CSC 348 – Intro to Compilers
Lecture 8

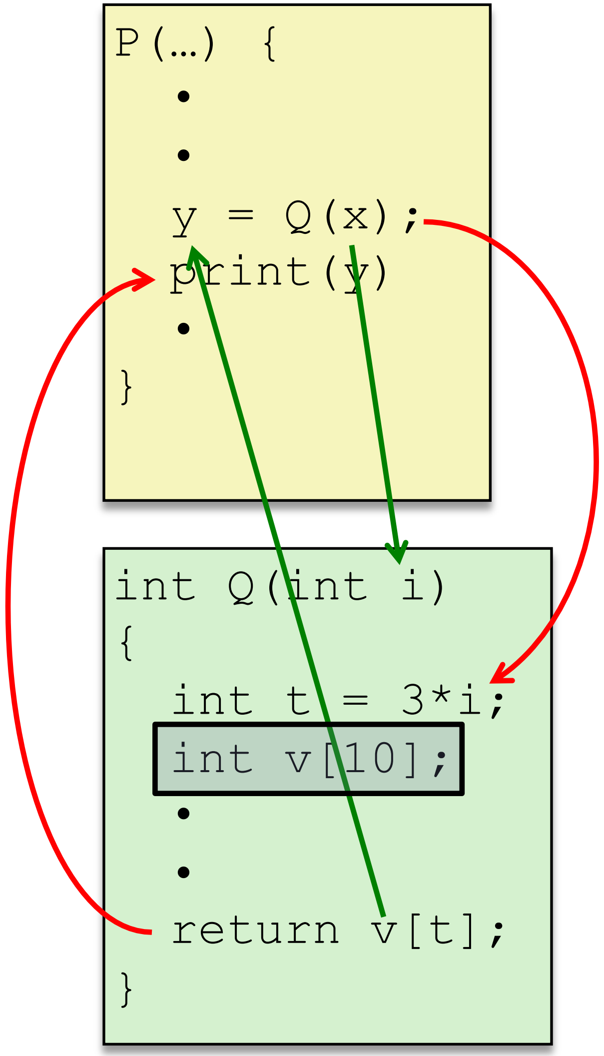
Dr. David Zaretsky
david.zaretsky@depaul.edu

Agenda

- Code Generation for the x86-64 architecture
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Stack Structure
  - Register Saving Conventions
  - Illustration of Recursion
  - Code construction
    - Statements
    - Expressions
    - Control Flow
    - Methods
    - Objects
- Programming Assignment 5: Code Translation

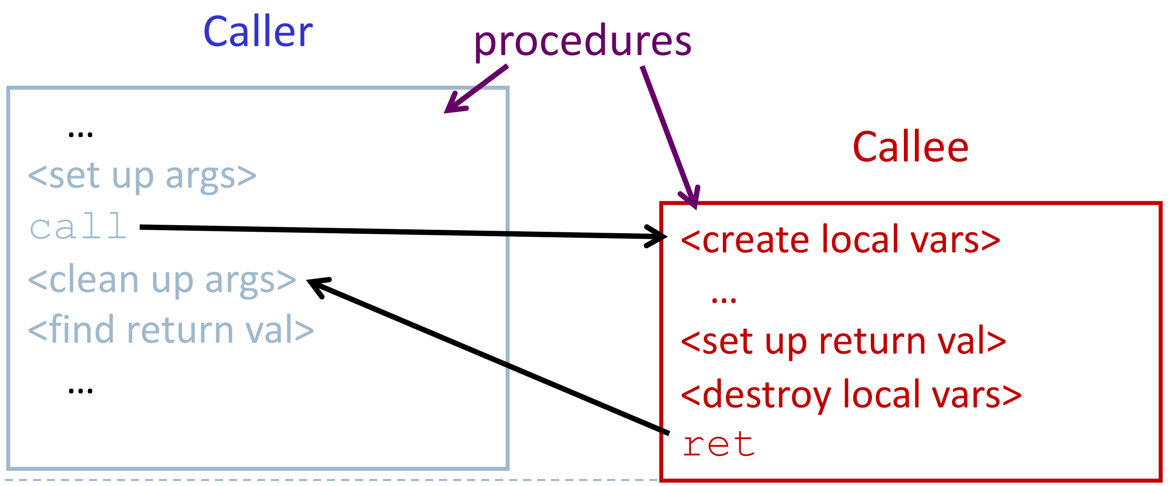
Mechanisms required for procedures

- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value
- Memory management
  - Allocate during procedure execution
  - Deallocate upon return
- All implemented with machine instructions!
  - An x86-64 procedure uses only those mechanisms required for that procedure



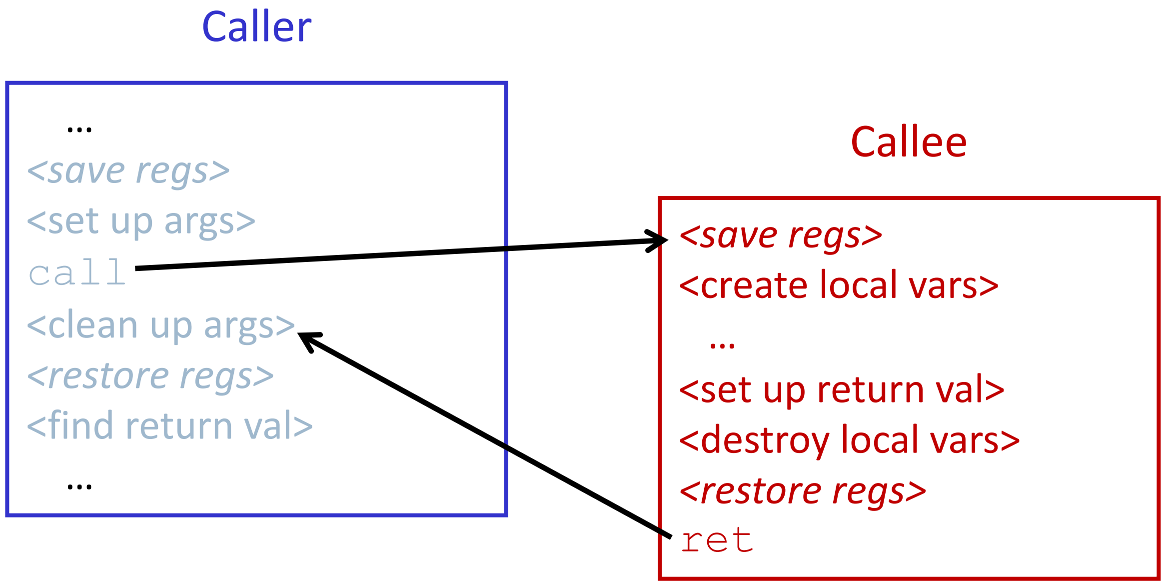
Method Call Overview

- Callee must know where to find args
- Callee must know where to find return address
- Caller must know where to find return value
- Caller and Callee run on same CPU, so use the same registers
  - How do we deal with register reuse?
- Unneeded steps can be skipped (e.g. no arguments)



Method Call Overview

- The convention of where to leave/find things is called the calling convention (or procedure call linkage)
  - Details vary between systems
  - We will see the convention for x86-64/Linux in detail
  - What could happen if our program didn't follow these conventions?



Call Example

```
int multstore(int x, int y)
{
    return mult2(x, y);
}
```

```
0000000000400540 <multstore>:
  400540: push   %rbx            # Save %rbx
  400541: movq   %rdx,%rbx       # Save dest
  400544: call   400550 <mult2>  # mult2(x,y)
  400549: movq   %rax,(%rbx)     # Save at dest
  40054c: pop    %rbx            # Restore %rbx
  40054d: ret                    # Return
```

```
int mult2(int a, int b)
{
  return a * b;
}
```

```
0000000000400550 <mult2>:
  400550:  movq   %rdi,%rax   # a
  400553:  imulq  %rsi,%rax   # a * b
  400557:  ret                # Return
```

Method Control Flow

- Use stack to support method call and return
- Method call: `call label`
  - Push return address on stack (why? which address?)
  - Jump to label
- Return address:
  - Address of instruction immediately after call instruction
  - Example from disassembly:

```
400544: call    400550 <mult2>
400549: movq    %rax,(%rbx)
```

next instruction happens to be a move, but could be anything

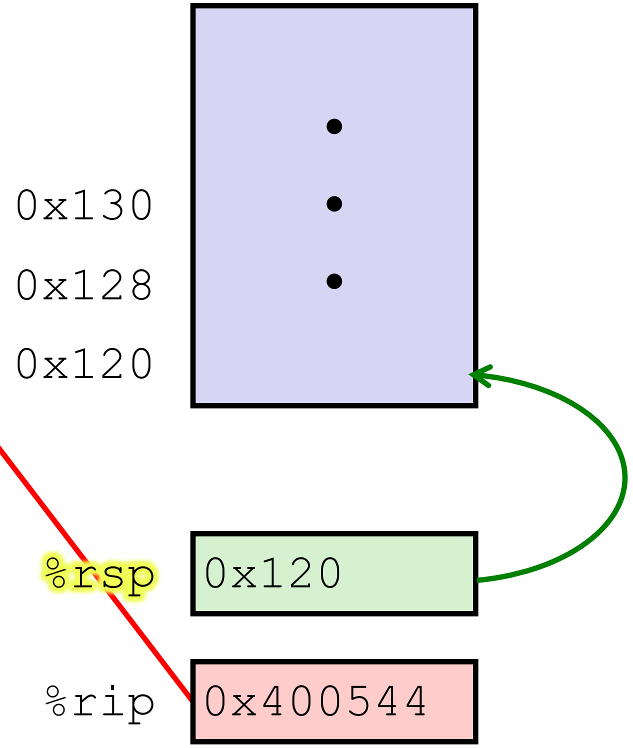
- Procedure return: ret
  - Pop return address from stack
  - Jump to address

Method Call Example (step 1)

- Push return address on stack
  - update %rsp → 0x118
  - Push 0x400549
- Jump to new function
  - Update %rip

```
0000000000400540 <multstore>:
  •
  •
  400544: call    400550 <mult2>
  400549: movq    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:   movq    %rdi,%rax
  •
  •
  400557:   ret
```

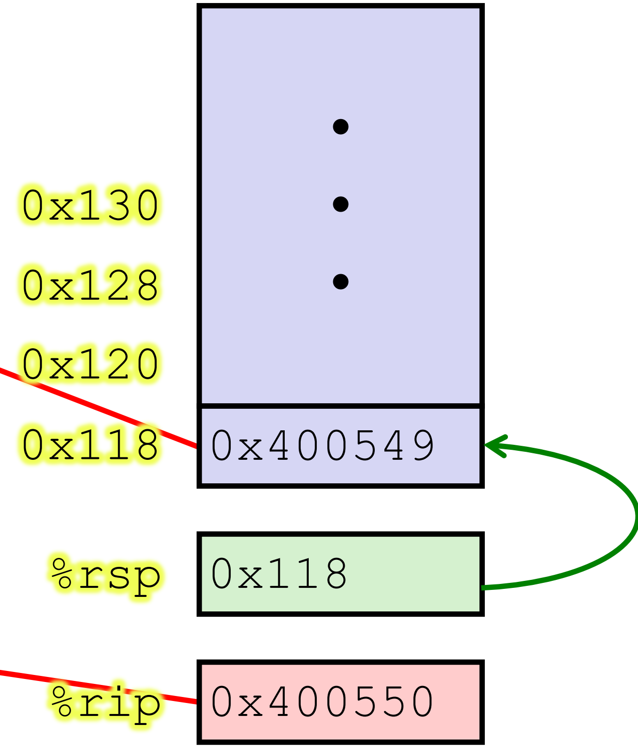


Method Call Example (step 2)

- Now return address points at where we'll jump back to
- **%rip** is where we are executing now

```
0000000000400540 <multstore>:
  •
  •
  400544: call    400550 <mult2>
  400549: movq    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:   movq    %rdi,%rax
  •
  •
  400557:   ret
```

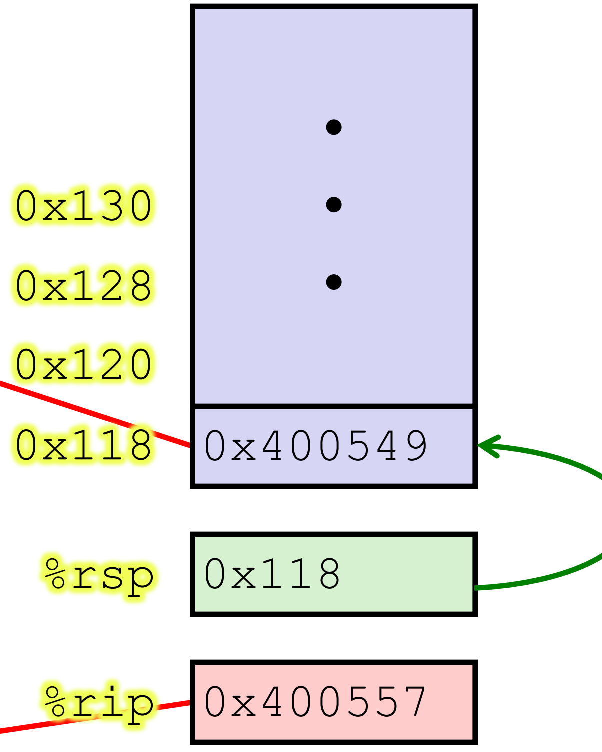


Method Return Example (step 1)

- Now, after we've executed this function, we're at the ret
- Pop return address
  - read return address
  - increment %rsp
- Jump back to return address
  - update %rip with return address
  - execute again starting there

```
0000000000400540 <multstore>:
  •
  •
  400544: call    400550 <mult2>
  400549: movq    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:  movq    %rdi,%rax
  •
  •
  400557:  ret
```

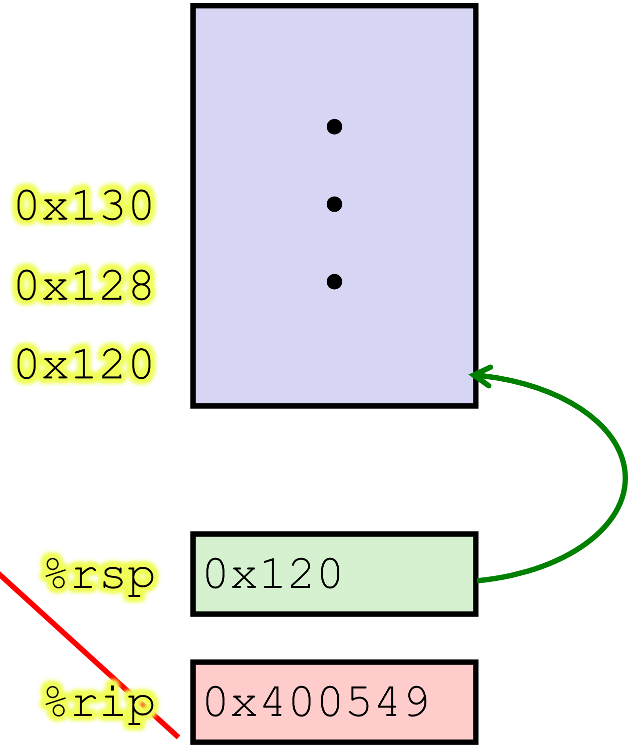


Method Return Example (step 2)

- Continue to instruction 0x400549
- Copy return value %rax to stack.

```
0000000000400540 <multstore>:
  •
  •
  400544: call    400550 <mult2>
  400549: movq    %rax,(%rbx)
  •
  •
```

```
0000000000400550 <mult2>:
  400550:   movq    %rdi,%rax
  •
  •
  400557:   ret
```



Method Data Flow

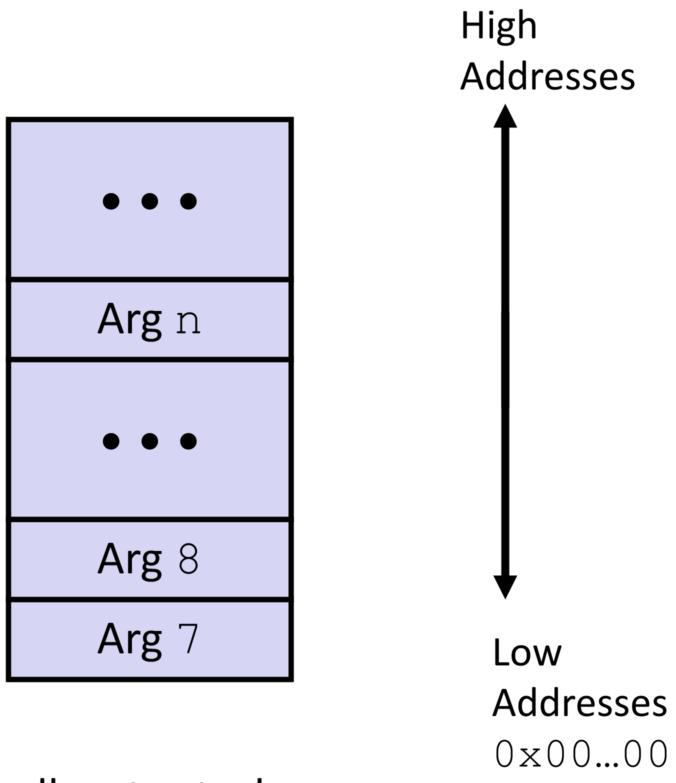
- Registers (NOT in Memory)
- First 6 arguments

Register	Argument
`%rdi`	Arg0 or this
`%rsi`	Arg1
`%rdx`	Arg2
`%rcx`	Arg3
`%r8`	Arg4
`%r9`	Arg5

- Return value

`%rax`

Stack (Memory)



- Only allocate stack space when needed

x86-64 Return Values

- By convention, values returned by methods are placed in `%rax`
  - Choice of `%rax` is arbitrary
- Caller must make sure to save the contents of `%rax` before calling a callee that returns a value
  - Part of register-saving convention
- Callee places return value into `%rax`
  - Any type that can fit in 8 bytes – integer, float, pointer, etc.
  - For return values greater than 8 bytes, best to return a pointer to them
- Upon return, caller finds the return value in `%rax`

Data Flow Examples

```
int multstore(int x, int y)
{
    return mult2(x, y);
}
```

```
0000000000400540 <multstore>:
  # x in %rdi, y in %rsi, dest in %rdx
    • • •
  400541: movq    %rdx,%rbx       # Save dest
  400544: call    400550 <mult2> # mult2(x,y)
  # t in %rax
  400549: movq    %rax,(%rbx)     # Save at dest
    • • •
```

```
int mult2(int a, int b)
{
  return a * b;
}
```

```
0000000000400550 <mult2>:
  # a in %rdi, b in %rsi
  400550:  movq   %rdi,%rax   # a
  400553:  imulq  %rsi,%rax   # a * b
  # s in %rax
  400557:  ret                # Return
```

Stack-Based Languages

- Languages that support recursion
  - e.g. C, Java, most modern languages
  - Code must be **re-entrant** – need to be able to call the same function multiple times
  - Need some place to store state of each call
    - Arguments, local variables, return pointer
- Stack allocated in frames
  - State for a single procedure instantiation
- Stack discipline
  - State for a given procedure needed for a limited time – from when it is called to when it returns
  - Callee always returns before caller does

Recursion Example

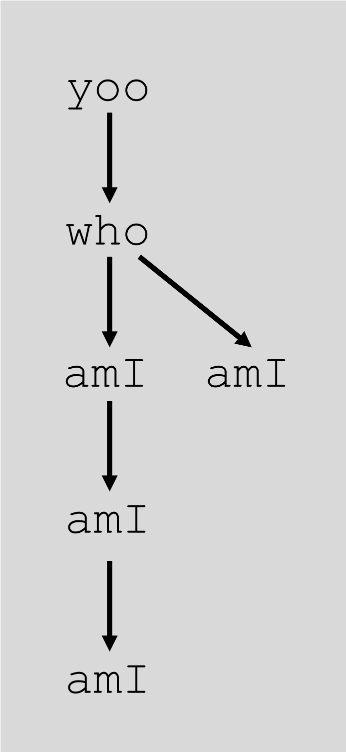
```
yoo(…)
{
    •
    •
    who();
    •
    •
}
```

```
who(…)
{
    •
    amI();
    •
    amI();
    •
}
```

```
amI(…)
{
    •
    if(…){
      amI()
    }
    •
}
```

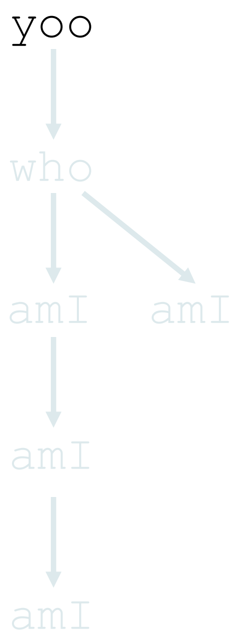
Procedure `amI` is recursive
(calls itself)

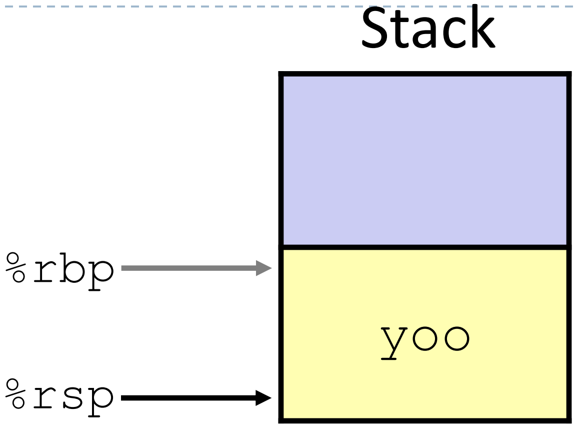
Example
Call Chain



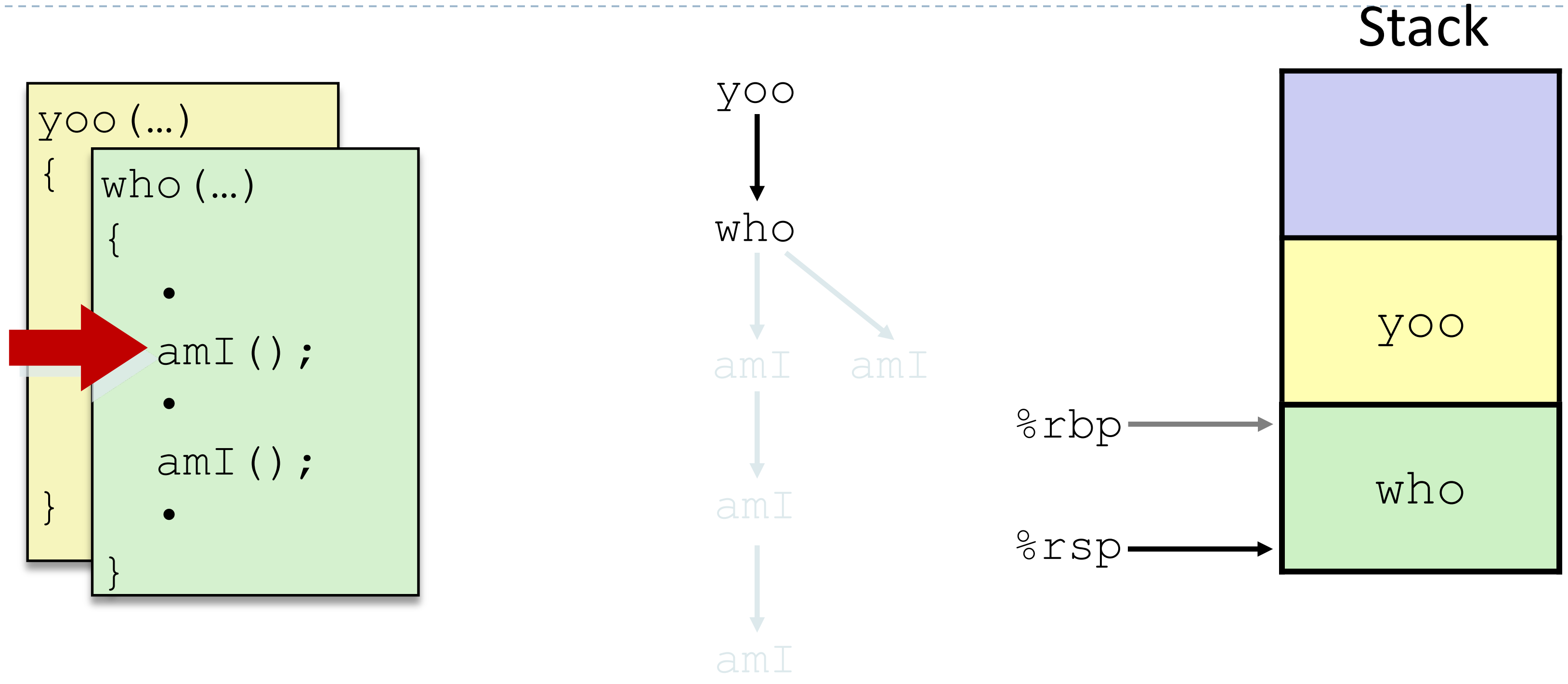
1) Call to yoo

```
yoo(…)
{
  •
  •
  who();
  •
  •
}
```

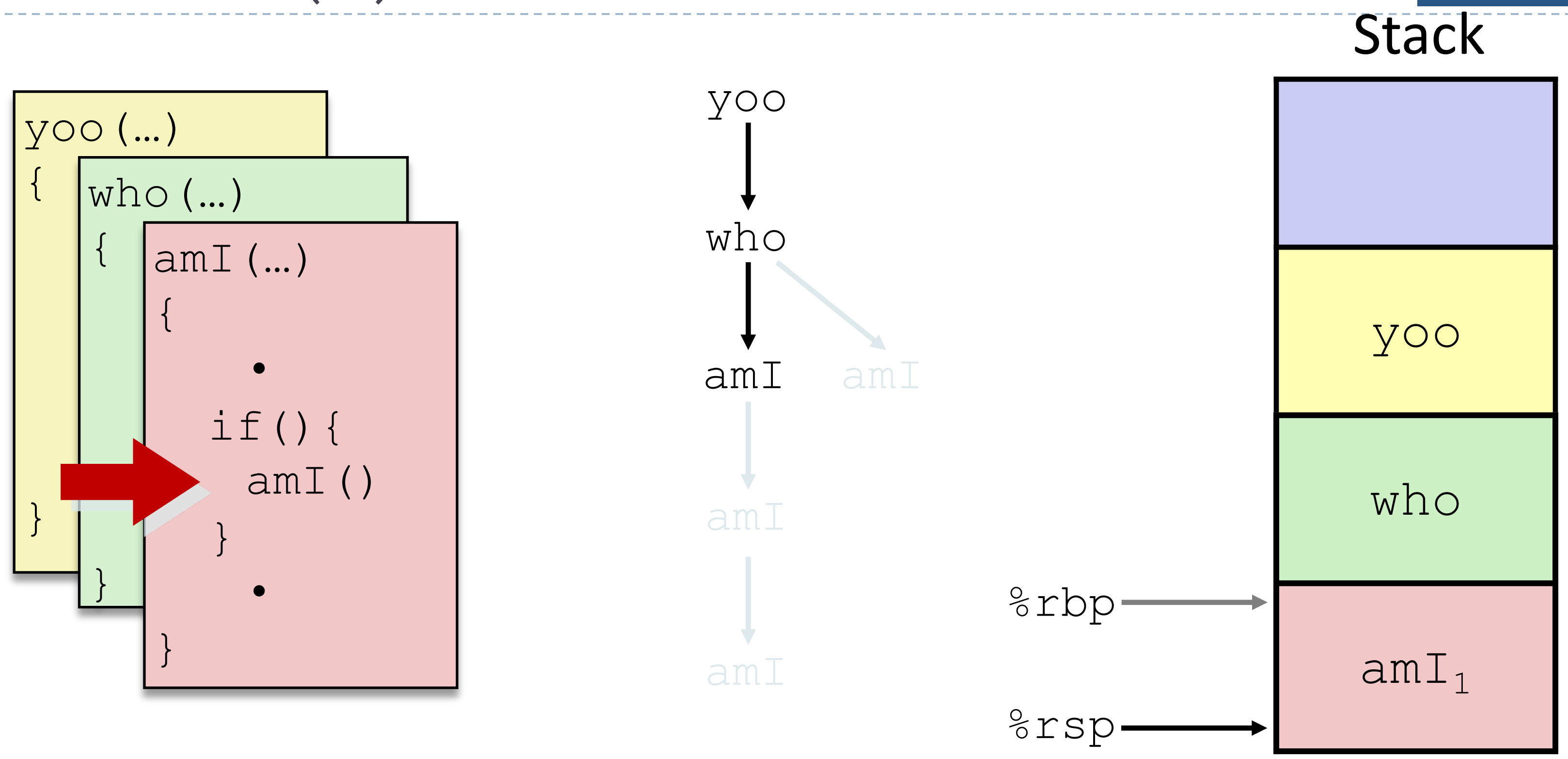




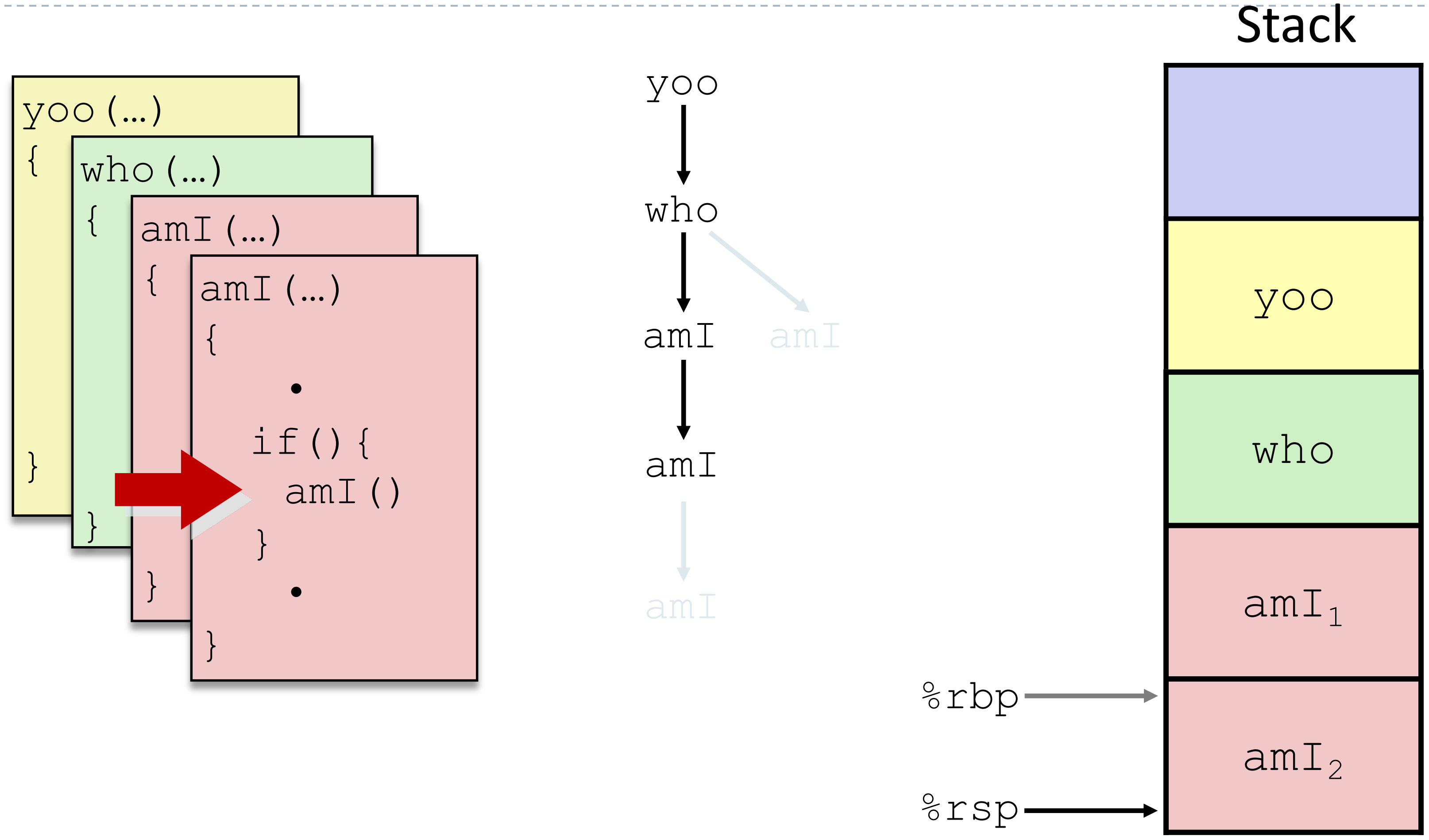
2) Call to who



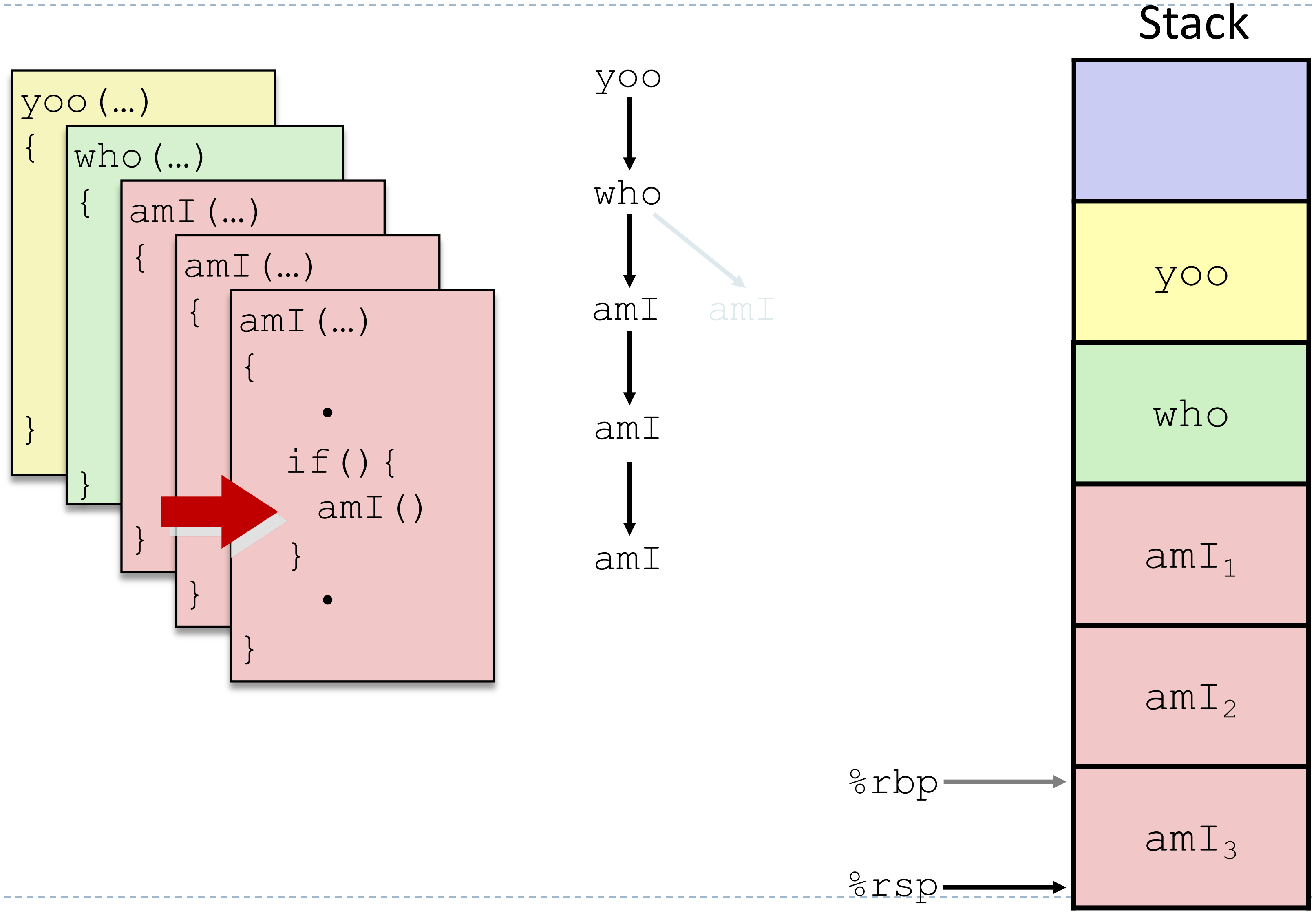
3) Call to amI (1)



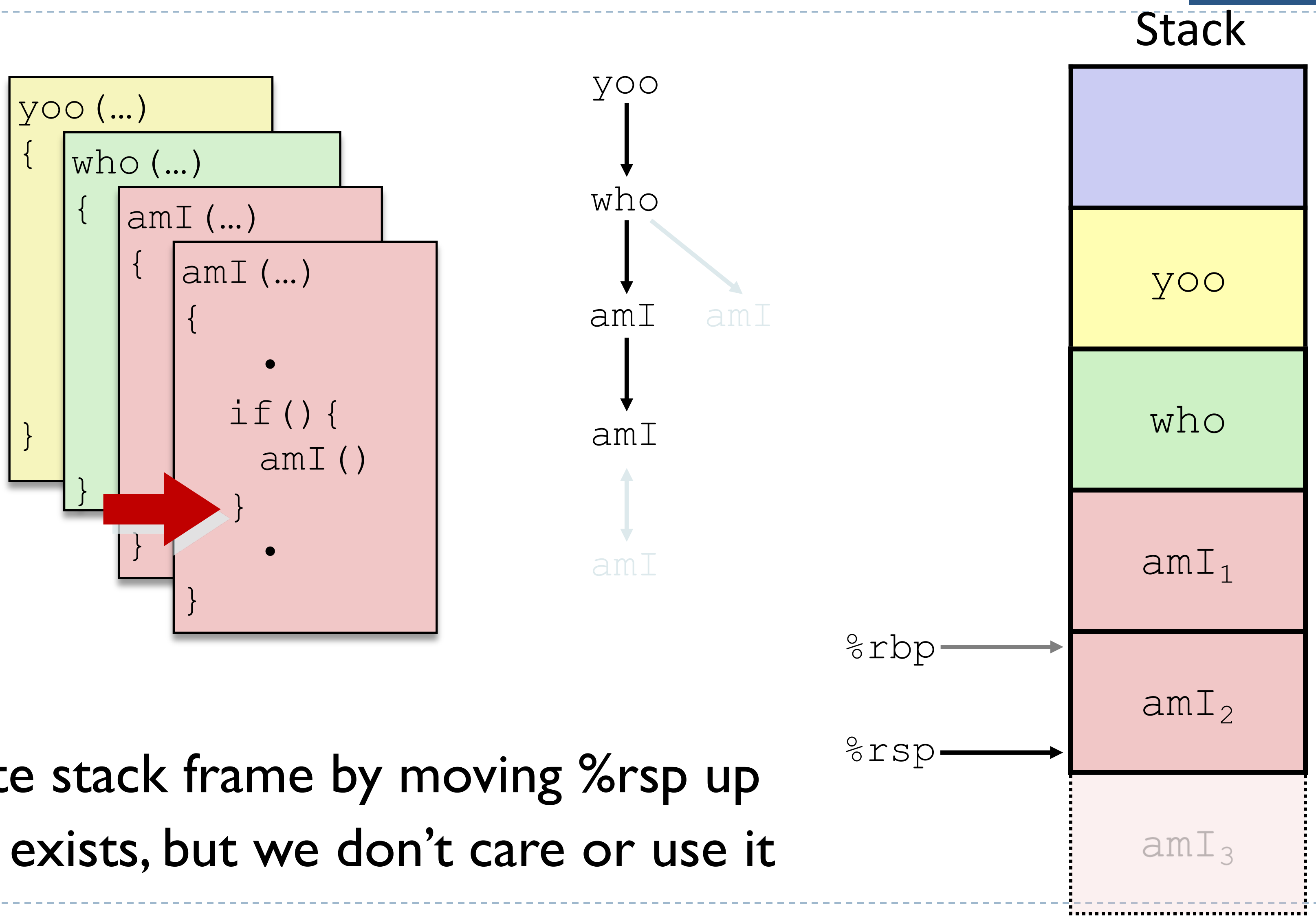
4) Recursive call to amI (2)



5) Another recursive call to amI (3)

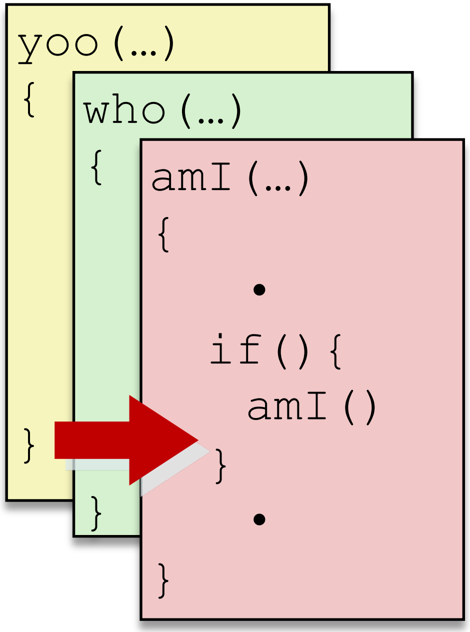


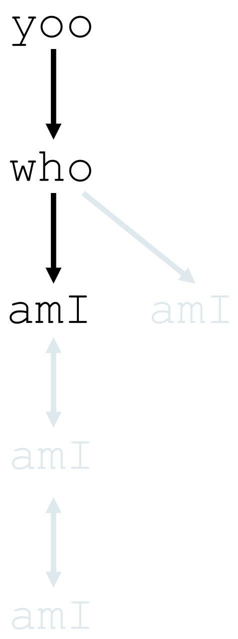
6) Return from recursive call to amI

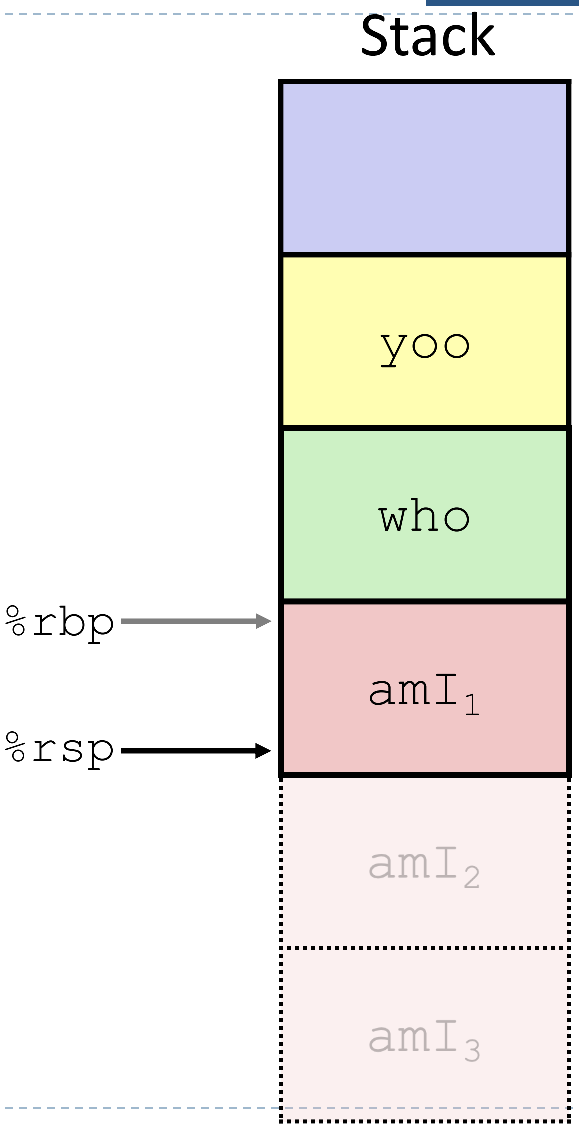


- Deallocate stack frame by moving %rsp up
- Data still exists, but we don't care or use it

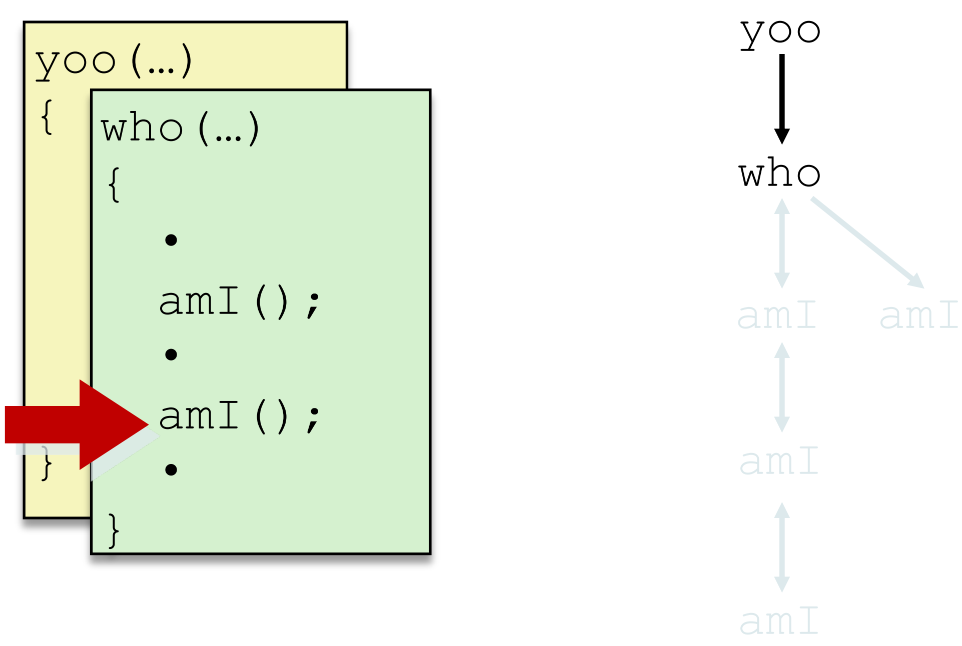
7) Return from recursive call to amI

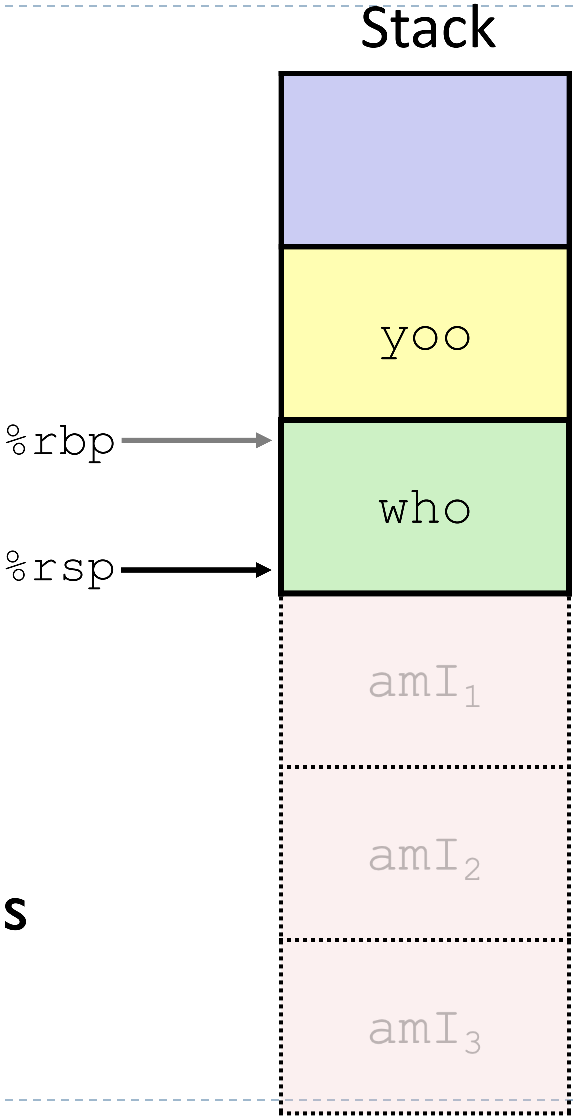






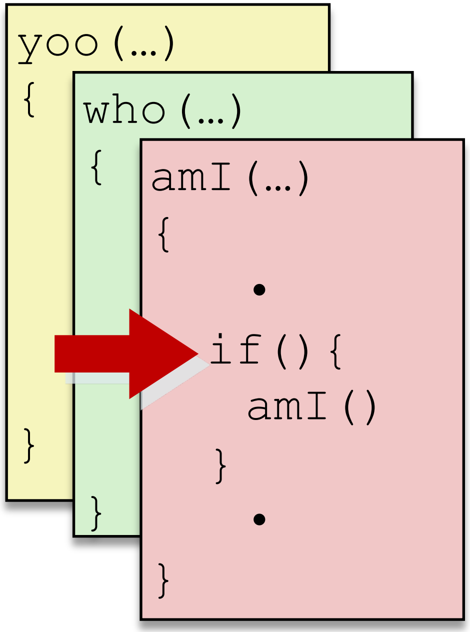
8) Return from call to amI

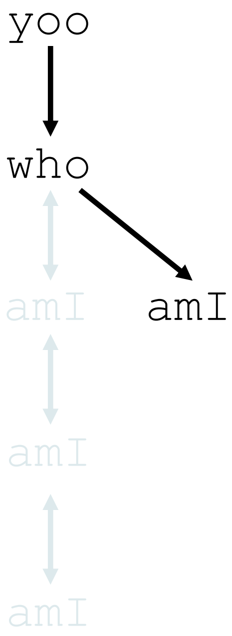


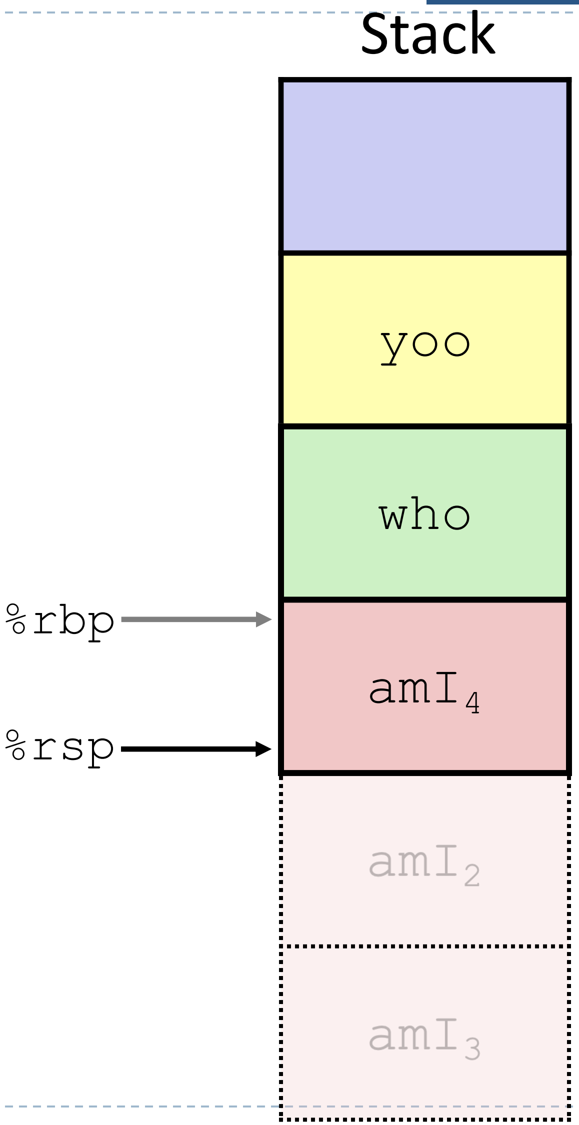


- At this point, we're back up to who, which calls amI a second time

9) Second Call to amI (4)



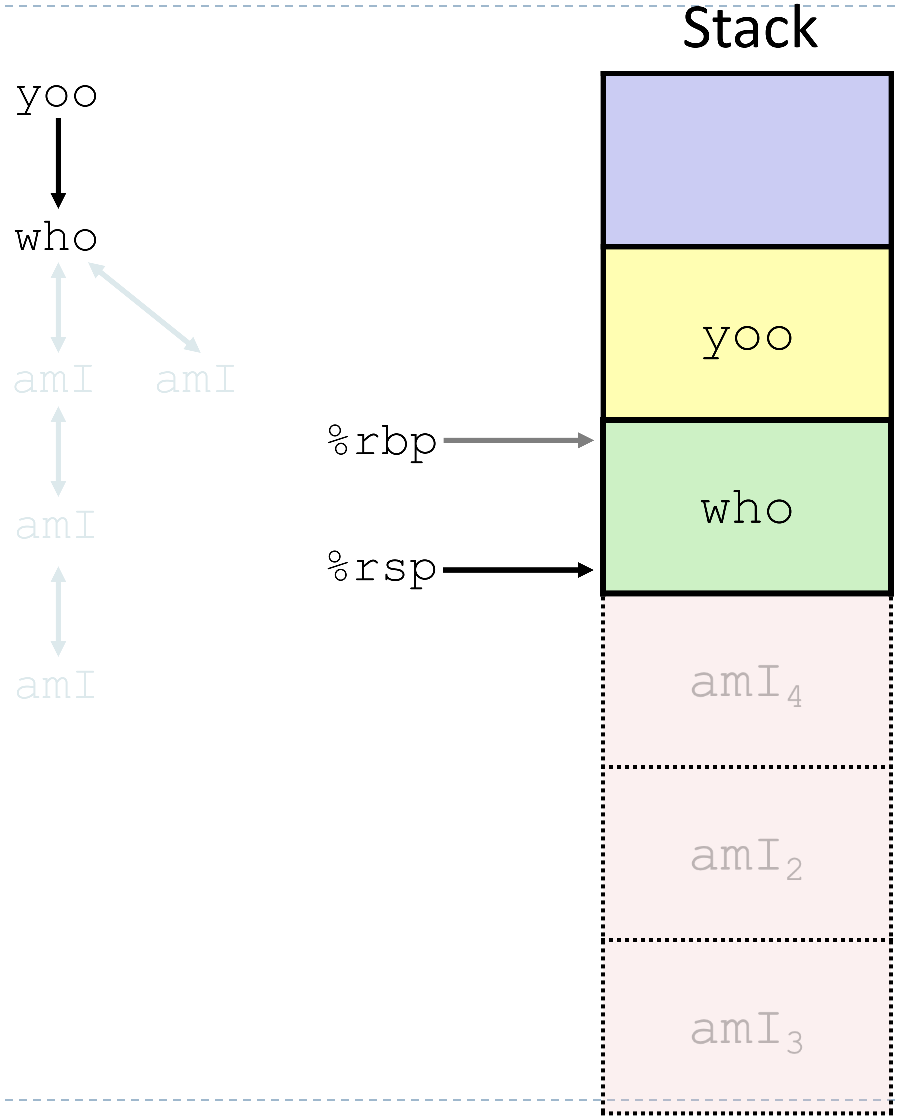




- Overwriting the stack from amI_3
- Similar to Step 3, but assume if() condition is false
- No recursion.

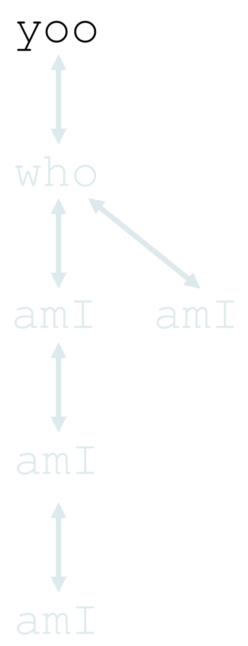
10) Return from second call to amI

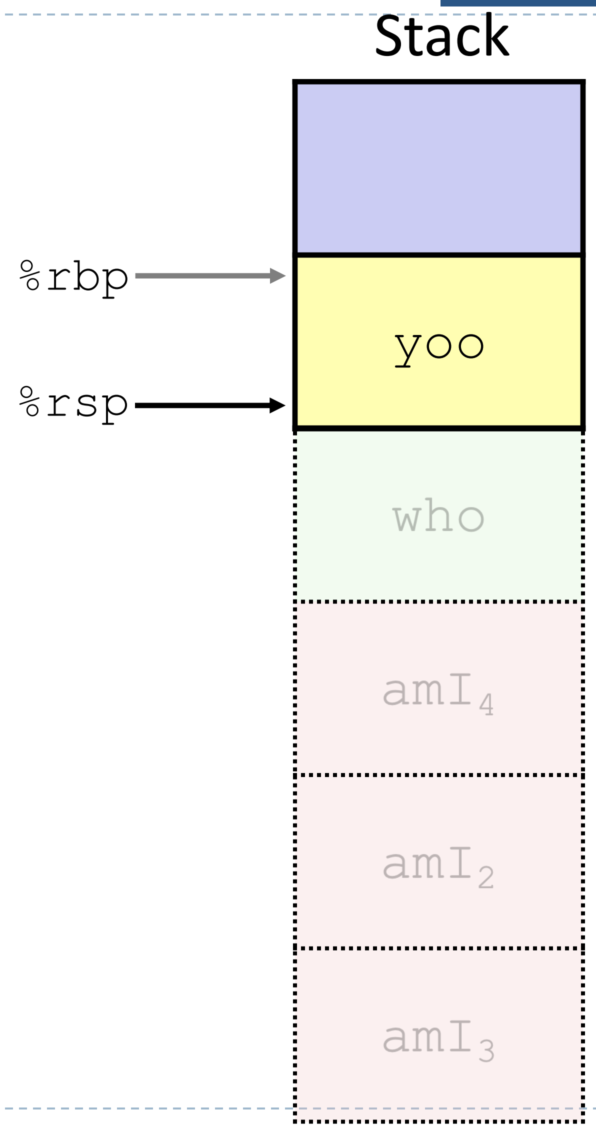
```
yoo(…)
{
  who(…)
  {
     •
     amI();
     •
     amI();
     •
  }
```



11) Return from call to who

```
yoo(…)
{
  •
  •
  who();
  •
  •
}
```

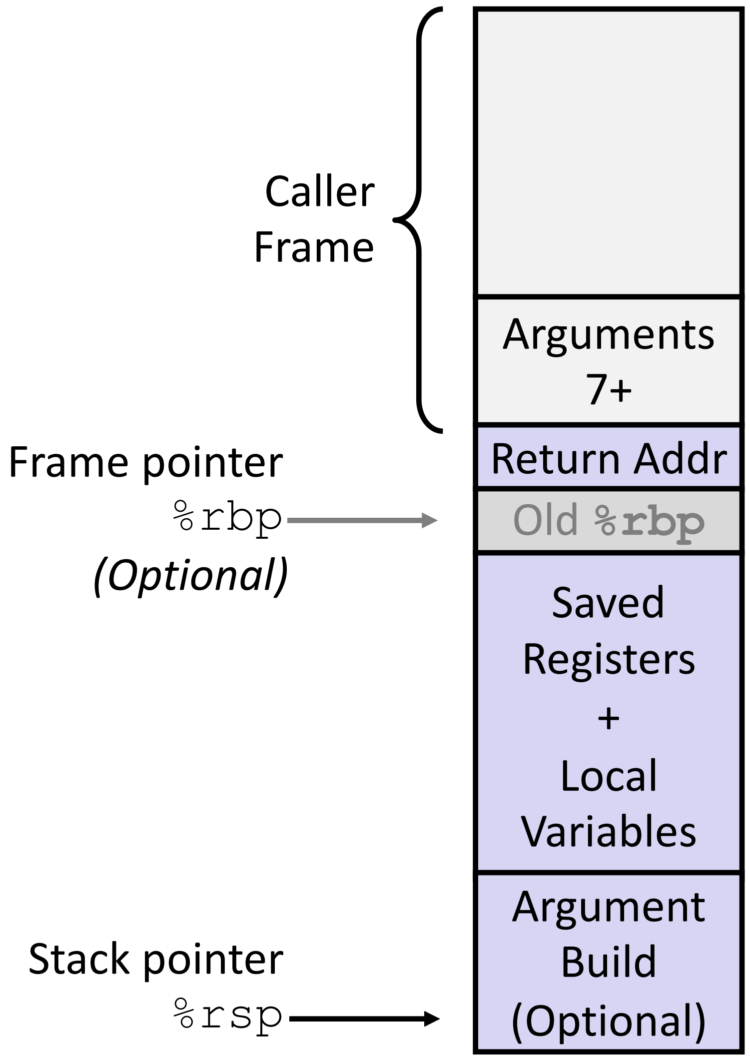




- Total stack frames created: 7
- Maximum stack depth: 6 frames

x86-64/Linux Stack Frame

- Caller's Stack Frame
  - Extra arguments (if > 6 args) for this call
- Current / Callee Stack Frame
  - Return address pushed by call instruction
  - Old frame pointer (optional)
  - Saved register context (when reusing registers)
  - Local variables (If can't be kept in registers)
  - "Argument build" area
    (If callee needs to call another function - parameters for function about to call, if needed)



Register Saving Conventions

- When procedure **yoo** calls **who**:
  - yoo is the caller
  - who is the callee
- Can registers be used for temporary storage?

```
yoo:
   • • •
   movq $15213, %rdx
   call who
   addq %rdx, %rax
   • • •
   ret
```

```
who:
   • • •
   subq $18213, %rdx
   • • •
   ret
```

  - No! Contents of register %rdx overwritten by who!
  - This could be trouble – something should be done. Either:
    - Caller should save %rdx before the call (and restore it after the call)
    - Callee should save %rdx before using it (and restore it before returning)

Register Saving Conventions

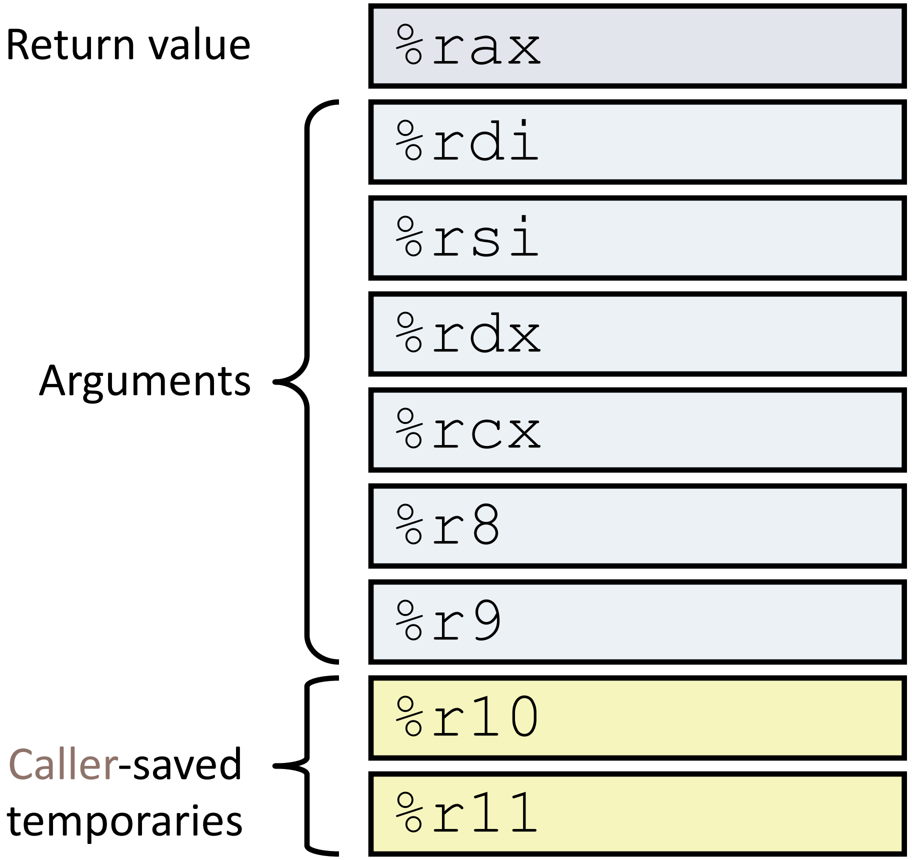
- "Caller-saved" registers
  - It is the caller's responsibility to save any important data in these registers before calling another procedure (i.e. the callee can freely change data in these registers)
  - Caller saves values in its stack frame before calling Callee, then restores values after the call
- "Callee-saved" registers
  - It is the callee's responsibility to save any data in these registers before using the registers (i.e. the caller assumes the data will be the same across the callee procedure call)
  - Callee saves values in its stack frame before using, then restores them before returning to caller

Silly Register Convention Analogy

- Parents (caller) leave for the weekend and give the keys to the house to their child (callee)
  - Being suspicious, they put away/hid the valuables (caller-saved) before leaving
  - Warn child to leave the bedrooms untouched: "These rooms better look the same when we return!"
- Child decides to throw a wild party (computation), spanning the entire house
  - To avoid being disowned, child moves all of the stuff from the bedrooms to the backyard shed (callee-saved) before the guests trash the house
  - Child cleans up house after the party and moves stuff back to bedrooms
- Parents return home and are satisfied with the state of the house
  - Move valuables back and continue with their lives

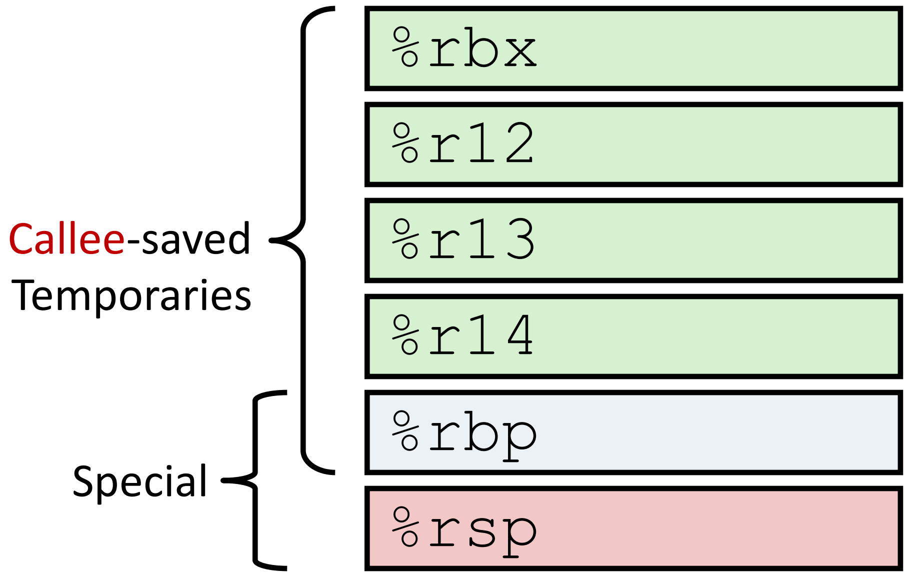
x86-64 Linux Register Usage (1)

- %rax
  - Return value
  - Also caller-saved & restored
  - Can be modified by procedure
- %rdi, ..., %r9
  - Arguments
  - Also caller-saved & restored
  - Can be modified by procedure
- %r10, %r11
  - Caller-saved & restored
  - Can be modified by procedure



x86-64 Linux Register Usage (2)

- %rbx, %r12, %r13, %r14
  - Callee-saved
  - Callee must save & restore
- %rbp
  - Callee-saved
  - Callee must save & restore
  - May be used as frame pointer
  - Can mix & match
- %rsp
  - Special form of callee save
  - Restored to original value upon exit from procedure



x86-64 64-bit Registers Usage

`%rax`	Return value - Caller saved
`%rbx`	Callee saved
`%rcx`	Argument #4 - Caller saved
`%rdx`	Argument #3 - Caller saved
`%rsi`	Argument #2 - Caller saved
`%rdi`	Argument #1 - Caller saved
`%rsp`	Stack pointer
`%rbp`	Callee saved

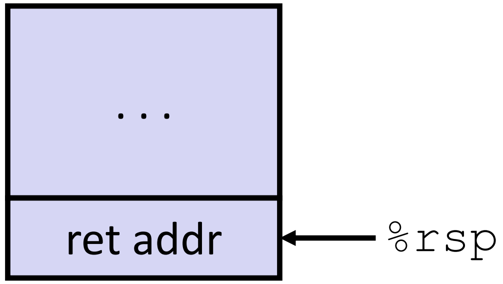
`%r8`	Argument #5 - Caller saved
`%r9`	Argument #6 - Caller saved
`%r10`	Caller saved
`%r11`	Caller Saved
`%r12`	Callee saved
`%r13`	Callee saved
`%r14`	Callee saved
`%r15`	Callee saved

Callee-Saved Example (step 1)

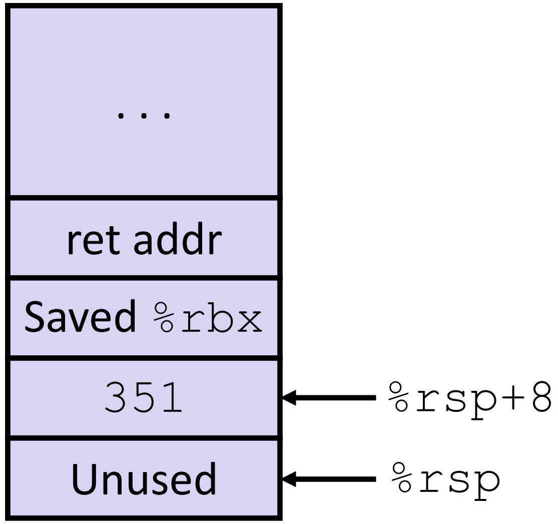
```
int call_incr2(int x) {
    int v1 = 351;
    int v2 = increment(v1, 100);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $351, 8(%rsp)
  movl    $100, %rsi
  leaq    8(%rsp), %rdi
  call    increment
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Initial Stack Structure



Resulting Stack Structure

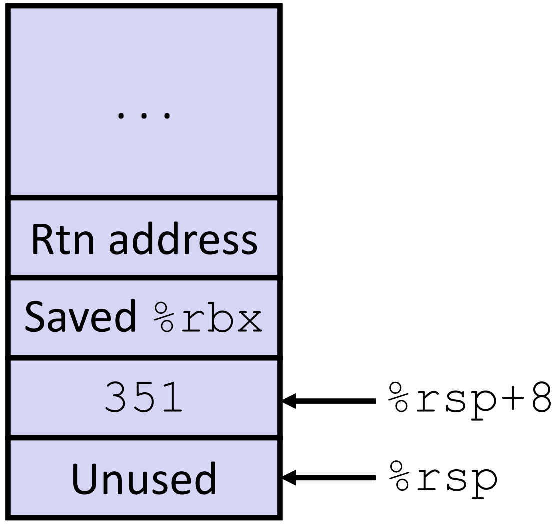


Callee-Saved Example (step 2)

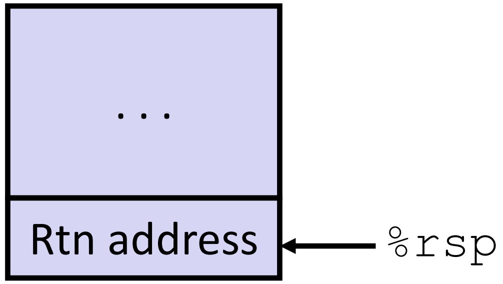
```
int call_incr2(int x) {
    int v1 = 351;
    int v2 = increment(v1, 100);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $351, 8(%rsp)
  movq    $100, %rsi
  leaq    8(%rsp), %rdi
  call    increment
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Stack Structure



Pre-return Stack Structure



Why Caller and Callee Saved?

- We want one calling convention to simply separate implementation details between caller and callee
- In general, neither caller-save nor callee-save is "best":
  - If caller isn't using a register, caller-save is better
  - If callee doesn't need a register, callee-save is better
  - If "do need to save", callee-save generally makes smaller programs
    - Functions are called from multiple places
- So… "some of each" and compiler tries to "pick registers" that minimize amount of saving/restoring

Register Conventions Summary

- Caller-saved register values need to be pushed onto the stack before making a procedure call only if the Caller needs that value later
  - Callee may change those register values
- Callee-saved register values need to be pushed onto the stack only if the Callee intends to use those registers
  - Caller expects unchanged values in those registers
- Don’t forget to restore/pop the values later!

Observations About Recursion

- Works without any special consideration
  - Stack frames mean that each function call has private storage
    - Saved registers & local variables
    - Saved return pointer
  - Register saving conventions prevent one function call from corrupting another's data
    - Unless the code explicitly does so (e.g. buffer overflow)
  - Stack discipline follows call / return pattern
    - If P calls Q, then Q returns before P
    - Last-In, First-Out (LIFO)
- Also works for mutual recursion (P calls Q; Q calls P)

x86-64 Stack Frames

- Many x86-64 procedures have a minimal stack frame
  - Only return address is pushed onto the stack when procedure is called

- A procedure needs to grow its stack frame when it:
  - Has too many local variables to hold in caller-saved registers
  - Has local variables that are arrays or structs/objects
  - Calls another function that takes more than six arguments
  - Is using caller-saved registers and then calls a procedure
  - Modifies/uses callee-saved registers

Example: Code for obj.func(e1,e2,…en)

- In principal the code should work like this:
  - Visit obj – leaves reference to object (”this”) in %rax
  - Push obj pointer to stack: gen(push %rax)
  - Visit e1, e2, …, en. For each argument, push to stack: gen(push %rax)
  - generate code to load method table pointer 0(%rdi) into %rax
  - Pop the stack to argument registers in reverse order
    - gen(popq correct_argument_register)
  - generate call instruction with indirect jump

```
public void visit(Call n)  {
   final String c_regs[] = {"%rdi","%rsi","%rdx","%rcx","%r8","%r9"};

   system.out.println("pushq " + getExpr(n.e));
   for (int i = 0; i < n.el.size(); i++) {
      system.out.println("pushq " + getExpr(n.el.get(i)));
   }

   for (int i = n.el.size(); i >= 0; i--) {      // pop in reverse order
      system.out.println(”popq " + c_regs[i+1]);
   }
   system.out.println("call " + getLabel(n.i) );
}
```

```
_Foo:
          pushq   %rbp
          movq    %rsp, %rbp
          subq    $64, %rsp
          movq    %rdi, -8(%rbp)
          movq    %rsi, -16(%rbp)
          movq    %rdx, -24(%rbp)
          movq    %rcx, -32(%rbp)
          movq    %r8, -40(%rbp)
          movq    %r9, -48(%rbp)
             :
          movq    ..., %rax
          leave
          ret
```

```
_Bar:        :
          popq    %r9
          popq    %r8
          popq    %rcx
          popq    %rdx
          popq    %rsi
          popq    %rdi
          call    _Foo
```

Method Call Complications

- Big one: code to evaluate any argument might clobber argument registers (i.e., computing an argument value might require a method call)
  - Make stack frame big enough to store arguments, local variables and temporaries
  - Push arguments on the stack immediately, then pop off just before method call
- Other one: what if a method has too many parameters?
  - Okay to assume all methods have ≤ 5 parameters plus "this" – do better if you want
  - Make sure that Sematic Analysis checks for number of parameters.

Code Gen for Method Definitions

- Generate label for method
  - Classname$methodname:
- Upon entry, a callee needs to initialize its linkage and stack frame. This method prologue is accomplished by the following sequence:
  - push %rbp,
  - movq %rsp, %rbp
  - subq $*N*, %rsp
- Subtract frame size from %rsp – how do we calculate the frame size?
- Visit statements in order
  - Method epilogue is normally generated as part of a return statement (next)
  - In MiniJava the return is generated after visiting the method body to generate its code

Method Arguments & Local Variables

- Method parameters are in registers
- But code generated for methods also will be using the registers
- So how do we avoid clobbering parameters & local variables?
- Strategy:
  - Allocate enough space in the stack frame
  - Save copies of all parameter registers to stack on method entry
  - Use stack location when you need to reference a parameter – how do we track the location?

```
public void visit(Formal n) {
     Symbol sym = st.lookup(n.i.s);
     if ( sym != null && sym instanceof VarSymbol ) {
        VarSymbol vs = (VarSymbol)sym;
        String stack_loc = Integer.toString(stack_pos) + "(%rbp)";
        stack_pos -= 8;
        stack_table.put(vs, stack_loc);  // map sym to stack location for later
        // print mov instr - lookup argument register based on current position
        System.out.println("movq " + call_regs[arg_pos++] + ", ” + stack_loc);
     }
}
```

```
movq    %rdi,  -8(%rbp)
movq    %rsi, -16(%rbp)
movq    %rdx, -24(%rbp)
movq    %rcx, -32(%rbp)
movq    %r8,  -40(%rbp)
movq    %r9,  -48(%rbp)
```

Extended Objects & Overriding Methods

```
class One {
  int tag;
  int it;
  void setTag()       { tag = 1; }
  int getTag()        { return tag; }
  void setIt(int it)  { this.it = it; }
  int getIt()         { return it; }
}

class Two extends One {
  int it;
  void setTag()       { tag = 2;  it = 3;}
  int getThat()       { return it; }
  void resetIt()      { super.setIt(42); }
}
```

```
public static void main(String[] args) {
    Two two = new Two();
    One one = two;

    one.setTag();        Which setTag() is this?
    System.out.println(one.getTag());

    one.setIt(17);
    two.setTag();
    System.out.println(two.getIt());
    System.out.println(two.getThat());

    two.resetIt();
    System.out.println(two.getIt());
    System.out.println(two.getThat());
}
```

Object Representation

- The naïve explanation is that an object contains
  - Fields declared in its class and in all superclasses
    - Redeclaration of a field hides (shadows) superclass instance – but the superclass field is still there
  - Methods declared in its class and all superclasses
    - Redeclaration of a method overrides (replaces) – but overridden methods can still be accessed by super

- When a method is called, the method "inside" that particular object is called
  - (But we really don't want to copy all those methods, do we?)

Actual representation

- Each object contains:
  - Storage for every field (instance variable)
    - Including all inherited fields (public or private or …)
  - A pointer to a runtime data structure for its class
    - Key component: method dispatch table (next slide)
- An object is basically a C struct
- Fields hidden (shadowed) by declarations in subclasses are still allocated in the object and are accessible from superclass methods

Method Dispatch Tables

- One of these per class, not per object
- Often called “vtable”, virtual function table, or dynamic dispatch table
- One pointer per method – points to beginning of method code
- Dispatch table (vtable) offsets fixed at compile time

Method Tables and Inheritance

- An initial, really simple implementation
  - Method table for each class has pointers to all methods declared in it
  - Method table also contains a pointer to parent class method table
  - Method dispatch
    - Look in current table and use if method declared locally
    - Look in parent class table if not local
    - Repeat
    - “Message not understood” if you can’t find it after search
  - Actually used in typical implementations of some dynamic languages

O(1) Method Dispatch

- Idea: First part of method table for extended class has pointers for the same methods in the same order as the parent class
  - BUT pointers actually refer to overriding methods if these exist
  - Method dispatch can be done with indirect jump using fixed offsets known at compile time – O(1)
    - In C: *(object->vtbl[offset])(parameters)
- Pointers to additional methods defined (added) in subclass are included in the table following inherited/overridden ones from superclass(es)

MiniJava Method Tables (vtbls)

- Generate these as initialized data in the assembly language source program
- Need to pick a naming convention for assembly language labels
  - For methods, classname$methodname
    - Would need something more sophisticated for overloading
  - For the vtables themselves, classname$$
- First method table entry points to superclass table (we might not use this in our project, but is helpful if you add instanceof or type cast checks)

Method Tables For Previous Example

```
class One {
   void setTag()    { … }
   int getTag()     { … }
   void setIt(int it) {…}
   int getIt()      { … }
}

class Two extends One {
   void setTag() { … }
   int getThat() { … }
   void resetIt() { … }
}
```

```
.data
One$$:   .quad  0         # no superclass
         .quad  One$setTag
         .quad  One$getTag
         .quad  One$setIt
         .quad  One$getIt
Two$$:   .quad  One$$     # superclass
         .quad  Two$setTag
         .quad  One$getTag
         .quad  One$setIt
         .quad  One$getIt
         .quad  Two$getThat
         .quad  Two$resetIt
```

Method Table Layout

- Key point: First entries in Two's method table are pointers to methods in exactly the same order as in One's method table
  - Actual pointers reference code appropriate for objects of each class (inherited or overridden)
- Compiler knows correct offset for a particular method pointer regardless of whether that method is overridden and regardless of the actual (dynamic) type of the object

Object Layout

- Typically, allocate fields sequentially
- Follow processor/OS alignment conventions for struct/object when appropriate/available
  - Include padding bytes for alignment as needed
- Use first word of object for pointer to method table/class information
- Objects are allocated on the heap
  - No actual storage bits in the generated code

Object Field Access

- Source
  - int n = obj.fld;
- x86-64
  - Assuming that obj is a local variable in the current method's stack frame
    - movq offsetobj(%rbp),%rax # load obj ptr
    - movq offsetfld(%rax),%rax # load fld
    - movq %rax,offsetn(%rbp) # store n (assignment stmt)
  - Same idea used to reference fields of "this"
    - Use implicit "this" parameter passed to methods instead of a local variable to get object address

Local Fields

- A method can refer to fields in the receiving object either explicitly as “this.f” or implicitly as “f”
  - Both compile to the same code – an implicit “this.” is assumed if not present explicitly
  - A pointer to the object (i.e., “this”) is an implicit, hidden parameter to all methods

Source Level View

- What you write:

```
int getIt() {
  return it;
}
void setIt(int it) {
  this.it = it;
}
…
obj.setIt(42);
k = obj.getIt();
```

- What you really get:

```
int getIt(Objtype this) {
  return this.it;
}
void setIt(ObjType this, int it) {
  this.it = it;
}
…
setIt(obj, 42);
k = getIt(obj);
```

x86-64 "this" Convention

- "this" is an implicit first parameter to every non-static method
- Address of object ("this") placed in %rdi for every non-static method call
- Remaining parameters (if any) in %rsi, etc.

- We'll use this convention in our project

Object Creation – new

- Steps needed
  - Call storage manager (malloc or equivalent) to get the raw bits
  - Initialize bytes to 0 (for Java, not in e.g., C++)
  - Store pointer to method table in the first 8 bytes of the object
  - Call a constructor with "this" pointer to the new object in %rdi and other parameters as needed
    - (Not in MiniJava since we don't have constructors)
  - Result of new is a pointer to the new object

Object Creation

- Source
  - One one = new One(…);
- x86-64
  - movq $nBytesNeeded,%rdi # obj size + 8 (include space for vtbl ptr)
  - call mallocEquiv # addr of allocated bytes returned in %rax
  - <zero out allocated object, or use calloc instead of malloc to get bytes>
  - leaq One$$,%rdx # get method table address
  - movq %rdx,0(%rax) # store vtbl ptr at beginning of object
  - movq %rax,%rdi # set up "this" for constructor
  - movq %rax,offsettemp(%rbp) # save "this" for later (or maybe pushq)
  - <load constructor arguments> # arguments (if needed)
  - call One$One # call ctor if we have one (no vtbl lookup)
  - movq offsettemp(%rbp),%rax # recover ptr to object
  - movq %rax,offsetone(%rbp) # store object reference in variable one

Constructor

- Why don't we need a vtable lookup to find the right constructor to call?

- Because at compile time we know the actual class (it says so right after "new"), so we can generate a call instruction to a known label
  - Same with super.method(…) or superclass constructor calls – at compile time we know all of the superclasses (need this to compile subclass and construct method tables), so we know statically what class "super.method" belongs to

Method Calls

- Steps needed
  - Parameter passing: just like an ordinary C function, except load a pointer to the object in %rdi as the first ("this") argument
  - Get a pointer to the object's method table from the first 8 bytes of the object
  - Jump indirectly through the method table

Method Call

- Source
  - obj.method(…);
- x86-64
  - <load arguments in registers as usual> # as needed
  - movq offsetobj(%rbp),%rdi # first argument is obj ptr (“this”)
  - movq 0(%rdi),%rax # load vtable address into %rax
  - call *offsetmethod(%rax) # call function whose address is at
  - # the specified offset in the vtable *

- *Can get same effect with:
  - addq $offsetmethod,%rax
  - call *(%rax)
- or with:
  - movq $offsetmethod(%rax),%rax
  - call *%rax

Generating Assembly Code

- Suggestion: isolate the actual compiler output operations in a handful of routines
  - Usual modularity reasons & saves some typing
  - Possibilities

```
// write code string s to .asm output
void gen(String s) { … }
// write "op  src,dst" to .asm output
void genbin(String op, String src, String dst) { … }
// write label L to .asm output as "L:"
void genLabel(String L) { … }
```

  - A handful of these methods should do it

External Names

- In a Linux environment, an external symbol is used as-is (xyzzy)
- In Windows and OS X, an external symbol xyzzy is written in asm code as _xyzzy (leading underscore)
- Your compiler needs to generate code that runs on attu using the correct convention depending on which OS it's running.

```
public String getLabel(String class_name, String call_name)
{
    String label = !class_name.isEmpty() ? class_name+"$"+call_name : call_name;
    String os = System.getProperty("os.name");
    if ( os.contains("Windows") || os.contains("OS X") ) {
        return "_"+label;
    }
    return label;
}
```

A Simple Code Generation Strategy

- Goal: quick 'n dirty correct code, optimize later if time
- Traverse AST primarily in execution order and emit code during the traversal
  - Visitor might want to traverse the tree in ad-hoc ways depending on sequence that parts need to appear in the code
- Treat the x86-64 as a 1-register machine with a stack for additional intermediate values
- Store all values (reference, int, boolean) in 64-bit quadwords
  - Natural size for 64-bit pointers, i.e., object references (variables of class types)

x86 as a Stack Machine

- Idea: Use x86-64 stack for expression evaluation with %rax as the "top" of the stack
- Invariant: Whenever an expression (or part of one) is evaluated at runtime, the generated code leaves the result in %rax
- If a value needs to be preserved while another expression is evaluated, push %rax, evaluate, then pop when first value is needed
  - Remember: ***always* pop what you push**
  - Will produce lots of redundant, but correct, code
- Examples below follow code shape examples, but with some details about where code generation fits

System.out.println(exp)

- MiniJava's "print" statement

  <compile exp; result in %rax>

  ```
  movq    %rax,%rdi    # load argument register
  call    put          # call external put routine
  ```

- If the stack is not kept 16-byte aligned, calls to external library code can cause a runtime error (will cause on OS X)

Constants and Identifiers

- Integer constants, say 17
  - gen(movq $17, %rax)
    - leaves value in %rax
- Local variables (any type – int, bool, reference)
  - gen(movq varoffset(%rbp), %rax)

Expressions

- Recall for operations:
  - The RHS is also the destination
  - The RHS may only be a register or memory (not an immediate)
  - Operations do not support memory in both LHS and RHS
- Strategy:
  - Consider separate functions to handle LHS and RHS
  - Immediate or Memory (stack) in RHS should be moved to %rax
  - Inlined expressions (a + (b * c)) should move temporary results to stack

```
pushq   %rax              // push a
movq    -8(%rbp), %rax    // c
imulq   -16(%rbp), %rax   // b * c
popq    %rdx              // pop a
addq    %rdx, %rax        // a + (b * c)
pushq   %rax              // push result
```

addq	reg_1, reg_2	$reg_2 \leftarrow reg_2 + reg_1$
addq	reg, mem	$\mathrm{M}[mem] \leftarrow \mathrm{M}[mem] + reg$
addq	$imm32, reg$	$reg \leftarrow reg + imm32$
addq	$imm32, mem$	$\mathrm{M}[mem] \leftarrow \mathrm{M}[mem] + imm32$
addq	mem, reg	$reg \leftarrow reg + \mathrm{M}[mem]$

Binary Expressions: exp1 + exp1

- Visit exp1
  - generates code to evaluate exp1 with result in %rax
- gen(pushq %rax)
  - push exp1 onto stack
- Visit exp2
  - generates code for exp2; result in %rax
- gen(popq %rdx)
  - pop left argument into %rdx; cleans up stack
- gen(addq %rdx,%rax)
  - perform the addition; result in %rax

Assignment Statements: var = exp (1)

- Assuming that var is a local variable
  - Visit node for exp
  - Generates code to eval exp and leave result in %rax
  - gen(movq %rax, offset_of_variable(%rbp))

Assignment Statements: var = exp (2)

- If var is a more complex expression (object or array reference, for example)
  - visit var
  - gen(pushq %rax)
    - push lvalue (address) of variable or object containing variable onto stack
  - visit exp
    - leaves rhs value in %rax
  - gen(popq %rdx)
  - gen(movq %rax, appropriate_offset(%rdx))

Processing Expressions

```
public String getExpr(ASTNode n)
{
    if ( n == null ) {
        return "";
    }
    else if ( n instanceof IntegerLiteral ) {
        IntegerLiteral i = (IntegerLiteral)n;
        i.accept(this);
        return "%rax";
    }
    else if ( n instanceof True ) {
        True i = (True)n;
        i.accept(this);
        return "%rax";
    }
    else if ( n instanceof False ) {
        False i = (False)n;
        i.accept(this);
        return "%rax";
    }
    else if ( n instanceof IdentifierExp ) {
        IdentifierExp i = (IdentifierExp)n;
        Symbol s = st.getSymbol(i.s);
        :
    }
    else if ( n instanceof Identifier ) {
        Identifier i = (Identifier)n;
        Symbol s = st.getSymbol(i.s);
        :
    }
    else if ( n instanceof ArrayLookup ) {
        ArrayLookup e = (ArrayLookup)n;
        e.accept(this);
        return "%rax";
    }
    else if ( n instanceof Exp ) {
        Exp e = (Exp)n;
        e.accept(this);
        return "%rax";
    }
    report_error( n.getLineNo(), "Undefined
expression.");
    return "";
}
```

Function Calls: return exp;

- Visit exp; this leaves result in %rax where it needs to be
- Generate method epilogue to unwind the stack frame
  - movq %rpb, %rsp
  - pop %rpb
  - ret
- The **leave** instruction sets the stack pointer (%rsp) to the frame pointer (%rbp) and then sets the frame pointer to the saved frame pointer, which is popped from the stack:
  - leave
  - ret

Control Flow: Unique Labels

- Needed in code generator: a String-valued method that returns a different label each time it is called (e.g., L1, L2, L3, …)
- Improvement: a set of methods that generate different kinds of labels for different constructs (can really help readability of the generated code)
  - while1, while2, while3, …;
  - if1, if2, …;
  - else1, else2, …;
  - fi1, fi2, … .

Control Flow: Tests

- Recall that the context for compiling a boolean expression is
  - Label or address of jump target
  - Whether to jump if true or false
- So the visitor for a boolean expression should receive this information from the parent node

Join Points

- Loops and conditional statements have join points where execution paths merge
- Generated code must ensure that machine state will be consistent regardless of which path is taken to get there
  - i.e., the paths through an if-else statement must not leave a different number of values pushed onto the stack
  - If we want a particular value in a particular register at a join point, both paths must put it there, or we need to generate additional code to move the value to the correct register
- With a simple 1-accumulator model of code generation, this should usually be true without needing extra work; with better use of registers it becomes a bigger issue
  - With more registers, would need to be sure they are used consistently at join point regardless of how we get there

Loops: while(exp) body

- Assuming we want the test at the bottom of the generated loop…
  - gen(jmp testLabel)
  - gen(bodyLabel:)
  - visit body
  - gen(testLabel:)
  - visit exp (condition) with target=bodyLabel and sense="jump if true"

Boolean Operators

&& (and || if you add it)

- Create label(s) needed to skip around the two parts of the expression
- Generate subexpressions with appropriate target labels and conditions

!exp

- Generate exp with same target label, but reverse the sense of the condition

```
public void visit(Not n) {
    String e = getExpr(n.e);
    System.out.println("cmpq $0, " + e);
    System.out.println("sete %al");
    System.out.println("movzbq %al %rax");
}
```

Boolean Expressions: exp1 < exp2

- Similar to other binary operators
- Difference: context is a target label and whether to jump if true or false
- Code

  visit exp1

  gen(pushq %rax)

  visit exp2

  gen(popq %rdx)

  gen(cmpq %rdx,%rax)

  gen(condjump targetLabel)

  - appropriate conditional jump depending on sense of test

Running MiniJava Programs

- To run a MiniJava program
  - Space needs to be allocated for a stack and a heap
  - %rsp and other registers need to have sensible initial values
  - We need some way to allocate storage (new) and communicate with the outside world

Bootstraping from C

- Idea: take advantage of the existing C runtime library
- Use a small C main program to call the MiniJava main method as if it were a C function
- C's standard library provides the execution environment and we can call C functions from compiled code for I/O, malloc, etc.

Assembler File Format

- Compiler output is an assembly language program (ascii)
- GNU syntax is roughly this (src/runtime/demo.s in project starter code is a runnable example, although not generated by a MiniJava compiler)

```
 .text                      # code segment
 .globl asm_main            # label at start of compiled static main
 <generated code>           # repeat .code/.data as needed
asm_main:                   # start of compiled "main"
        ...
    .data
    <generated method tables>
    # repeat .text/.data as needed
    …
    end
```

Main Program Label

- Compiler needs special handling for the static main method label
  - Label must be the same as the one declared extern in the C bootstrap program
  - asm_main used above
    - Could be changed, but probably no point
    - Why not "main"? (`main` is in boot.c)
- Strategy: Declare .text and .globl and .text in Program

```
public void visit(Program n) {
    System.out.println(".text");
    System.out.println(".globl " + getLabel("", "asm_main"));
    n.m.accept(this);
    if ( n.cl != null ) {
        for (int i = 0; i < n.cl.size(); i++) {
            n.cl.get(i).accept(this);
        }
    }
}
```

Main Method

- Create the asm_main method
- Set up the stack / frame pointers
- Process the single statement
- Leave

```
// Identifier i1,i2;
// Statement s;
public void visit(MainClass n) {
	n.i1.accept(this);
	st = st.findScope(n.i1.toString());
	n.i2.accept(this);
	st = st.findScope("main");
	System.out.println("");
	System.out.println( getLabel("", "asm_main") + ":" );
	System.out.println("pushq %rbp");
	System.out.println("movq "%rsp, rbp");
	n.s.accept(this);
	System.out.println("leave");
	System.out.println("ret");
	st = st.exitScope();
	st = st.exitScope();
}
```

Interfacing to “Library” code

- Trivial to call “library” functions
- Evaluate parameters using the regular calling conventions
- Generate a call instruction using the “library” function label
  - (External names need leading _ in Windows, OS X)
  - Linker will hook everything up

Bootstrap Program

- The bootstrap is a tiny C program that calls your compiled code as if it were an ordinary C function
- It also contains some functions that compiled code can call as needed
  - Mini "runtime library"
  - Add to this if you like
    - Sometimes simpler to generate a call to a new library routine instead of generating in-line code
    - Suggestion: do this for "exit if subscript out of bounds" check
- File: src/runtime/boot.c in project starter code

Bootstrap Program Sketch

```
#include <stdio.h>
extern void asm_main();  /* compiled code */
/* execute compiled program */
void main( ) { asm_main(); }
/* write x to standard output */
void put(int64_t x) { … }
/* return a pointer to a block of memory at least nBytes large (or null if insufficient
   memory available) */
char* mjcalloc(size_t nBytes) { return calloc(1,nBytes); }
```

Compiling & Testing the Program

- Run the Java program and get output:
  - javac BinarySearch.java
  - java BinarySearch > output1.txt
- Compile the program:
  - java -cp build/classes:lib/java-cup-11b.jar MiniJava -C BinarySearch.java > BinarySearch.s
- Compile the assembly program and boot.c with gcc:
  - gcc -g -o BinarySearch BinarySearch.s src/runtime/boot.c
- View the binary output in assembly
  - objdump -S BinarySearch
- Run your program and compare output:
  - ./BinarySearch > output2.txt
  - diff output1.txt output2.txt
- Debug your program with GDB:
  - gdb BinarySearch
  - > run
  - > bt

Next…

- Optimizations
- Assignment 5: Code Translation due next Wed
- Final Assignment will be to add support for division