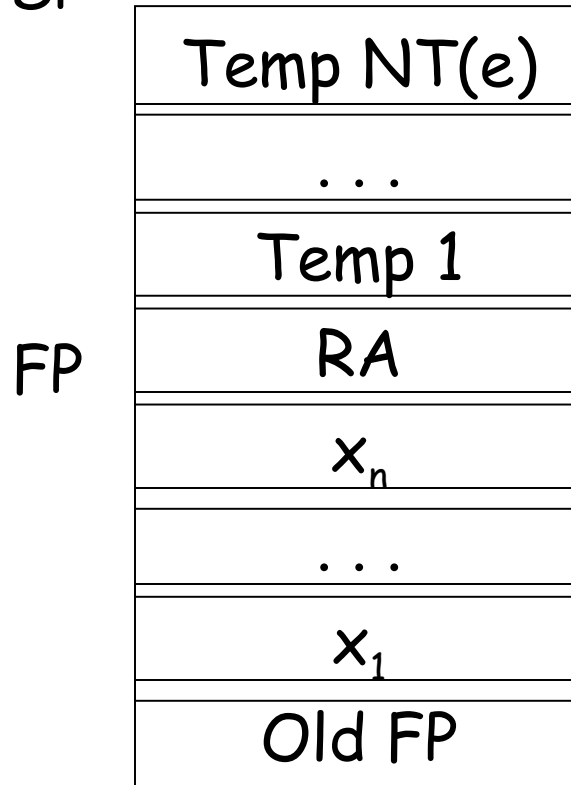
- For a function definition  $f(x_1, \dots, x_n) = e$  the AR has  $2 + n + NT(e)$  elements (so far)
  - Return address
  - Frame pointer
  - $n$  arguments
  - $NT(e)$  locations for intermediate results



# Stack Frame Picture

$$f(x_1, \dots, x_n) = e$$

SP



FP



high  
addresses

# Revised Code Generation

- Code generation must know how many temporaries are in use at each point
- Add a new argument to code generation: the position of the next available temporary

`cgen(e, n)` : generate code for `e` and use temporaries whose address is `(fp - n)` or lower

# Code Generation for +

$\text{cgen}(e_1 + e_2) =$

$\text{cgen}(e_1)$

push r1

$\text{cgen}(e_2)$

pop temp

add r1 <- r1 temp

$\text{cgen}(e_1 + e_2, nt) =$

$\text{cgen}(e_1, nt)$

st fp[-nt] <- r1

$\text{cgen}(e_2, nt+1)$

ld temp <- fp[-nt]

add r1 <- r1 temp

Where are the savings?

Hint: “push” is more expensive than it looks.

# Notes

- The temporary area is used like a small, fixed-size stack
- Exercise: Write out `cgen` for other constructs
- Hint: on function entry, you'll have to increment something by  $NT(e)$ 
  - ... and on function exit, decrement it ...

# Code Generation for Object-Oriented Languages





# Object Layout

- OO implementation =
  - Stuff from before + More stuff
- **Liskov Substitution Principle**: If B is a subclass of A, then an object of class B can be used wherever an object of class A is expected
- This means that code in class A **must work unmodified** on an object of class B

# Two Issues

- How are objects represented in memory?
- How is dynamic dispatch implemented?



# Object Layout (Cont.)

- An object is like a `struct` in C. The reference `foo.field` is an index into a `foo` struct at an offset corresponding to `field`
- Objects in Cool are implemented similarly
  - Objects are laid out in contiguous memory
  - Each attribute stored at a fixed offset in object
  - When a method is invoked, the object becomes `self` and the fields are the object's attributes

# Cool Object Layout

- The first 3 words of Cool objects contain header information:

	<i>Offset</i>
Class Type Tag	0
Object Size	1
Dispatch / Vtable Ptr	2
Attribute 1	3
Attribute 2	4
...	

(This is a convention that we made up, but it is similar to how Java and C++ lay things out. For example, you could swap #1 and #2 without loss. )

# Cool Object Layout

- **Class tag** (or “**type tag**”) is a raw integer
  - Identifies class of the object (Int=1, Bool=2, ...)
- **Object size** is an integer
  - Size of the object in words
- **Dispatch pointer** (or “**vtable pointer**”) is a pointer to a table of methods
  - More later
- **Attributes** are laid out in subsequent slots
- The layout is contiguous

# Object Layout Example

```
Class A {
```

```
  a: Int <- 0;
```

```
  d: Int <- 1;
```

```
  f(): Int { a <- a + d };
```

```
};
```

```
Class B inherits A {
```

```
  b: Int <- 2;
```

```
  f(): Int { a }; // Override
```

```
  g(): Int { a <- a - b };
```

```
};
```

```
Class C inherits A {
```

```
  c: Int <- 3;
```

```
  h(): Int { a <- a * c };
```

```
};
```

# Object Layout (Cont.)

- Attributes **a** and **d** are inherited by classes **B** and **C**
- All methods in all classes refer to **a**
- For **A** methods to work correctly in **A**, **B**, and **C** objects, attribute **a** must be in the same “place” in each object

# Subclass Layout

Observation: Given a layout for class **A**, a layout for subclass **B** can be defined by **extending** the layout of **A** with additional slots for the additional attributes of **B**

*(i.e., append new fields at bottom)*

Leaves the layout of **A** unchanged

(**B** is an extension)



# Object Layout Picture

```

Class A {
  a: Int <- 0;
  d: Int <- 1;
  f(): Int { a <- a + d };
};
  
```

```

Class B inherits A {
  b: Int <- 2;
  f(): Int { a }; // Override
  g(): Int { a <- a - b };
};
  
```

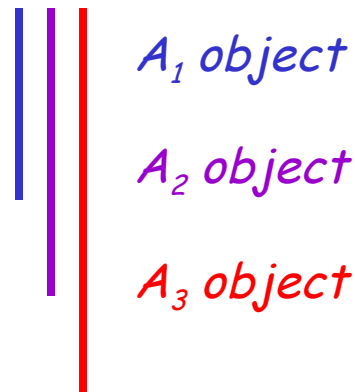
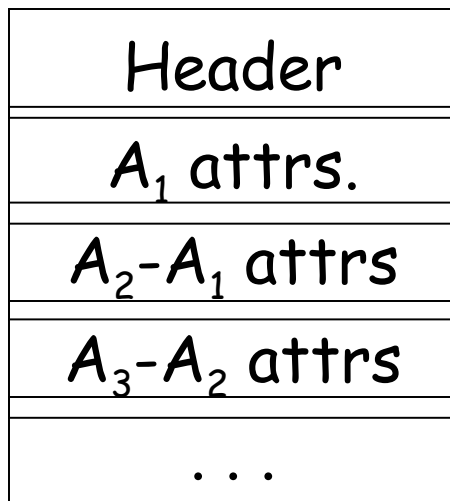
Class \ Offset	A	B	C
0 (tag)	Atag	Btag	Ctag
1 (size)	5	6	6
2 (vtable)	*	*	*
3 (attr#1)	a	a	a
4 ...	d	d	d
5		b	c

```

Class C inherits A {
  c: Int <- 3;
  h(): Int { a <- a * c };
};
  
```

# Subclasses (Cont.)

- The **offset for an attribute** is the **same** in a class and all of its subclasses
  - This choice allows any method for an  $A_1$  to be used on a subclass  $A_2$
- Consider layout for  $A_n \leq \dots \leq A_3 \leq A_2 \leq A_1$



*Extra Credit:  
What about  
multiple  
inheritance?*

# Dynamic Dispatch

- Consider **f** and **g**:

```
Class A {  
  a: Int <- 0;  
  d: Int <- 1;  
  f(): Int { a <- a + d };  
};
```

```
Class B inherits A {  
  b: Int <- 2;  
  f(): Int { a }; // Override  
  g(): Int { a <- a - b };  
};
```



```
Class C inherits A {  
  c: Int <- 3;  
  h(): Int { a <- a * c };  
};
```

# Dynamic Dispatch Example

- e.g()
  - g refers to method in B if e is a B
- e.f()
  - f refers to method in A if f is an A or C  
(inherited in the case of C)
  - f refers to method in B for a B object
- The implementation of methods and dynamic dispatch strongly resembles the implementation of attributes

# Dispatch Tables

- Every class has a fixed set of methods (including inherited methods)
- A **dispatch table** (or **virtual function table** or **vtable**) indexes these methods
  - A vtable is an array of method entry points
  - (Thus, a vtable is an array of function pointers.)
  - A method **f** lives at a **fixed offset** in the dispatch table for a class **and all of its subclasses**

# Dispatch Table Example

Class	A	B	C
Offset			
0	f_A	f_B	f_A
1		g	h

- The dispatch table for class **A** has only 1 method
- The tables for **B** and **C** extend the table for **A** with more methods
- Because methods can be overridden, the method for **f** is not the same in every class, but is always at the same offset
  - (i.e., offset 0 here)

# Using Dispatch Tables

- The dispatch pointer in an object of class  $X$  points to the dispatch table for class  $X$ 
  - i.e., all objects of class  $X$  **share** one table
- Every method  $f$  of class  $X$  is assigned an offset  $O_f$  in the dispatch table at compile time
  - i.e., by you in PA6 when you're generating the assembly code

# A Sense of Self

- Every method must know what object is “self”
  - Convention: **“self” is passed as the first argument** to all methods
- To implement a dynamic dispatch  $e.f()$  we
  - Evaluate  $e$ , obtaining an object  $x$
  - Find  $D$  by reading the dispatch-table field of  $x$
  - Call  $D[O_f](x)$ 
    - $D$  is the dispatch table for  $x$
    - In the call, **self** is bound to  $x$



# Dynamic Dispatch Hint

- To reiterate: objexp.mname(arg1)
  - push self
  - push fp
  - cgen(arg1)
  - push r1 ; push arg1
  - cgen(objexp)
  - bz r1 dispatch\_on\_void\_error
  - push r1 ; will be “self” for callee
  - ld temp <- r1[2] ; temp <- vtable
  - ld temp <- temp[X] ; X is offset of mname in vtables
  - ; for objects of typeof(objexp)
  - call temp
  - pop fp
  - pop self

# “Extra Credit”: Multiple Inheritance



# Example

- Assume that we extend Cool with multiple inheritance
- Consider the following 3 classes:

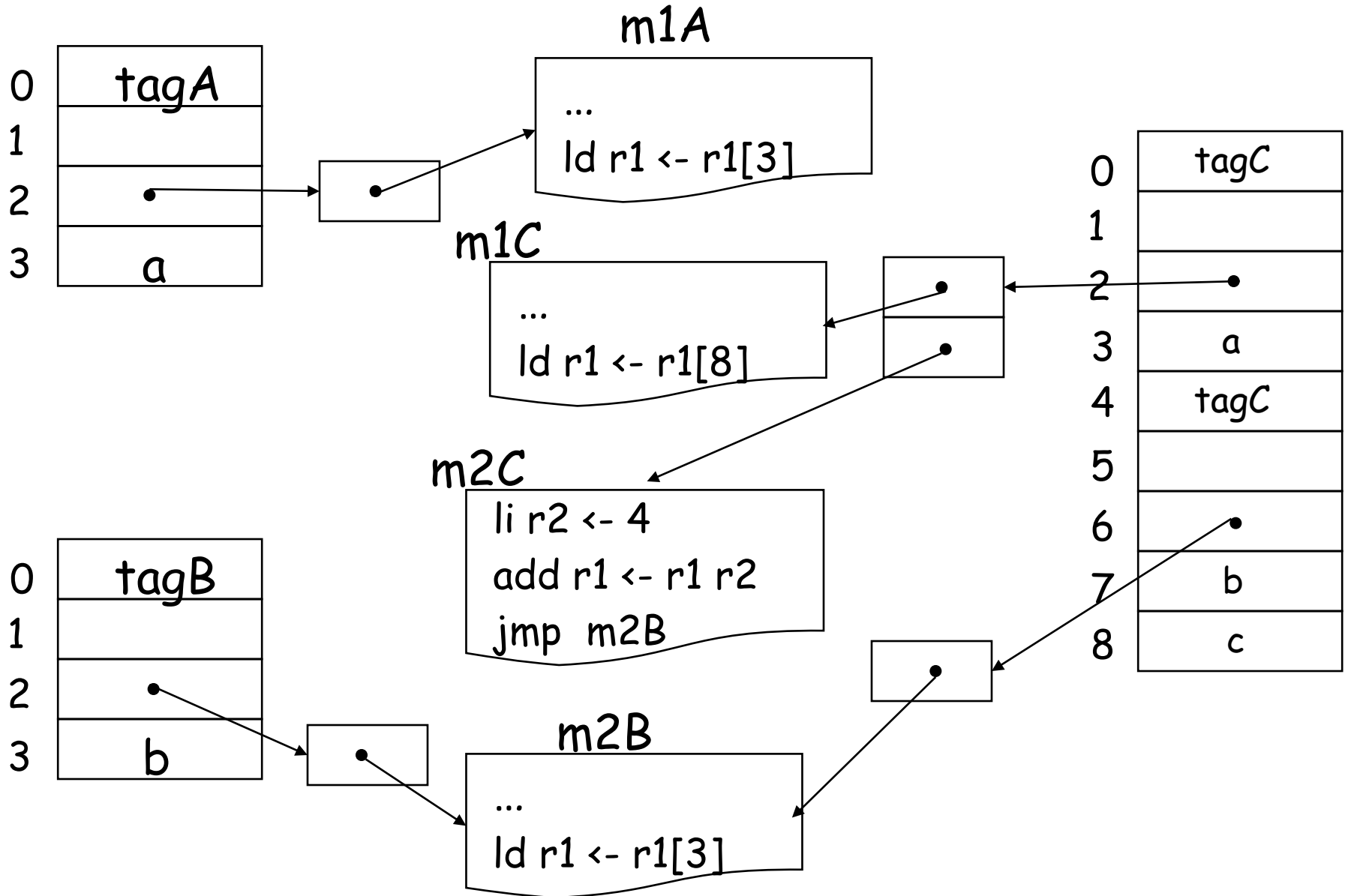
```
Class A { a : Int; m1() : Int { a }; }
```

```
Class B { b: Int; m2() : Int { b }; }
```

```
Class C inherit A, B { c : Int; m2() : Int { c }; }
```

- class C inherits attribute a and method m1 from A, attribute b from B and overrides m2

# Multi-Inherit Object Layout



# Homework

- PA3 (Parsing) Due
- WA3 Due
- Compilers: PA6c Due Next Week