Generazione di Codice

Slides based on material
by Ras Bodik available at
http://inst.eecs.berkeley.edu/~cs164/fa04

Status

- We have covered the front-end phases
  - Lexical analysis
  - Parsing
  - Semantic analysis
- Next are the back-end phases
  - Optimization
  - Code generation

- We'll discuss only code generation

Run-time environments

- Before discussing code generation, we need to understand what we are trying to generate

- There are a number of standard techniques for structuring executable code that are widely used

Outline

- Management of run-time resources
  (in particular, memory management)

- Correspondence between static (compile-time) and dynamic (run-time) structures

- Storage organization

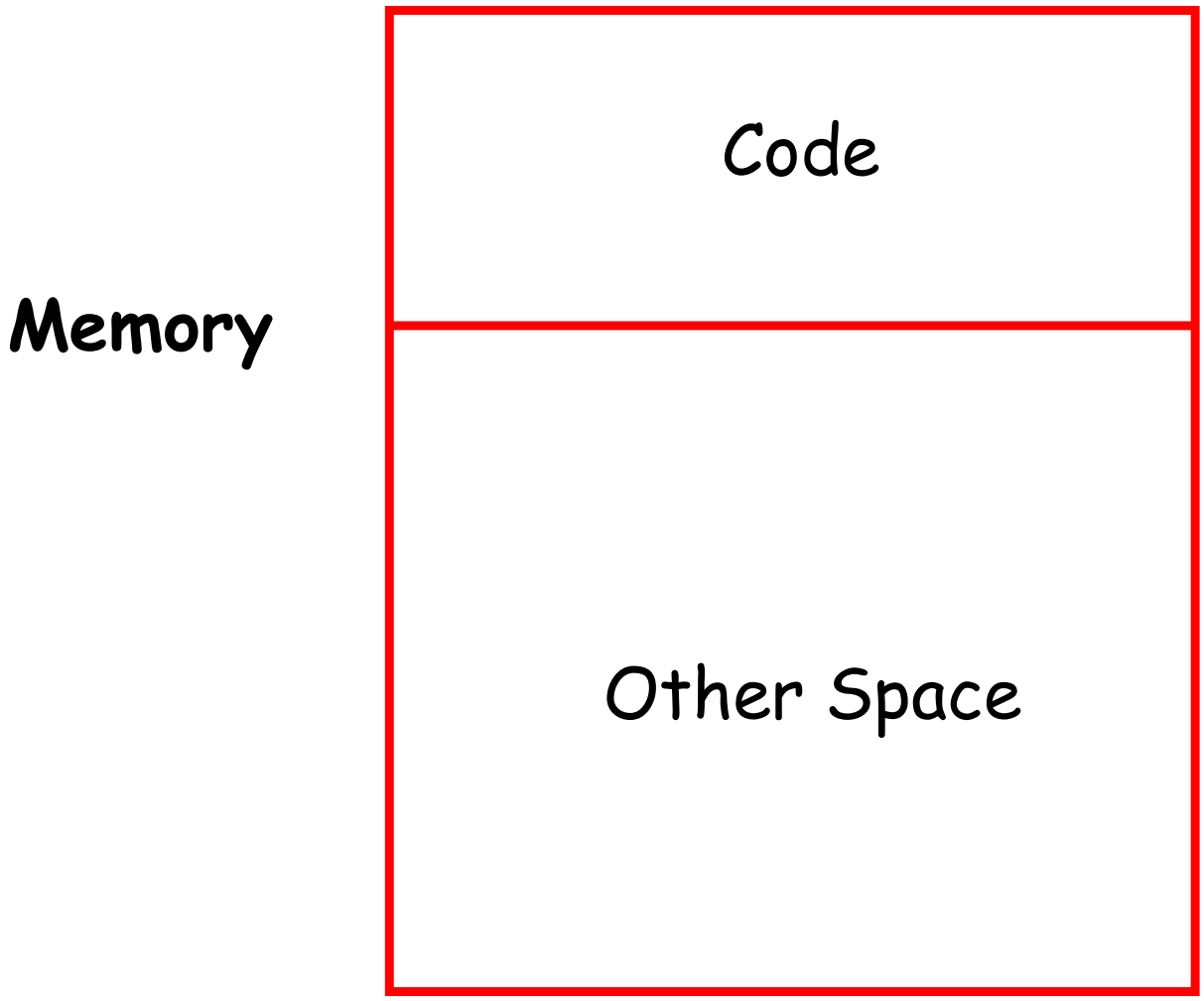
Memory Management

Run-time Resources

- Execution of a program is initially under the control of the operating system

- When a program is invoked:
  - The OS allocates space for the program
  - The code is loaded into part of the space
  - The OS jumps to the entry point (i.e., “main”)

Memory Layout



Notes

- By tradition, pictures of machine organization have:
  - Lines delimiting areas for different kinds of data
- These pictures are simplifications
  - E.g., not all memory needs be contiguous

What is Other Space?

- Holds all data for the program
- Other Space = Data Space

- Compiler is responsible for:
  - Generating code
  - Organizing use of the data area

Code Generation Goals

- Two goals:
  - Correctness
  - Speed

- Most complications in code generation come from trying to be fast as well as correct

Assumptions about Execution

1. Execution is sequential; control moves from one point in a program to another in a well-defined order (e.g. a single thread)

2. When a procedure is called, control eventually returns to the point immediately after the call

Activations

- An invocation of procedure P is an *activation* of P

- The *lifetime* of an activation of P is
  - All the steps to execute P
  - Including all the steps in procedures P calls

Lifetimes of Variables

- The *lifetime* of a variable x is the portion of execution in which x is defined

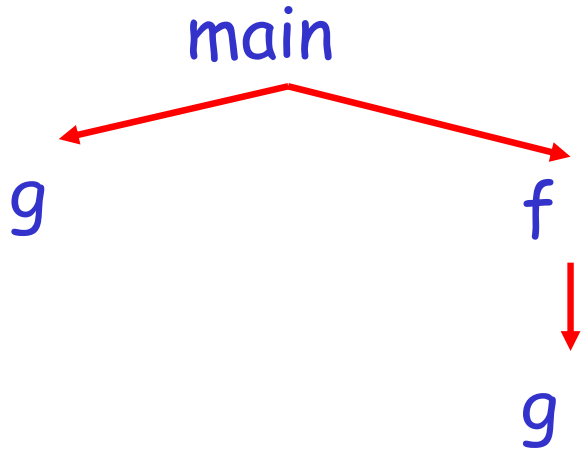
- Note that
  - Lifetime is a dynamic (run-time) concept
  - Scope is a static concept

Activation Trees

- Assumption (2) requires that when P calls Q, then Q returns before P does
- Lifetimes of procedure activations are properly nested
- Activation lifetimes can be depicted as a tree

Example

```
class Main {
    int g() { return 1; }
    int f() {return g(); }
    void main() { g(); f(); }
}
```



Example 2

```
class Main {
    int g() { return 1; }
    int f(int x) {
        if (x == 0) { return g(); }
        else { return f(x - 1); }
    }
    void main() { f(3); }
}
```

What is the activation tree for this example?

Notes

- The activation tree depends on run-time behavior
- The activation tree may be different for every program input
- Since activations are properly nested, a **stack** can track **currently active** procedures

Example

```
class Main {
   int g() { return 1; }
   int f() { return g(); }
   void main() { g(); f(); }
}
```

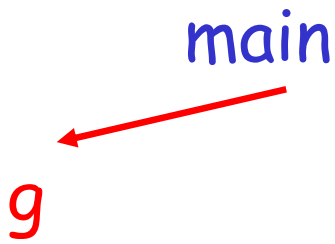
main

Stack

main

Example

```
class Main {
    int g() { return 1; }
    int f() { return g(); }
    void main() { g(); f(); }
}
```



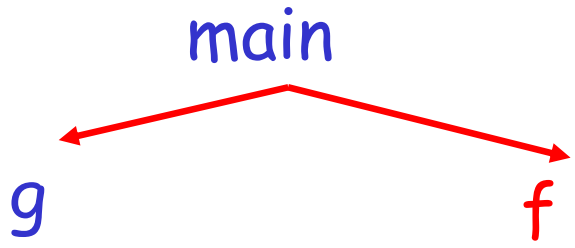
Stack

main

g

Example

```
class Main {
    int g() { return 1; }
    int f() { return g(); }
    void main() { g(); f(); }
}
```



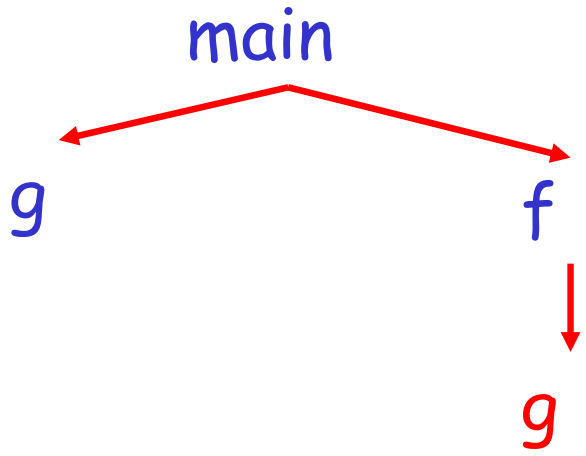
Stack

main

f

Example

```
class Main {
    int g() { return 1; }
    int f() { return g(); }
    void main() { g(); f(); }
}
```



Stack

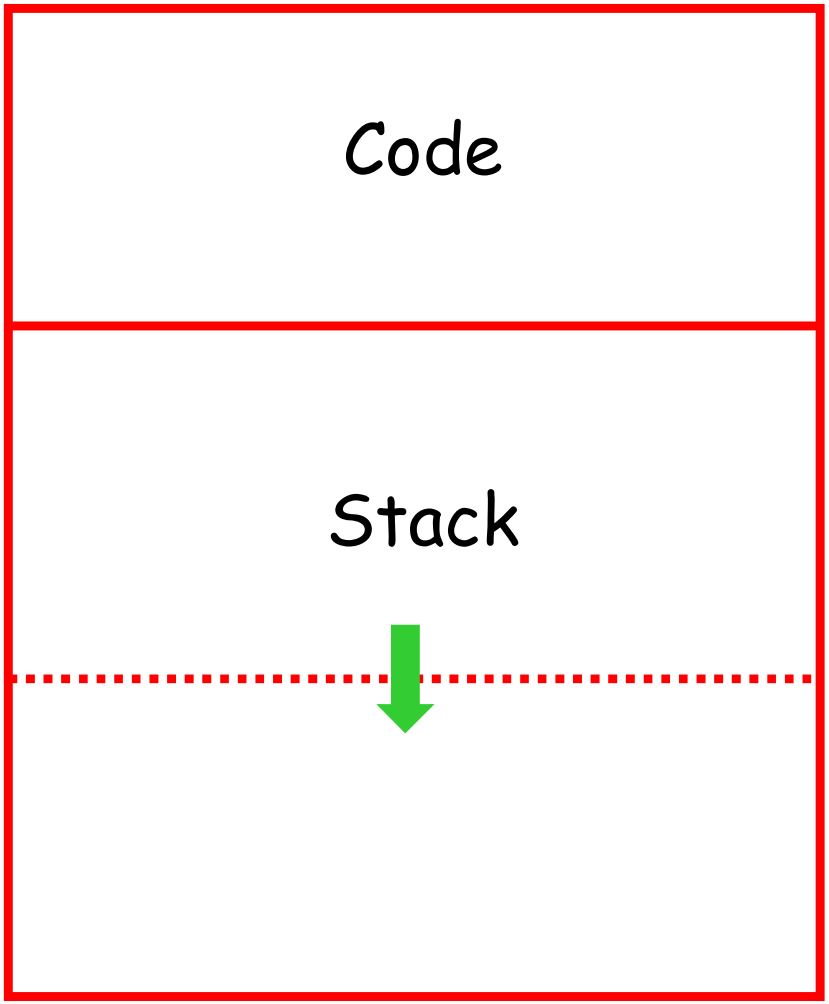
main

f

g

Revised Memory Layout

Memory



Activation Records

- The information needed to *manage one procedure activation* is called an ***activation record (AR)*** or ***frame***

- If procedure f calls g, then **g's activation record** contains a mix of info about f and g.

What is in g's AR when f calls g?

- f is "suspended" until g completes, at which point f resumes. g's AR contains information needed to **resume execution of f**.

- g's AR may also contain:
  - g's return value (needed by f)
  - Actual parameters to g (supplied by f)
  - Space for g's local variables

The Contents of a Typical AR for *g*

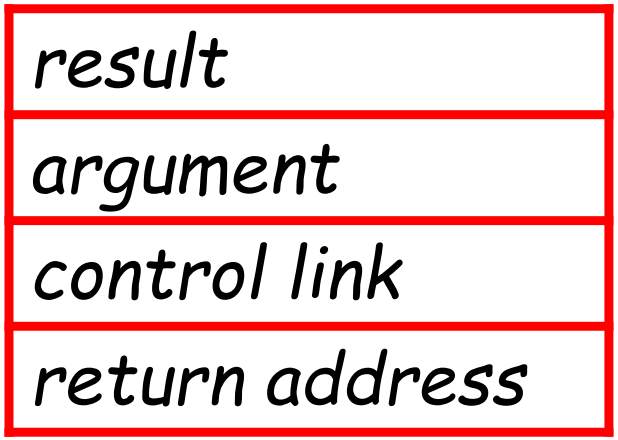
- Pointer to the previous activation record
  - The ***control link***; points to AR of caller of *g*
- Machine status prior to calling *g*
  - Contents of registers & **program counter**

- Space for *g*'s **return value**
- Actual **parameters**
- Local variables
- Other temporary values

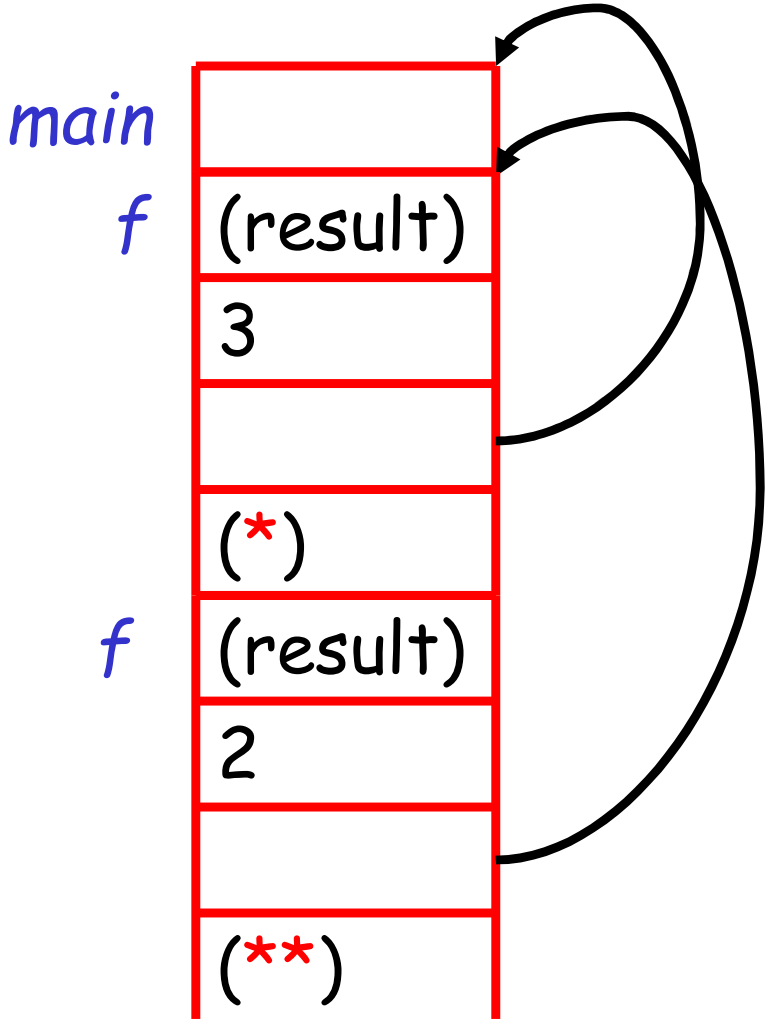
Example 2, Revisited

```
class Main {
    int g() { return 1; }
    int f(int x) {
        if (x == 0) { return g(); }
        else { return f(x - 1); (**) }
    }
    void main() { f(3); (*) }
}
```

AR for f:



Stack After Two Calls to *f*



Notes

- main has no argument, no return address and its result is never used; its AR is uninteresting
- (*) and (**) are return addresses of the invocations of f
  - The return address is where execution resumes after a procedure call finishes

- This is only one of many possible AR designs
  - Would also work for C, Pascal, FORTRAN (from 90, not 77 that disallows recursion), etc.

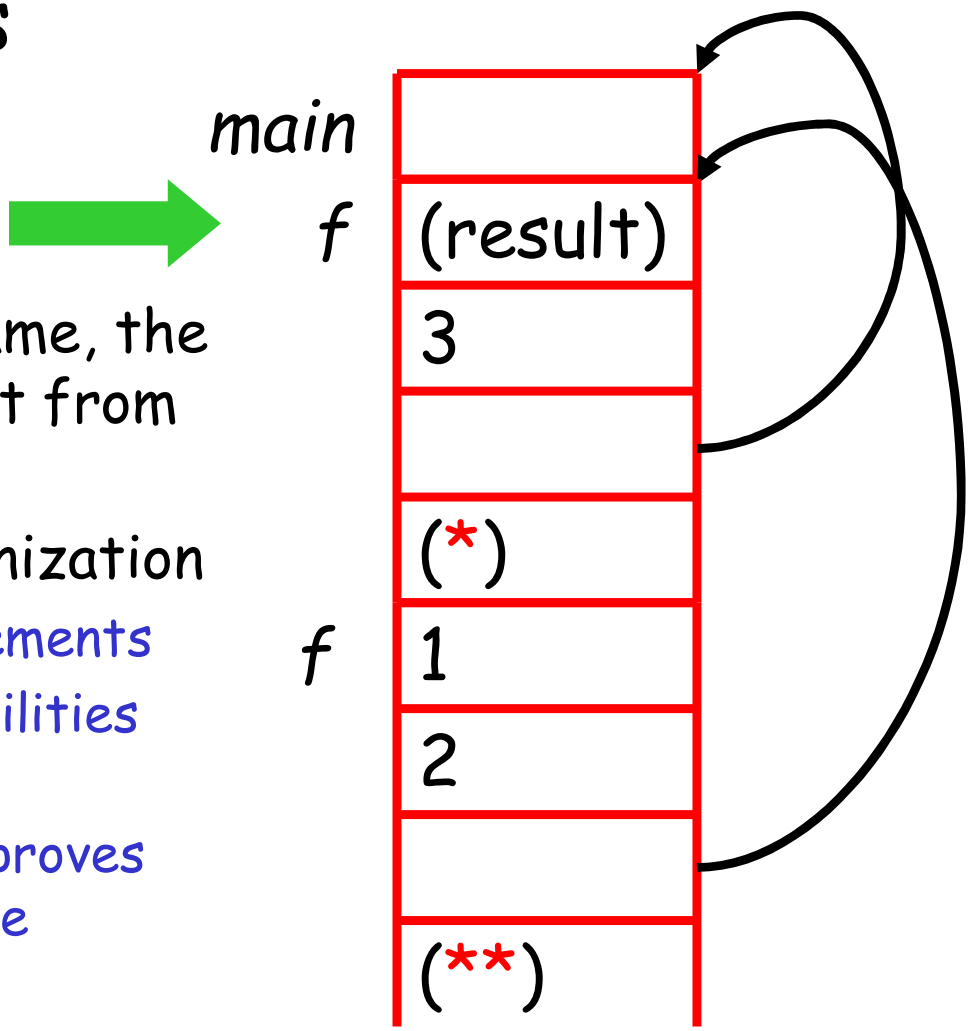
The Main Point

The compiler must determine, at compile-time, the **layout** of activation records and generate code that correctly accesses locations in the activation record

Thus, the AR layout and the code generator must be designed together!

Example

The picture shows the state when the call to 2nd invocation of f returns



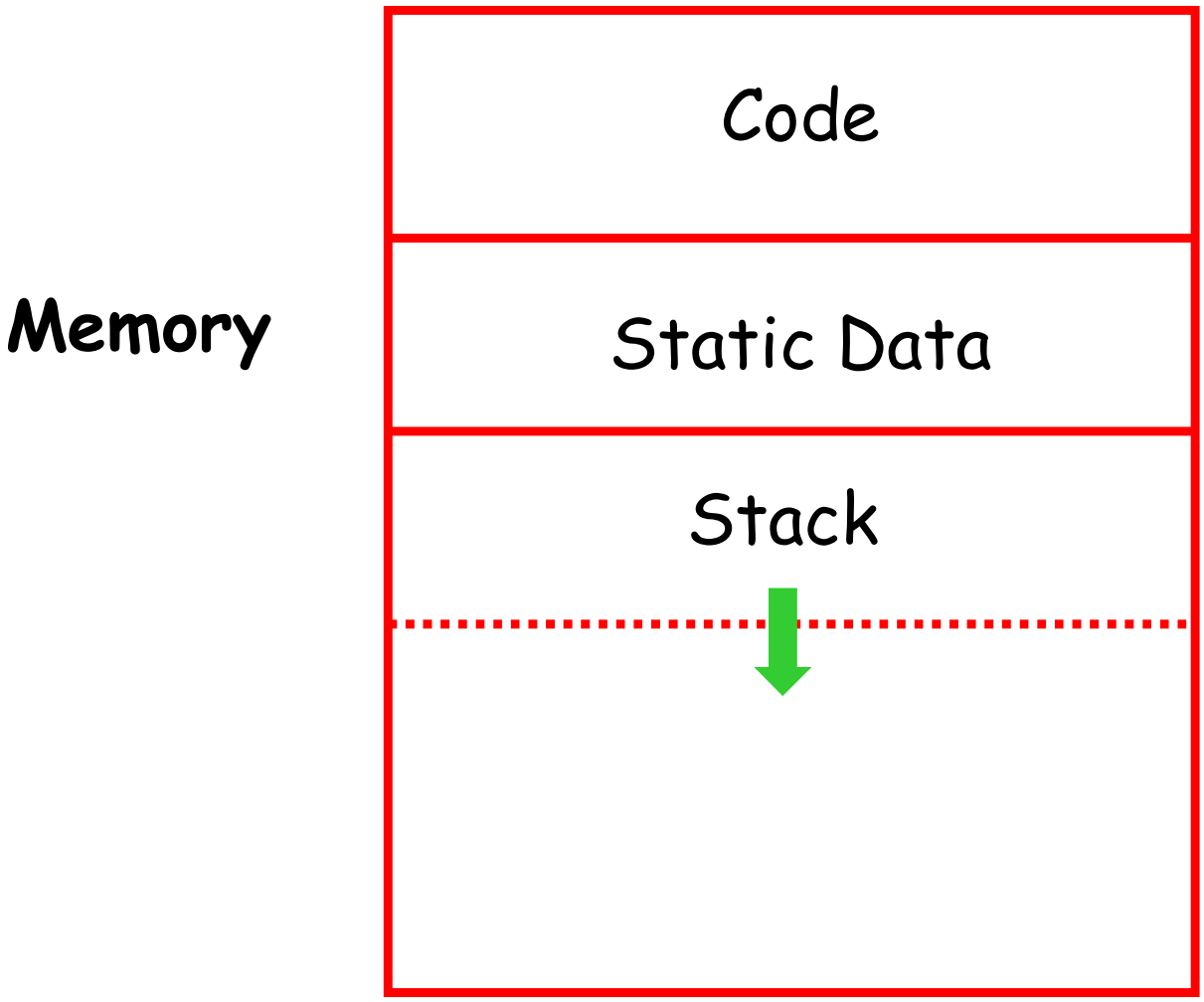
- As the return value is 1st in a frame, the caller can find it at a fixed offset from **its own frame**
- This is not the only possible organization
  - Can rearrange order of frame elements
  - Can divide caller/callee responsibilities differently
  - An organization is better if it improves execution speed or simplifies code generation

Global variables

- All references to a global variable point to the same element
  - Can't store a global in an activation record

- Globals are **assigned a fixed address** once
  - Variables with fixed address are "**statically allocated**"
- Depending on the language, there may be other statically allocated values

Memory Layout with Static Data

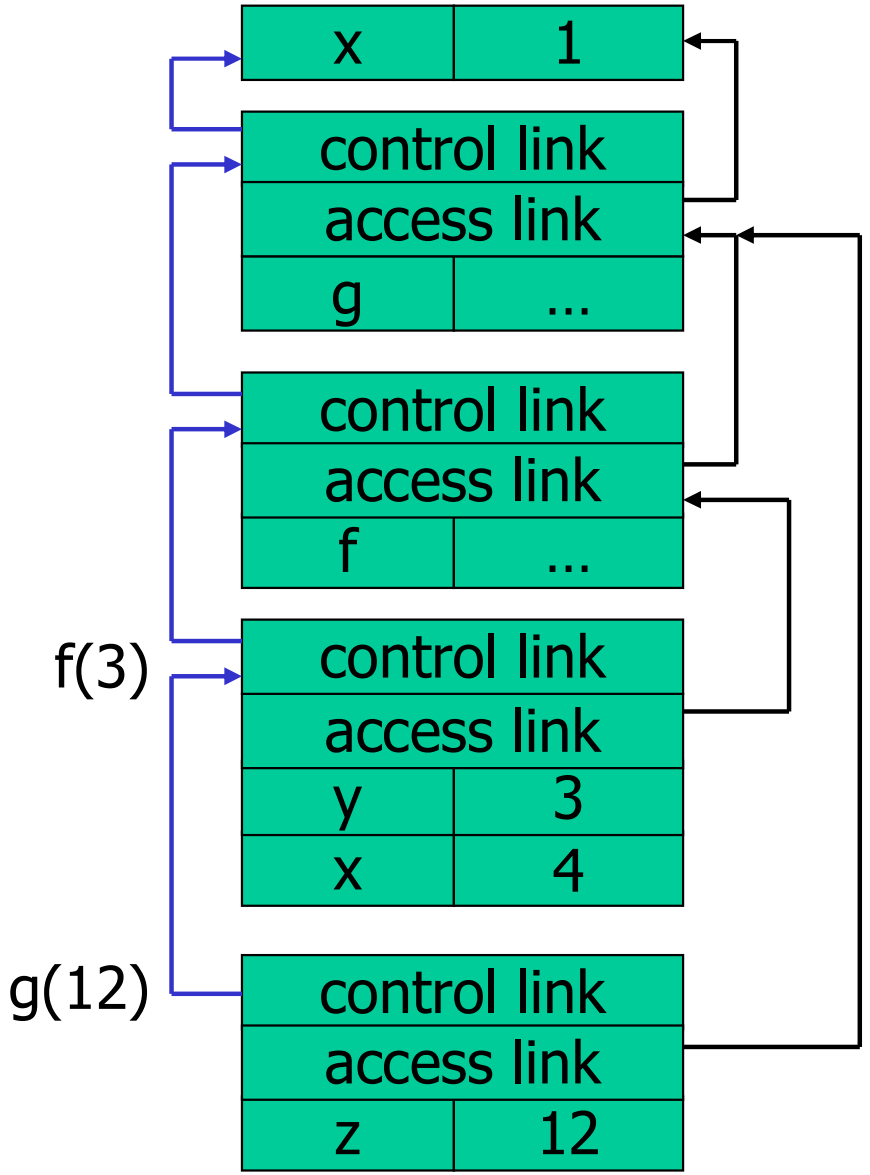


Variables declared in outer scopes

- References to a variable declared in an outer scope
  - Should point to a variable stored in another activation record
  - Also the execution of a block {…} has its own activation record
- To which activation record?
  - According to the "most closely nested" rule, an activation record should **point to the most recent activation record of its immediately enclosing scope**

Static scope with access links

```
int x = 1;
  { function g(z)={ return x+z };
    { function f(y)=
        { int x = y+1;
          return g(y*x) };
      f(3);
```



Use access link to find variables defined in outer scopes:

- Access link is always set to frame of enclosing syntactical block
- For function body, this is block that contains function declaration

Setting the access link

- **Value** of the ACCESS LINK of a new activation record **to be created** is established as follows:
  - An inner **block is entered** or a **function declared in the current scope** is called:
    - ACCESS LINK = current AR
  - A **function** calles itself **recursively** or calls another function **declared in the enclosing syntactical block**:
    - ACCESS LINK = **value** of ACCESS LINK of the current AR
  - In general, call to a function **outside** the current scope:
    - ACCESS LINK = follow the chain of ACCESS LINKs for the **difference** between **current nesting level** and **that of function declaration** (such a difference is 1 in previous case)

Setting the access link

- **Value** of the ACCESS LINK of a new activation record **to be created** is established as follows:
  - An inner **block is entered** or a **function declared in the current scope** is called:
    - ACCESS LINK = (**address** of ACCESS LINK in) current AR
  - A **function** calles itself **recursively** or calls another function **declared in the enclosing syntactical block**:
    - ACCESS LINK = **value** of ACCESS LINK of the current AR
  - In general, call to a function **outside** the current scope:
    - ACCESS LINK = follow the chain of ACCESS LINKs for the **difference** between **current nesting level** and **that of function declaration** (such a difference is 1 in previous case)

Heap Storage

- A value that outlives the procedure that creates it cannot be kept in the AR

- Bar foo() { return new Bar }
  The Bar value must survive deallocation of foo's AR

- Languages with dynamically allocated data use a *heap* to store dynamic data

Garbage Collection

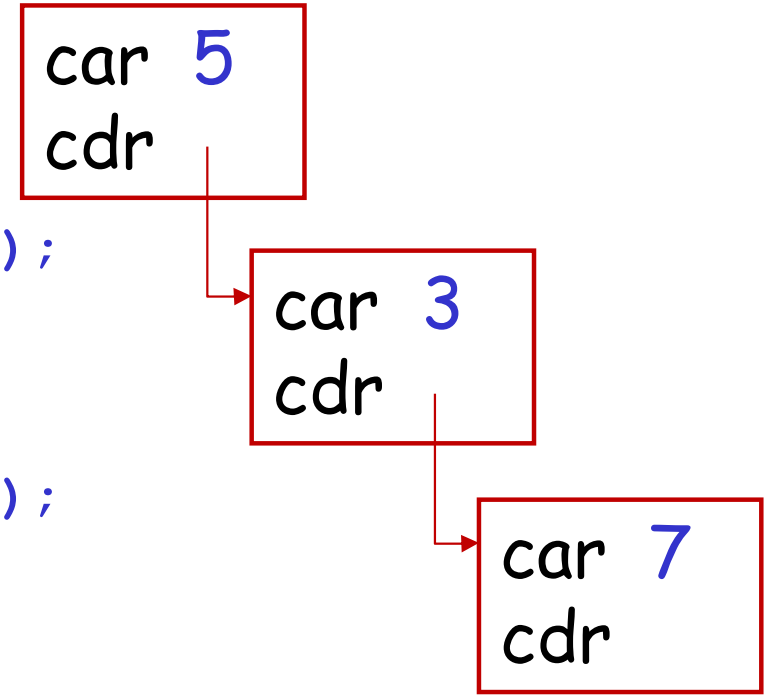
- The ARs in the stack are deallocated when the control exits from the corresponding scope
- What about data in the Heap?
  - They can be removed when they become "Garbage":
    - At a given point in the execution of a program P, a memory location m is *garbage* if **no continued execution of** P from this point can access location m
- Garbage Collection:
  - Detect garbage during program execution
  - GC invoked when more memory is needed
  - Decision made by run-time system, not program

Why Garbage Collection?

- In some languages, deallocation is under the responsibility of the programmer
- Example of deallocation in C:

```
for(ptr=lst; ptr!=NULL; ){
   if (ptr->car == x) return(ptr->cdr);
   else ptr = ptr->cdr;}

for(ptr=lst; ptr!=NULL; ){
   if (ptr->car == x) return(ptr->cdr);
   else{previous = ptr;
        ptr = ptr->cdr;
        free(previous);}}
```



- Problem: in the case of **sharing** (more pointers to the same location) the cell cannot be actually freed !!

Mark-and-Sweep Algorithm

- Assume tag bits associated with data
- Need list of locations named by program
- Algorithm:
  - Set all tag bits to 0.
  - Start from each location used directly in the program. Follow all links, changing tag bit to 1
  - Consider as garbage all cells with tag = 0

Reference Counting algorithm

- Each datum in memory has an associated reference counter
  - When a datum is allocated in memory, initialize the counter to 0
  - When a pointer to a datum is set, increments the counter
  - When a pointer to a datum is reset, decrement the counter
  - When the counter turns to be 0, the datum is garbage

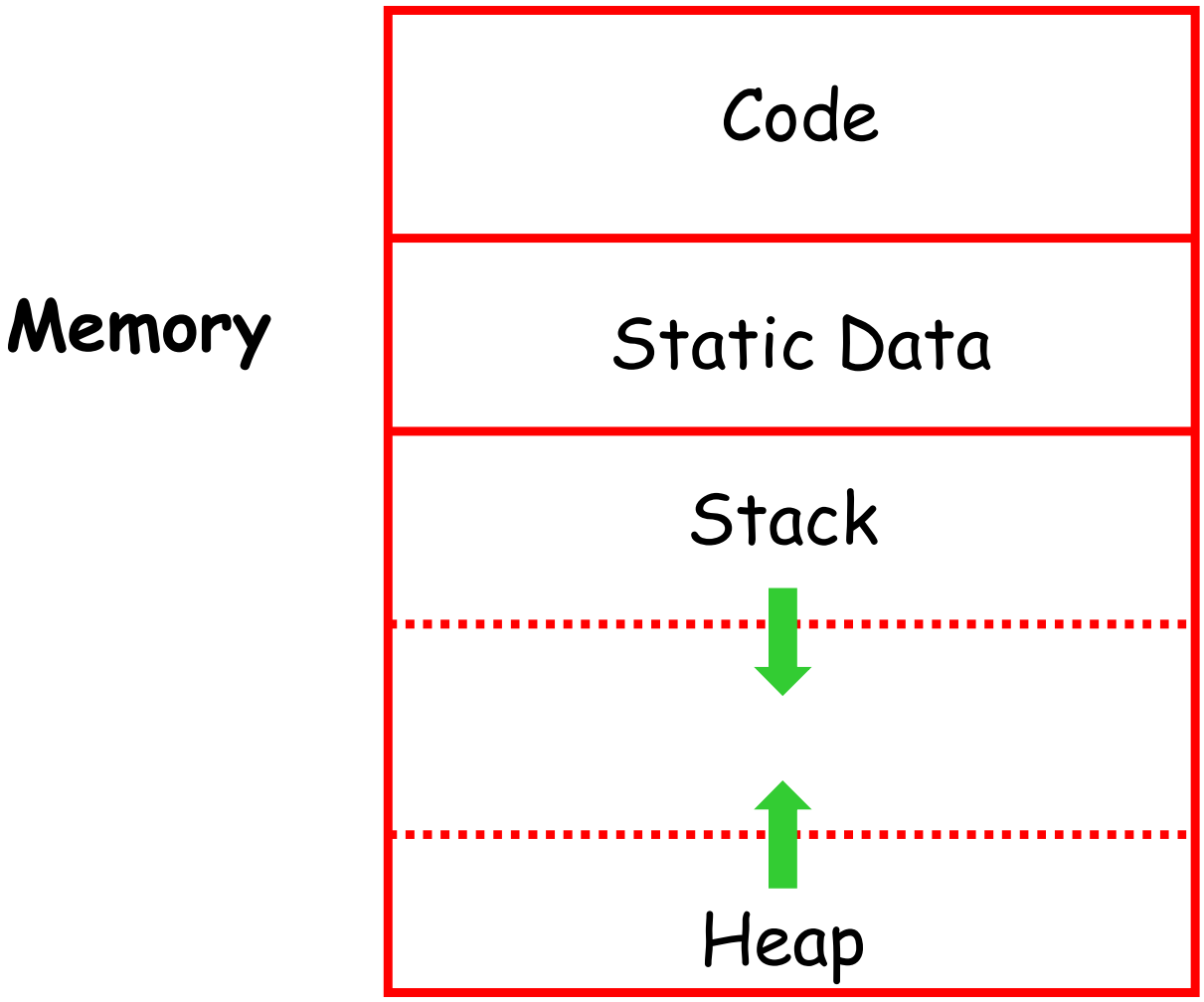
Notes

- The code area contains object code
  - For most languages, fixed size and read only
- The static area contains data (not code) with fixed addresses (e.g., global data)
  - Fixed size, may be readable or writable
- The stack contains an AR for each currently active procedure
  - Each AR usually fixed size (for a given procedure), contains locals
- Heap contains all other data

Notes (Cont.)

- Both the heap and the stack grow
- Must take care that they don't grow into each other
- Solution: start heap and stack at opposite ends of memory and let them grow towards each other

Memory Layout with Heap



Code Generation for Stack Machine

Lecture Outline

- Stack machines
- The MIPS assembly language
- A simple source language
- Stack-machine implementation of the simple language

Stack Machines

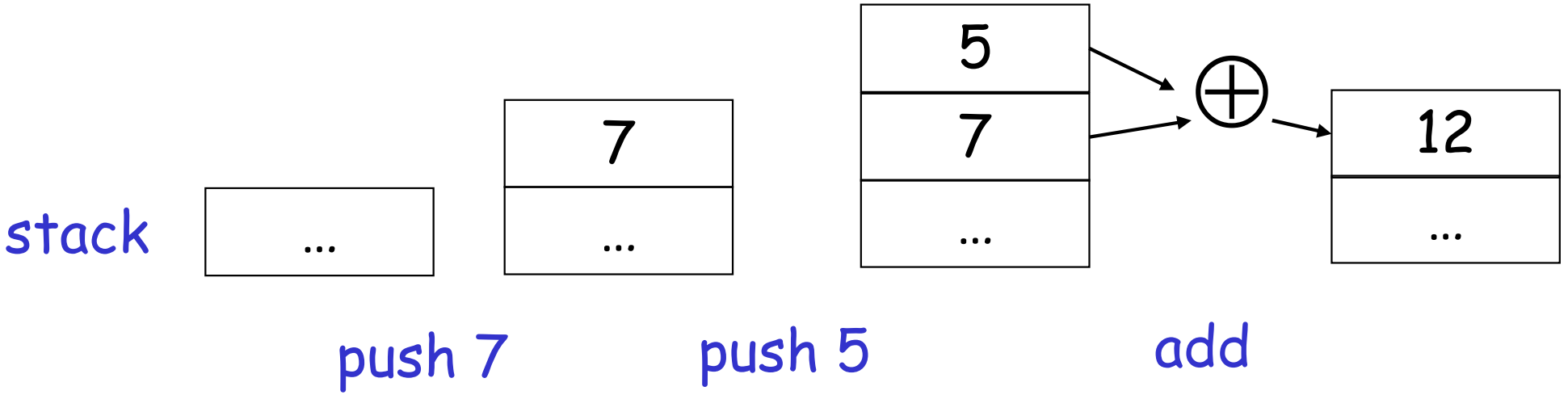
- A simple evaluation model
- No variables or registers
- A stack of values for intermediate results

Example of a 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 Using 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
- **Invariant**:
  - **After** computing an expression the **stack is as before** with **result** additionally **pushed on top**

Why Using 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_0, r_1, r_2" (three-address code)
  - $\Rightarrow$ Smaller encoding of instructions
  - $\Rightarrow$ More compact programs
- This is one reason why **Java Bytecodes** use a stack evaluation model

Computing Expressions with Stack Machine

Remember the **invariant**:

- **After** computing a (sub)expression the **stack is as before** with **result** additionally **pushed on top**

For an operation op(e_1,e_2):

- Compute e_1 (at the end the **stack is as before** with just **e_1 result** additionally **pushed on top**)
- Compute e_2 (at the end the **stack is as before** with just **e_2 result** additionally **pushed on top**)
- Compute op (**pops 2** values and **pushes op result**)

A Bigger Example: 13 - (7 + 5)

Code	Stack
	<init>
push 13	13, <init>
push 7	7, 13, <init>
push 5	5, 7, 13, <init>
add	12, 13, <init>
sub	1, <init>

Notes

- It is **very important** that the **stack is preserved** across the evaluation of a subexpression
  - Stack before the evaluation of 7 + 5 is: 13,<init>
  - Stack after the evaluation of 7 + 5 is: 12,13,<init>
  - After popping the result (12) of the subexpression (7+5) the stack **must be as before**

The Stack Machine can be optimized

- 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 **accumulator**)
  - Register accesses are **faster**
- The "add" instruction is now

  $$acc \leftarrow acc + top\_of\_stack$$

  - Only **one** memory operation!
- For simplicity we **will not use this optimization**

From Stack Machines to Assembly language

- We consider a compiler that generates code for a stack machine (**without accumulator**)
- We want to run the resulting code on some processor
- We consider the prototypical processor known as **MIPS** (there exist simulators for several real architectures)
  - Microprocessor without Interlocked Pipeline Stages
- More precisely:
  - We implement stack machine instructions using MIPS instructions and registers

MIPS Assembly

MIPS architecture

- Prototypical Reduced Instruction Set Computer (**RISC**) architecture
- Arithmetic operations use registers for operands and results
- Must use load and store instructions to use operands and results in memory
- 32 general purpose registers (32 bits each)
  - We will use $sp and $t0, $t1, $t2 (temporary registers)

Simulating a Stack Machine…

- The stack is kept in memory
- The stack grows towards lower addresses
  - (in previous presentation of "memory management" assume lower addresses to be at the bottom)
- Stack pointer is kept in MIPS register $sp
  - The top element of the stack is at address $sp

A Sample of MIPS Instructions

- lw reg_0 offset(reg_1)
  - Load 32-bit word from address reg_1 + offset into reg_0
- add reg_0 reg_1 reg_2
  - $reg_0 \leftarrow reg_1 + reg_2$
- sw reg_0 offset(reg_1)
  - Store 32-bit word in reg_0 at address reg_1 + offset
- addiu reg_0 reg_1 imm
  - $reg_0 \leftarrow reg_1$ + imm
- li reg imm
  - reg $\leftarrow$ imm

Some Useful Macros

- We define the following abbreviations
- push $t

```
addiu $sp $sp -4
sw $t 0($sp)
```

- pop

```
addiu $sp $sp  4
```

- $t ← pop

```
lw $t 0($sp)
addiu $sp $sp  4
```

- pop*n

```
addiu $sp $sp  z
```

with z = 4*n (pop n times)

MIPS Assembly. Example.

- The stack-machine code for 7 + 5 in MIPS:

push 7	li $t0 7 push $t0
push 5	li $t0 5 push $t0
add	$t2 ← pop $t1 ← pop add $t0 $t1 $t2 push $t0

A Small Language

- A language with integers, integer operations, and global functions with at least one parameter

$$P \rightarrow D\ ;\ P \mid D$$

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

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

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

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:

```
fun 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 MIPS code** that:
  - **Computes the result of e** and pushes it on the stack
  - **Preserves the contents** that were on the stack **before** evaluating e
    - the only difference is that, after evaluating e, the stack additionally contains the **result of e on top**

- 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 (expression "i", e.g. "5") simply pushes it on the stack:

```
cgen(i) = li $t0 i
          push $t0
```

- Note that this also preserves the stack, as required

Code Generation for Add

cgen(e_1 + e_2) =

```
cgen(e1)
cgen(e2)
$t2 ← pop
$t1 ← pop
add $t0 $t1 $t2
push $t0
```

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

''Optimized'' Code for Add. Wrong!!!

- Optimization: Put the result of e_1 immediately in $t1?

```
cgen(e1 + e2) =
        cgen(e1)
        $t1 ← pop
        cgen(e2)
        $t2 ← pop
        add $t0 $t1 $t2
        push $t0
```

- Try to generate code for : 13 + (7 + 5)
- Never expect the **value of a register** to be preserved **across generated code**!

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** is performed **bottom-up** by a recursive traversal of the **AST**

Code Generation for Sub

- New instruction: sub reg_0 reg_1 reg_2
  - Implements $reg_0 \leftarrow reg_1 - reg_2$

```
cgen(e1 - e2) =
    cgen(e1)
    cgen(e2)
    $t2 ← pop
    $t1 ← pop
    sub $t0 $t1 $t2
    push $t0
```

Code Generation for Conditional

- We need flow control instructions
- New instruction: beq reg_1 reg_2 label
  - Branch to label if $reg_1 = reg_2$
- New instruction: b label
  - Unconditional jump to label

Code Generation for If (Cont.)

cgen(if e_1 = e_2 then e_3 else e_4) =

```
    cgen(e1)
    cgen(e2)
    $t2 ← pop
    $t1 ← pop
    beq $t1 $t2 true_branch
    cgen(e4)
    b end_if
true_branch:
    cgen(e3)
end_if:
```

Actually **label names** need to be **freshly** generated by cgen( )!

e.g. try to generate code in the case e_3 is again an if-then-else

The Activation Record

- Code for **function calls** and **function definitions** depends on the **layout** of the activation record
- A very simple AR suffices for this language:
  - The **result** of a **function call** is simply pushed on top of the **stack** at the end of its execution (as for any other expression)
    - No need to reserve a space for the result in the AR
  - The **activation record** holds **parameter values**
    - For $f(x_1,\ldots,x_n)$ the activation record holds $x_n,\ldots,x_1$ values
    - These are the only variables in this language

The Activation Record (Cont.)

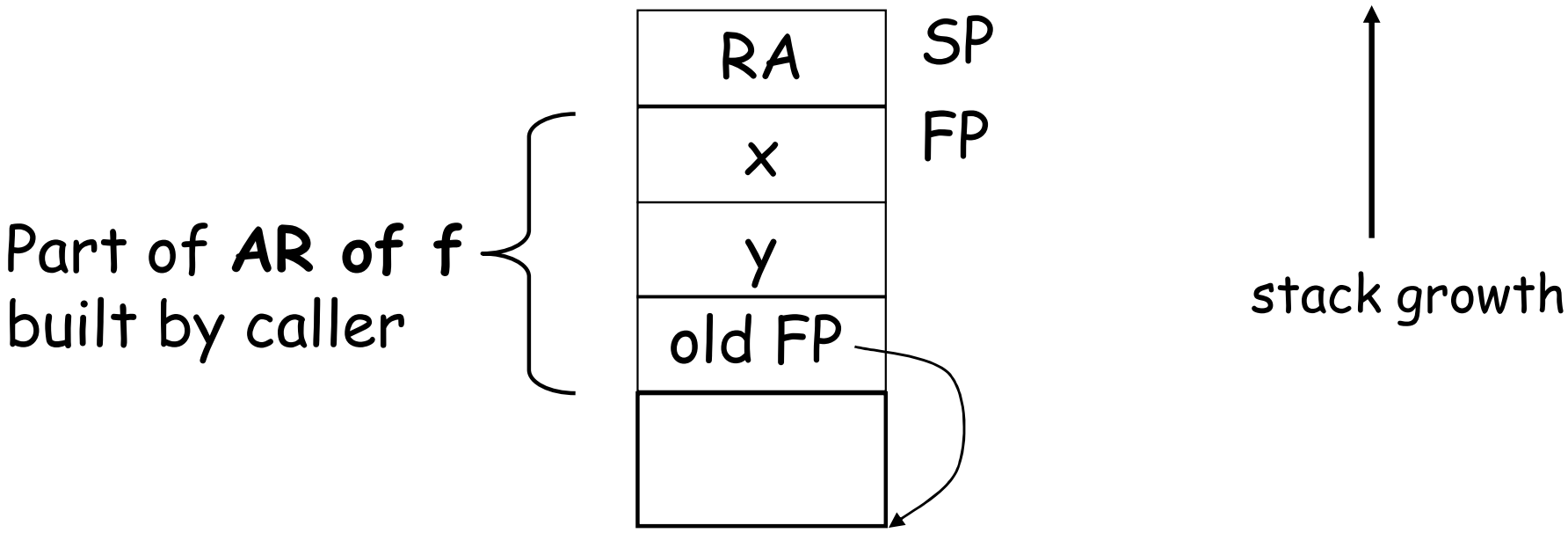
- We need to store the **return address**
- We also need the **Control Link** (next slide)
  - **Access Link** is not needed because we do not have nested declarations. We just have:
    - local parameter declarations and
    - functions that are all declared in the global scope that is allocated statically

The Activation Record (Cont.)

- It's handy to have a **pointer** to a **reference position** in the **current Activation Record**
  - This pointer lives in **register** $fp (frame pointer)
  - The reference position is **part of AR layout design**
    - Here we take $fp to point at **position of the first argument**
  - $fp used by generated code to **locate AR elements**, e.g. parameters, based on **offsets**
- We need to store $fp of the caller (**Control Link**) in the AR, so that we can restore it
  - in general **for linking to an AR** we use the **address of its reference position**

The Activation Record

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



Code Generation for Function Call

- The calling sequence is the instructions (of both caller and callee) to set up a function invocation
- New instruction: jal label
  - Jump to label, save address of next instruction in $ra
    - instruction name means: jump and link (call to subroutine)

Code Generation for Function Call (Cont.)

cgen($f(e_1,\ldots,e_n)$) =

```
push $fp
cgen(e_n)
...
cgen(e_1)
jal f_entry
```

- The caller saves its value of the frame pointer
- Then it saves the actual parameters in reverse order
- The caller saves the return address in register $ra
- The AR so far is 4*n+4 bytes long

Code Generation for Function Definition

- New instruction: jr reg
  - Jump to address in register reg

cgen(fun f(x_1,…,x_n) = e) =

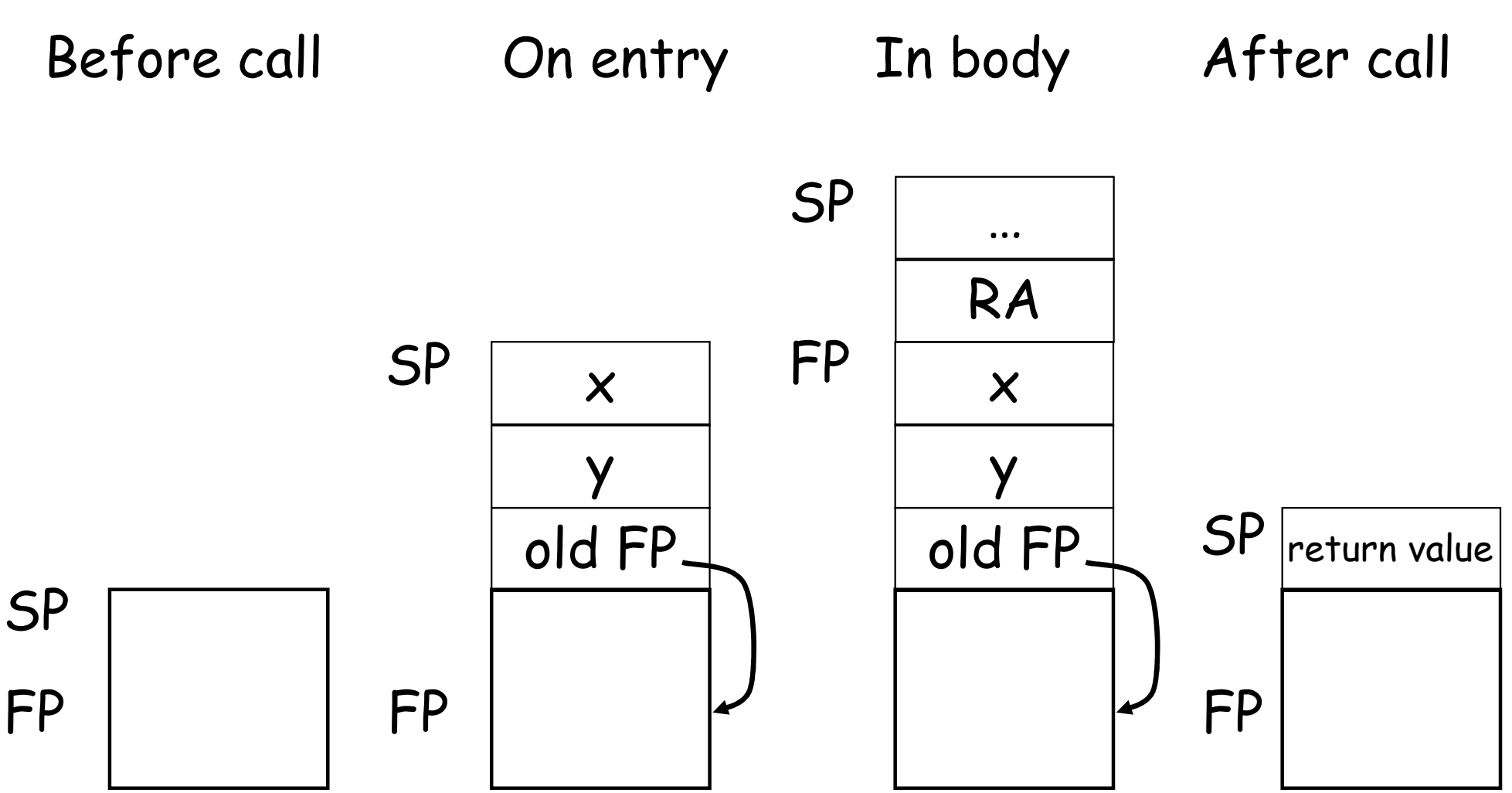
f_entry:

```
move $fp $sp
push $ra
cgen(e)
$t0 ← pop
$ra ← pop
pop*n
$fp ← pop
push $t0
jr $ra
```

- Content of $sp is copied into $fp

  Note: The frame pointer points to first argument (not to bottom of frame)
- $t0 is a register used to temporarily hold the return value
- The callee pops the return value and removes all parts of frame of f from the stack

Calling Sequence. Example for f(x,y).



Activation Records not adjacent on stack!

- Because the **stack grows when intermediate results are saved**, Activation Records are not adjacent on stack!
  - Example: fun f(x) = 3+g(5)
  - Upon execution 3 is on the stack between the AR of f and the AR of g

Code Generation for usage of Parameters

- Usages of Parameters/Variables are the last construct we consider
- In this language the "variables" of a function are just its parameters
  - They are in current AR (we don't have nested scopes)
  - Allocated by the caller (that pushes their value)

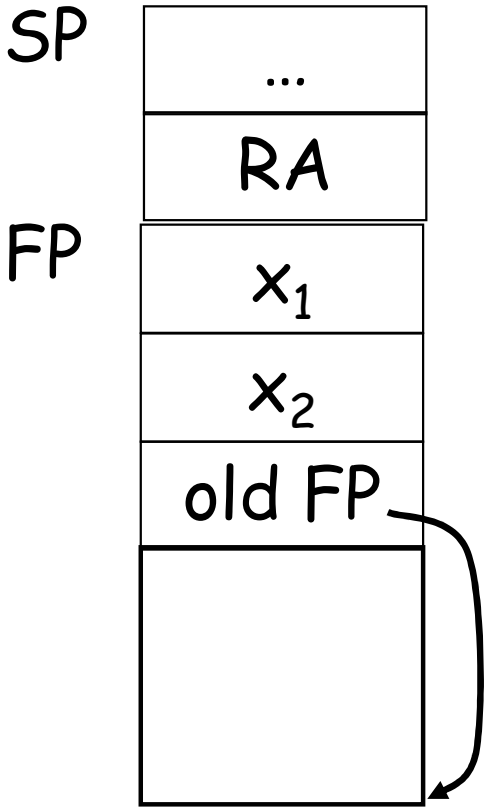
- Problem: because, again, the stack grows when intermediate results are saved, the variables are **not at a fixed offset** from $sp
  - Example: fun f(x) = 3+x

Code Generation for Variables (Cont.)

- Solution: use the **frame pointer** $fp
  - in our AR layout it always points to the first variable
- Let x_i be the i^{th} (i = 1,…,n) formal parameter of the function for which code is being generated

Code Generation for Variables (Cont.)

- Example: For a function fun f(x_1,x_2) = e the activation record and frame pointer are set up as follows:



x_1 is at $fp

x_2 is at $fp + 4

- Thus, based on **offset** of x_i:

cgen(x_i) = lw $t0 z($fp)
push $t0

with offset z = 4*(i-1)

- Info about the **offset** z of each parameter name **needs to be included** in its **symbol table entry**

Summary

- The **activation record layout** must be designed together with the code generator
- Code generation can be done by **recursive traversal** of the AST

- To simplify the presentation we have not discussed the Access Links!
  - But they can be easily added (as already discussed in "memory management"):
    - check the **difference in the nesting level** between the caller code and the declaration of the called function, and **follow** the already settled **access links** accordingly

Code Generation for Object-Oriented Languages

Two Issues

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

Object Layout Example

```
class A {
    int a = 0;
    int d = 1;
    int f() { return a = a + d }
}

class  B extends A {
    int b = 2;
    int f() { return a } //override
    int g() { return a = a - b }
}

class  C extends A {
    int c = 3;
    int h() { return a = a * c }
}
```

Object Layout (Cont.)

- Fields 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, field a must be in the same "place" in all objects

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

- Object is **laid out in a contiguous memory section**
- Each field stored at a fixed **offset** in object
  - offset needs to be **inserted in field symbol table entry**
- On object creation, the corresponding **layout is instantiated in the heap** (it will be eventually removed from the garbage collector)

Subclasses

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 **fields with new names** defined in B

Leaves the **layout of A "part" unchanged**

- layout of B is an extension of it

Therefore: objects of class B **preserve offsets of fields** declared by A

Dynamic Dispatch

- Consider again our example

```
class A {
    int a = 0;
    int d = 1;
    int f() { return a = a + d }
}

class B extends A {
    int b = 2;
    int f() { return a }
    int g() { return a = a - b }
}

class C extends A {
    int c = 3;
    int h() { return a = a * c }
}
```

Dynamic Dispatch Example

- e.g() with e having static type B
  - calls method g of B
- e.f() with e having static type A
  - calls method f of A if e yields an A or C object (f is inherited in the case of C)
  - calls method f of B if e yields a B object (f is overridden)
- The implementation of methods and dynamic dispatch strongly **resembles the implementation of fields**

Dispatch Tables

- Every **class** has a **fixed set of methods** (including inherited methods)

- *A **dispatch table*** indexes these methods
  - An array of **method addresses**
  - A method f lives at a **fixed offset** in the dispatch table for a class **and all of its subclasses**

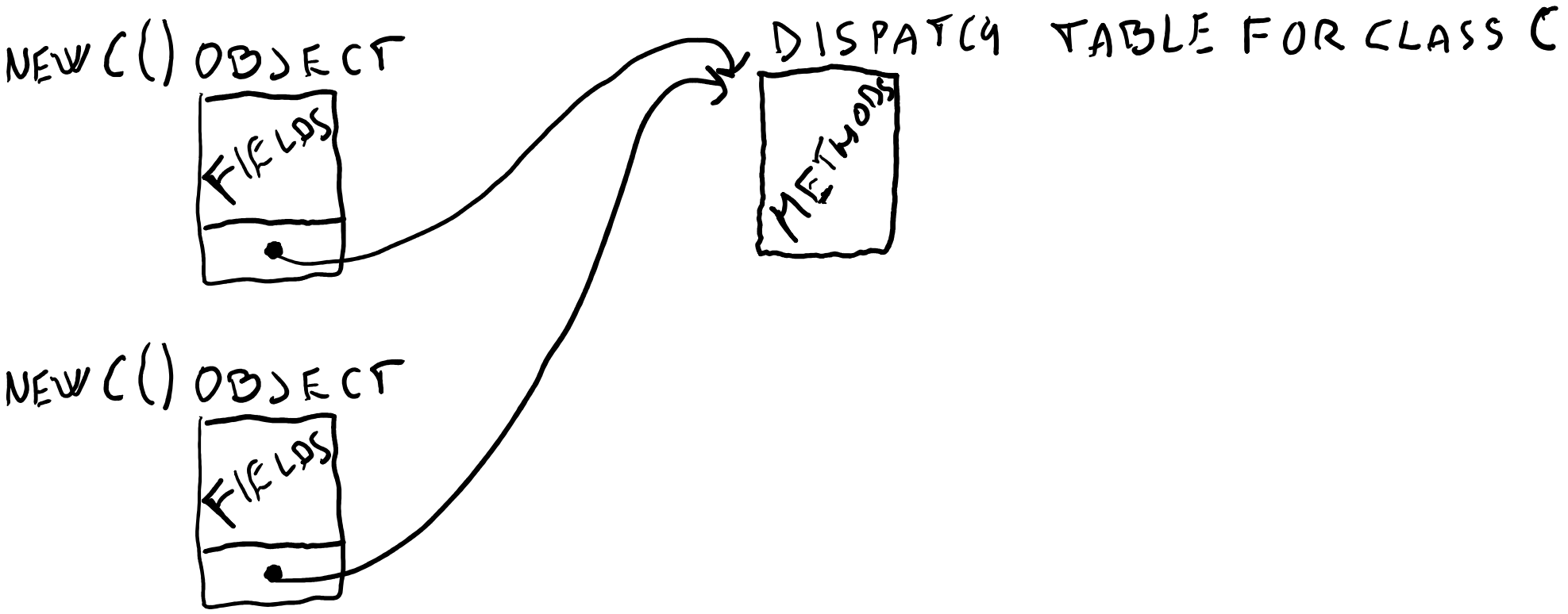
Dispatch Table Example

Class \ Offset	0	4
A	f of A	
B	f of B	g
C	f of A	h

- The dispatch table for class A has only **1 method**
- The tables for B and C **extend** the table for A **to the right**
- Because methods can be overridden, the address for f is **not the same in every class**, but is always **at the same offset**

Using Dispatch Tables

- The **dispatch pointer** in an **object** of class *C* points to the **dispatch table for class** *C*



Using Dispatch Tables

- The **dispatch pointer** in an **object** of class *C* points to the **dispatch table for class** *C*

- Every method *f* of class *C* is assigned an **offset** O_f in the dispatch table at **compile time**
  - The offset is **inserted in the symbol table entry** of method *f* of class *C* as usual

Using Dispatch Tables (Cont.)

- To implement a dynamic dispatch e.f() we
  - Let O_f be the **offset** of the method f in the dispatch-table associated to the **static type** of e
  - Evaluate e, obtaining **an object** o (that could be of any subclass)
  - Let D be the **dispatch-table** of o
  - Execute the method pointed by $D[O_f]$