Run-time Environments

Lecture 11

Status

- We have covered the front-end phases
  - Lexical analysis
  - Parsing
  - Semantic analysis

  ] enforce language definition
- No more looking for errors
- No longer trying to determine whether it is a valid program
- But, we will still use data structures generated by front end

Status

- Next are the back-end phases
  - Optimization
  - Code generation
- We'll do code generation first . . .

Run-time environments

- Before discussing code generation, we need to understand what we are trying to generate
  - Before we can actually try to generate it!
- First we'll talk about what the translated program looks like and how it's organized
- Then we'll talk about the code generation algorithms that are actually producing those things
- There are a number of standard techniques for structuring executable code that are widely used

Outline

- We'll discuss management of run-time resources

- Stressing the correspondence between
  - static (compile-time) and
  - dynamic (run-time) structures
  - This is an important piece of understanding how a compiler really works: what happens at compile time vs what is deferred to runtime

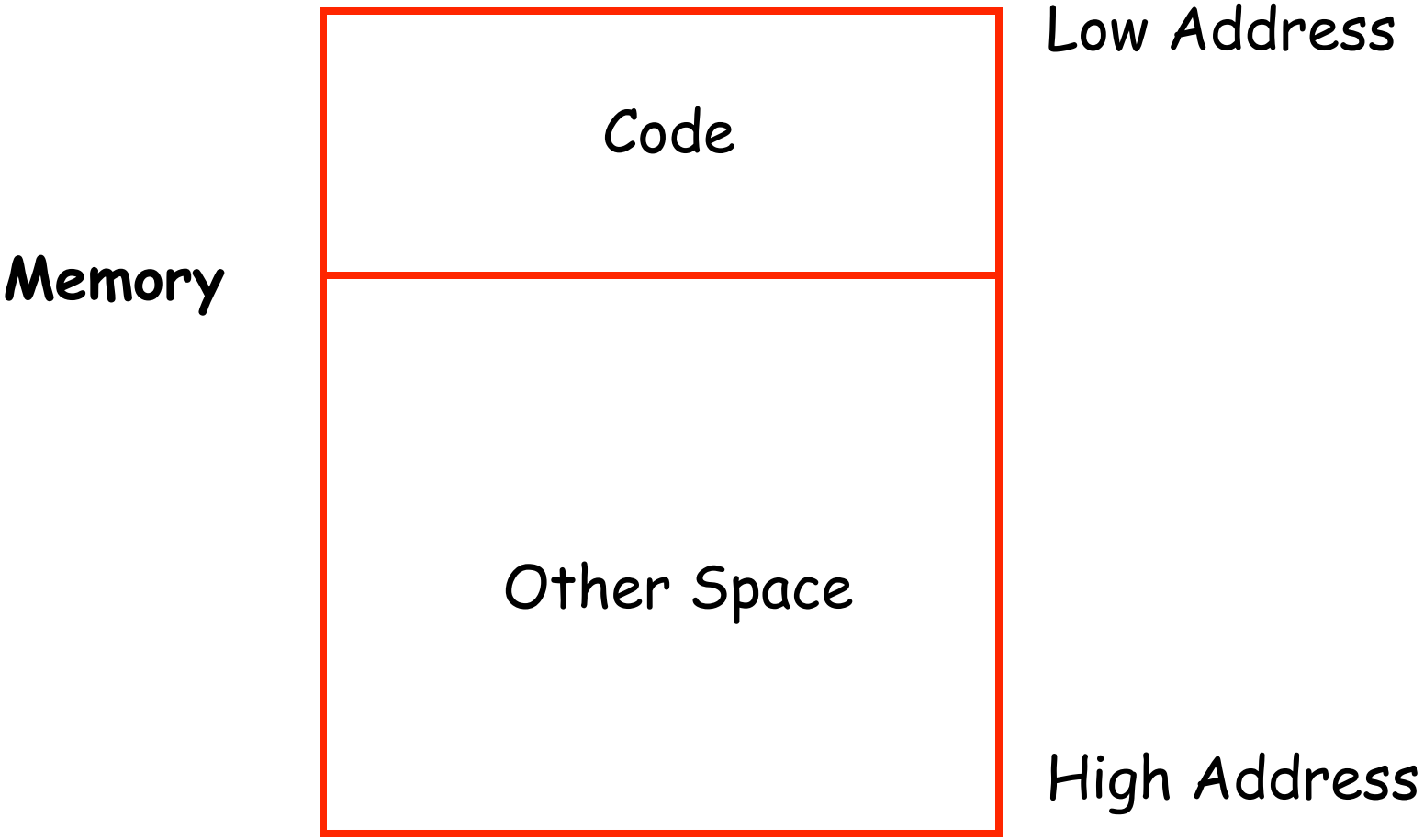
- Storage organization

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 (Roughly)



Entirety of memory that is allocated to program

Notes

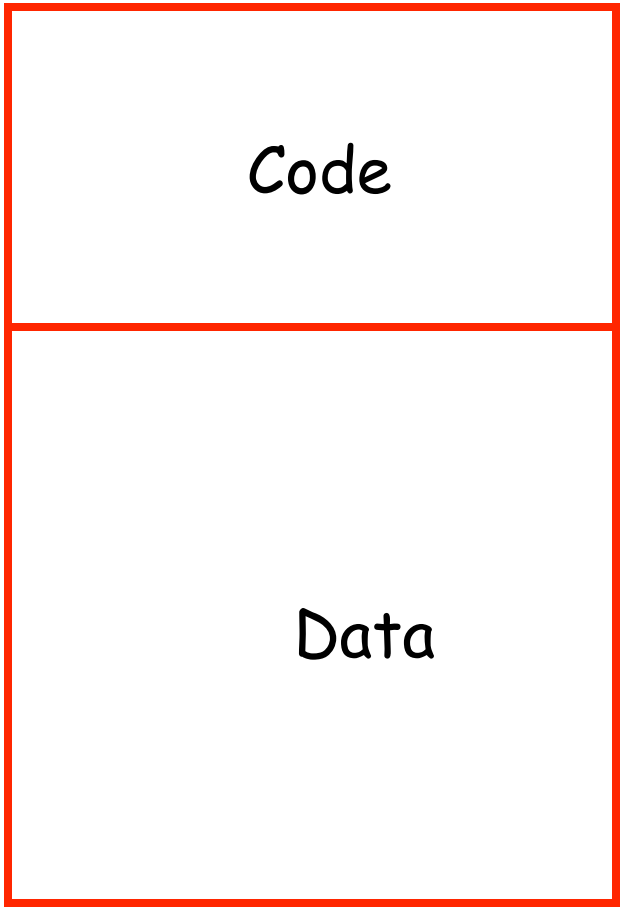
- By tradition, pictures of machine runtime memory organization have:
  - Low address at the top
  - High address at the bottom
  - Lines delimiting areas for different kinds of data

- These pictures are simplifications
  - E.g., not all memory need be contiguous
    - Think, for example, of what happens with virtual memory
  - But pictures help describe what the kinds of memory are and what compiler needs to do with them

What is Other Space?

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

- Compiler is responsible for:
  - Generating code
  - Orchestrating use of the data area
    - This is tricky part of code generation: deciding what layout of data will be and then generating code that correctly manipulates that data (because code contains references to data)



Code Generation Goals

- Two overall goals for code generation:
  - Correctness
    - Generate code that faithfully implements the programmer's program
  - Speed
    - Make good use of resources and run reasonably quickly

- Most complications in code generation come from trying to be fast as well as correct
  - Easy to generate either in isolation
    - Correct code that runs slowly
    - Fast code that is not correct (this one is very easy – we can generate extremely fast code that gives wrong answers!)

Code Generation Goals

- Over time, a fairly elaborate framework has been developed detailing how a code generator and the corresponding runtime structures should be organized and implemented in order to meet the goal of producing code that is simultaneously fast and correct.
- First step in talking about this is to discuss something called activations

First, Two Assumptions about Execution

1. Execution is sequential; control moves from one point in a program to another in a well-defined order

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

Do these assumptions always hold?

First, Two Assumptions about Execution

1. Execution is sequential; control moves from one point in a program to another in a well-defined order

   Programming languages that have concurrency violate this assumption – statement after current statement might be in completely different thread

First, Two Assumptions about Execution

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

   Advanced Control constructs – exceptions and call/cc (call with current continuation – google it) – affect flow of control in fairly dramatic ways

   C++ and Java throw and catch style exceptions – a thrown exception might escape from multiple procedures before it is caught, so no guarantee that a call to [a procedure that can throw an exception] will end up with control returning to the line following the procedure call.

We may or may not discuss these advanced features later. We focus on ideas basic to all implementations. Those languages with advanced features build upon these basic principles.

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
  - So all the statements that are executed between the moment P is called, and the moment P returns, including all the functions that P itself calls

Lifetimes of Variables

- The *lifetime* of a variable x is the portion of execution in which x is defined
  - All the steps of execution from the time that x is created until the time that x is destroyed or deallocated

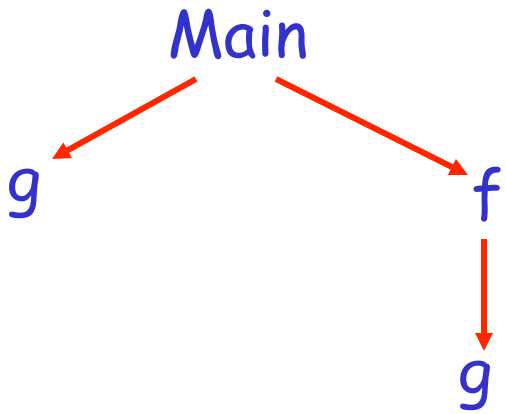
- Note that
  - Lifetime is a dynamic (run-time) concept
  - Scope is a static concept
    - Recall scope refers to the portion of the program text in which the variable is visible

Activation Trees

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

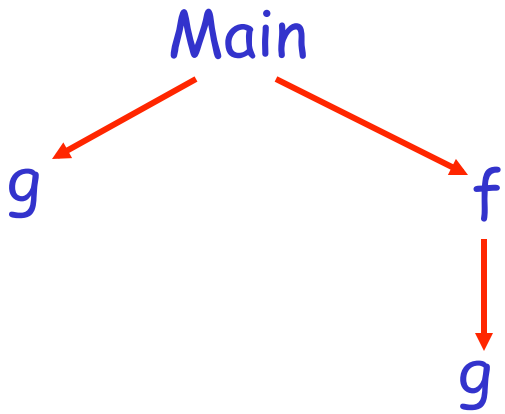
Example

```
Class Main {
    g() : Int { 1 };
    f():  Int { g() };
    main(): Int {{ g(); f(); }};
}
```



Example

```
Class Main {
   g() : Int { 1 };
   f():  Int { g() };
   main(): Int {{ g(); f(); }};
}
```



Note: tree shows containment of lifetime.

E.g. g's lifetime is contained in main's

E.g., lifetime of left branch g is disjoint from lifetime of f, since they are siblings in tree.

E.g., multiple activations of g (every call is another activation)

Example 2 (Recursion)

```
Class Main {
   g() : Int { 1 };
   f(x:Int):  Int { if x = 0 then g() else f(x - 1) fi};
   main(): Int {{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 {
    g() : Int { 1 };
    f():  Int { g() };
    main(): Int {{ g(); f(); }};
}
```

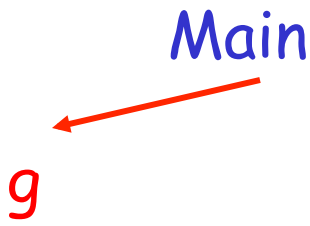
Main

Stack

Main

Example

```
Class Main {
    g() : Int { 1 };
    f():  Int { g() };
    main(): Int {{ g(); f(); }};
}
```



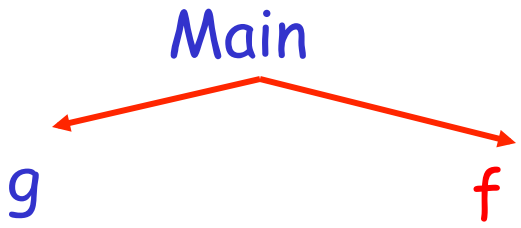
Stack

Main

g

Example

```
Class Main {
    g() : Int { 1 };
    f():  Int { g() };
    main(): Int {{ g(); f(); }};
}
```



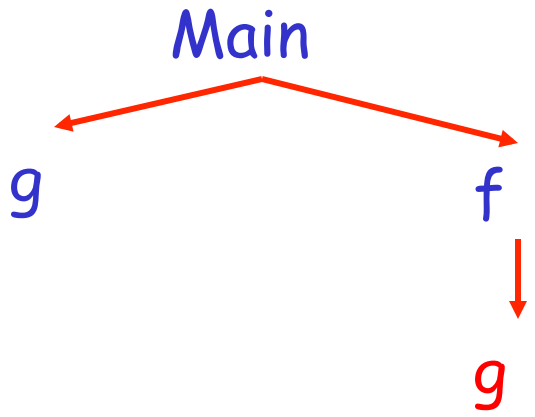
Stack

Main

f

Example

```
Class Main {
    g() : Int { 1 };
    f():  Int { g() };
    main(): Int {{ g(); f(); }};
}
```



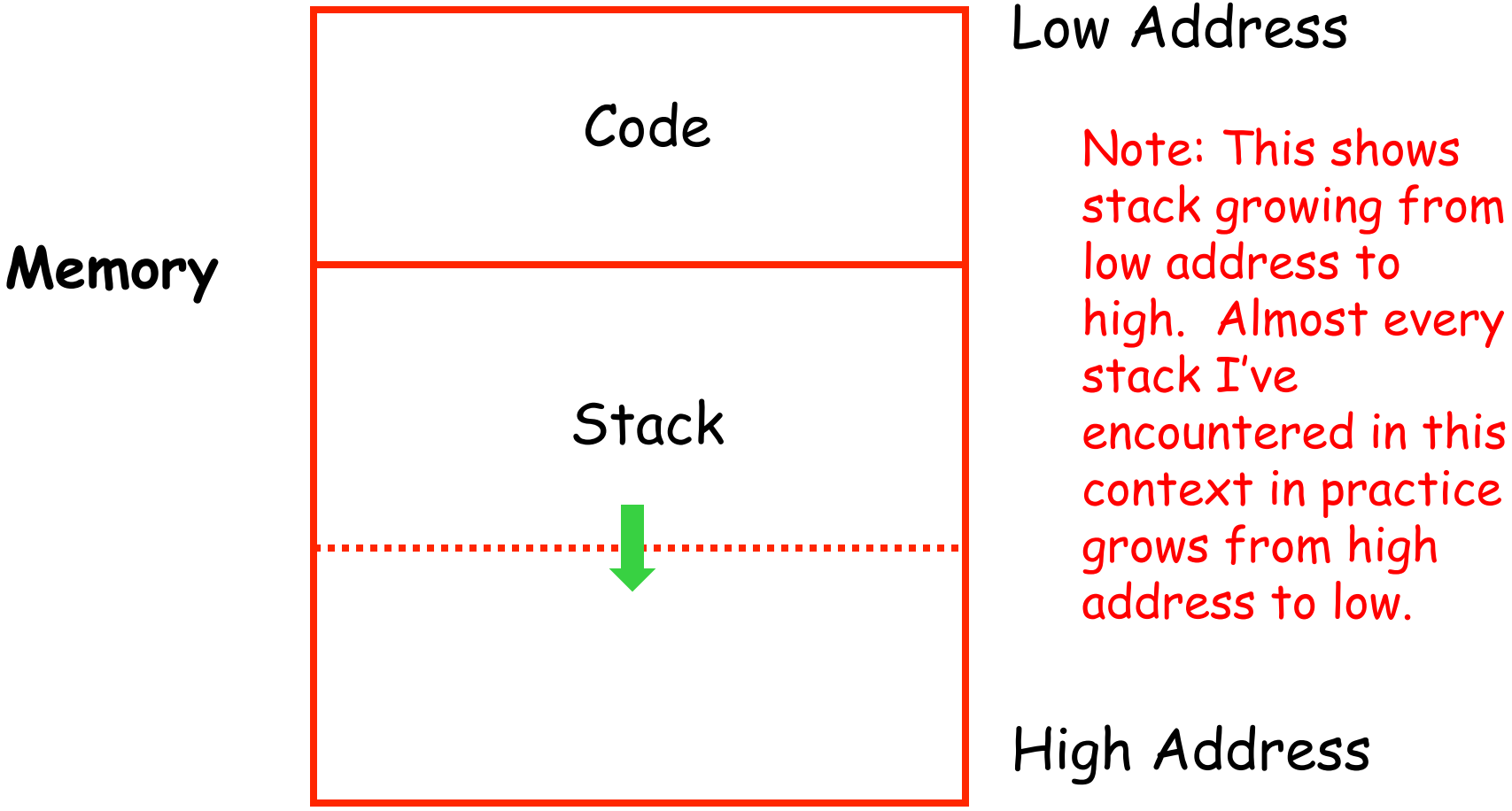
Stack

Main

f

g

Revised Memory Layout



Note: This shows stack growing from low address to high. Almost every stack I've encountered in this context in practice grows from high address to low.

Activation Records (What's in Them?)

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

- Activation records contain more than you might expect:
  - If procedure F calls G, then G's activation record contains a mix of info about F and G.

Activation Records (Why?)

- There is state associated with each procedure – information required in order to properly execute the procedure, and this needs to be kept somewhere
  - Activation record is for storing this state

- So, let's look at this in a bit more detail…

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.
  - And F's activation record contains state relevant to suspended procedure F
    - E.g., values of variables local to 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

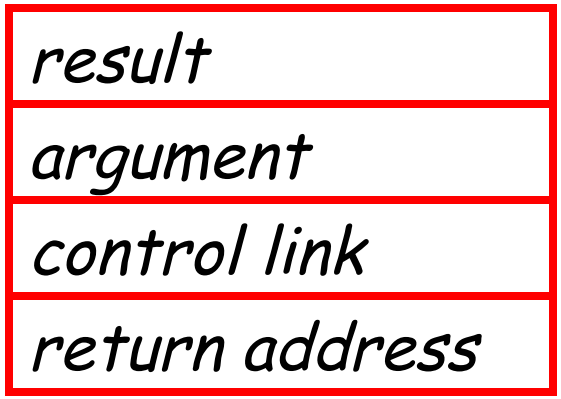
The Contents of a Typical AR for *G*

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

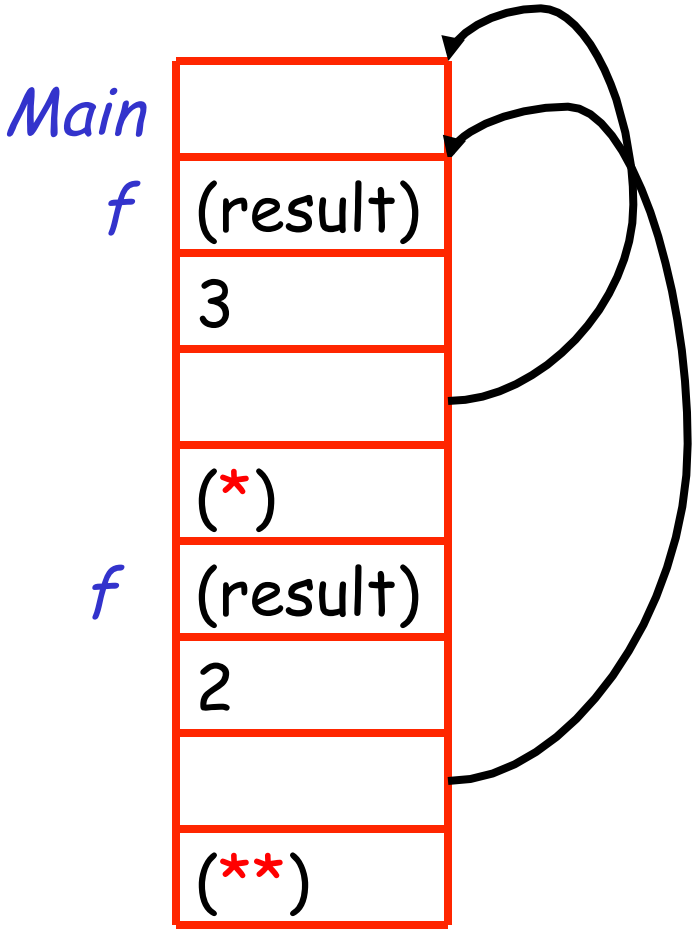
Example 2, Revisited

```
Class Main {
   g() : Int { 1 };
   f(x:Int):Int {if x=0 then g() else f(x - 1)(**)fi};
   main(): Int {{f(3); (*)
 }};}
```

AR for f:



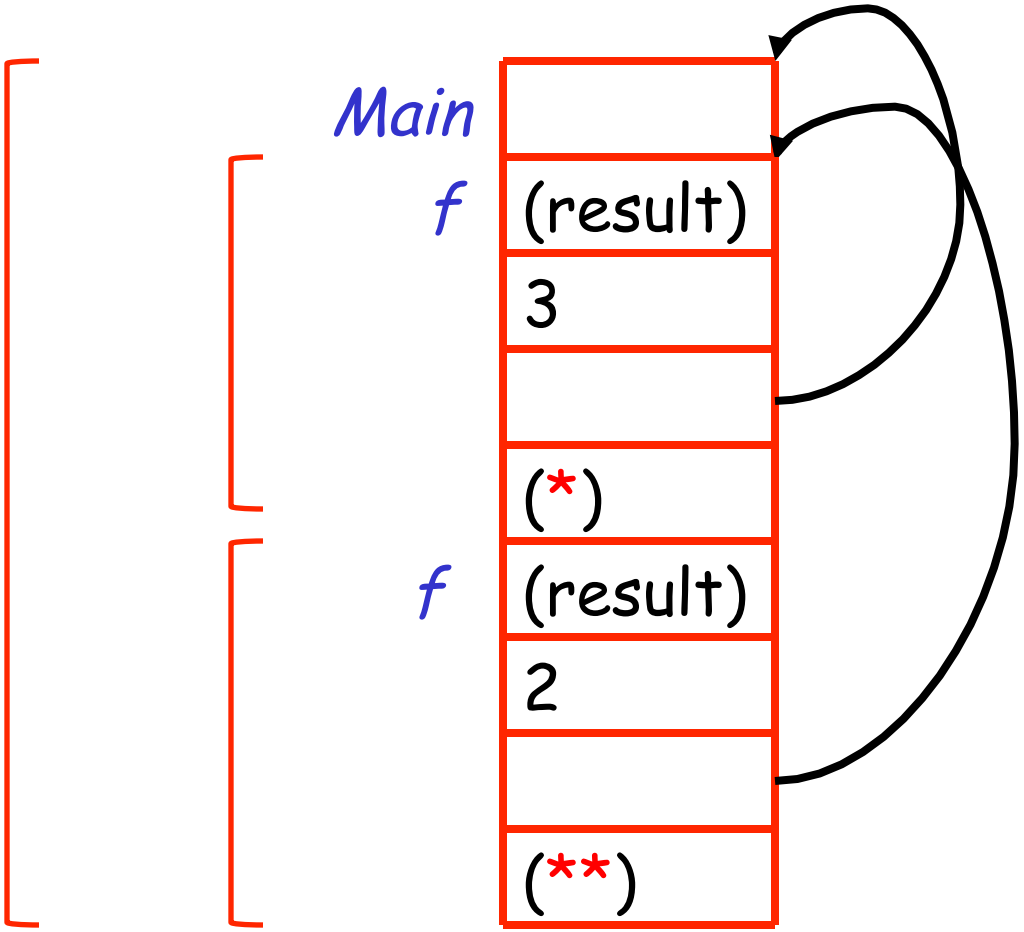
Stack After Two Calls to f



Notes

- Main has no argument or local variables 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, etc.
  - Ex. Many compilers don't use a control link entry: don't need it to find the calling activation record
    - In project, the COOL compiler does not use a control link
  - Ex. Some compilers store return address in register

Note Also



Stack is both a "stack" of activation records and a "stack" of individual entries.
While we think of it as a collection of frames, it is really just one contiguous portion of memory, in effect a gigantic array.
And some compiler writers take advantage of this.

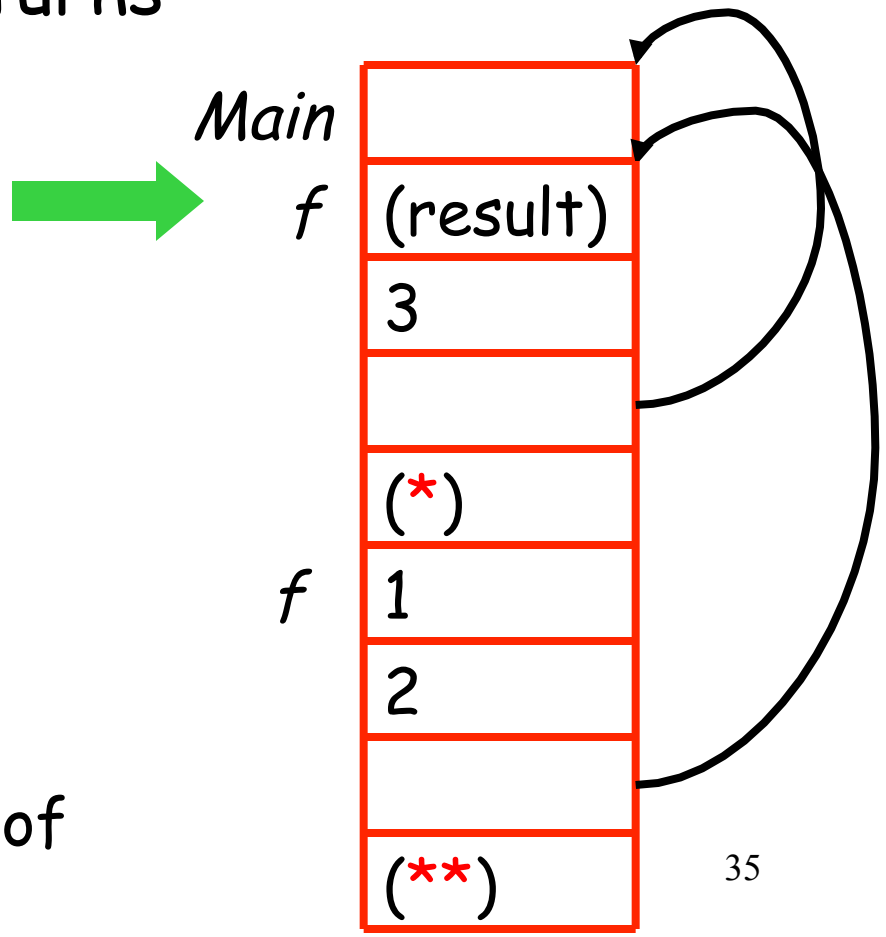
What Happens After an Activation Returns?

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

Green arrow indicates frame that is now top AR on stack

(But note previous top stack frame still resides in memory!)

AND we need to access it (to get the result of the call f(2), the result being at top of AR for call f(2))



Discussion

- The advantage of placing the return value 1st in a frame is that the caller can find it at a fixed offset from its own frame
  - Knows it is in first position of AR that was old top of stack – at fixed offset from current top of stack

- There is nothing magic about this organization
  - Can rearrange order of frame elements
  - Can divide caller/callee responsibilities differently
  - Only metric: An organization is better if it improves execution speed or simplifies code generation

Discussion (Cont.)

- Real compilers hold as much of the frame content as possible in registers
  - Especially the method result and arguments
    - Because these are accessed so frequently

The Main Point

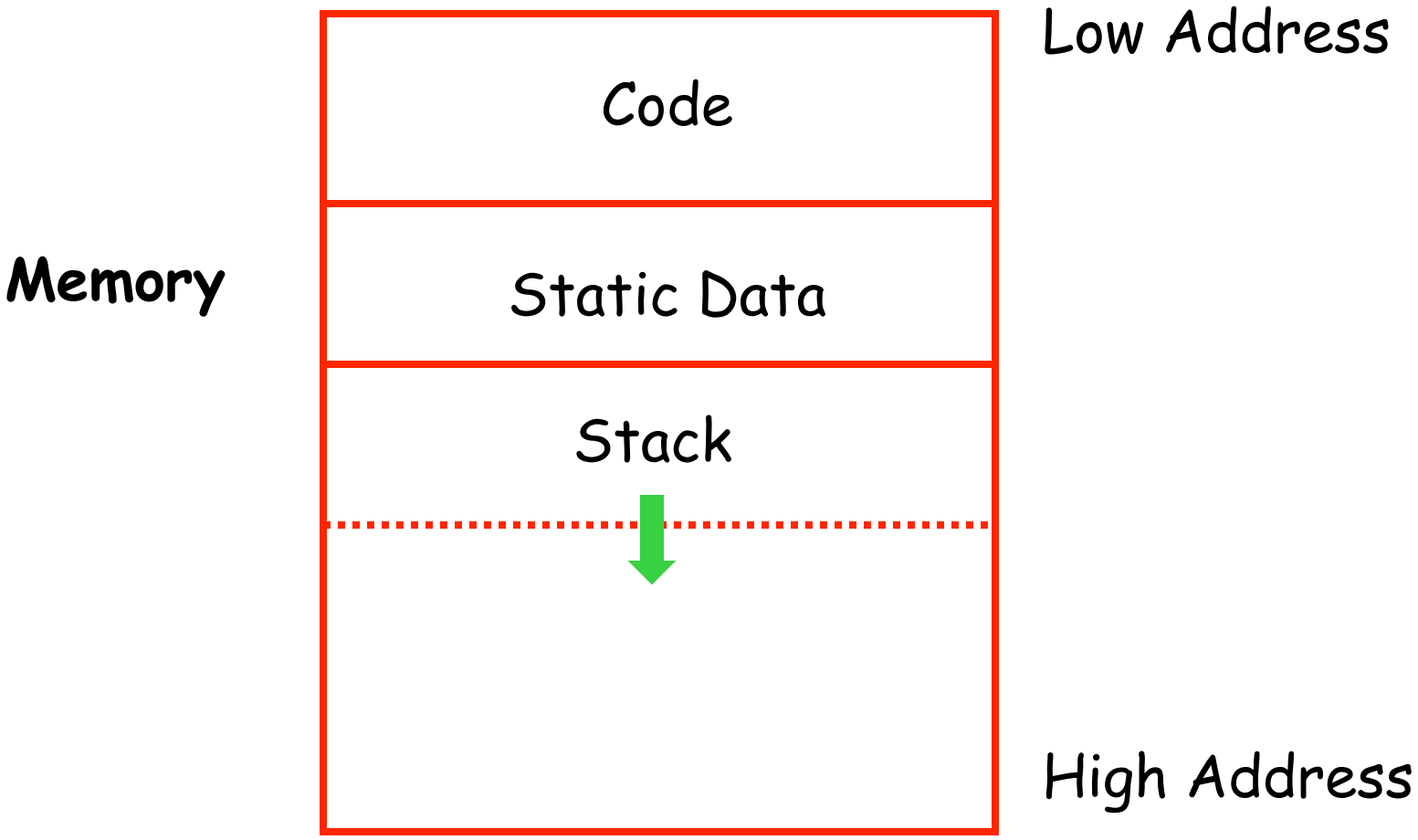
The compiler must determine, at compile-time, the layout of activation records and generate code that correctly accesses locations in the activation record

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

Globals

- All references to a global variable point to the same object (this is defn of global, after all)
  - Can't store a global in an activation record
    - Because ARs are deallocated when activation complete
- Globals are assigned a fixed address once
  - Variables with fixed address are said to be "statically allocated", because allocated at compile time
    - Compiled decides where they will be placed, and they stay there for entire execution of the program
- Depending on the programming language, there may be other statically allocated values
  - We'll see examples of this later

Memory Layout with Static Data



Heap Storage

- Also, a value that outlives the procedure that creates it cannot be kept in the AR

  method foo() { new Bar }

  The Bar value must survive deallocation of foo's AR, because the Bar object is the result of foo(), so it must be accessible to foo()'s caller after foo() exits

- Languages with dynamically allocated data (such as above) generally use a *heap* to store dynamic data

Summary (thus far)

- The code area contains object code
  - For many languages, fixed size and read only
  - Some languages allow code created at run time!

- The static area contains data (not code) with fixed addresses (e.g., global data)
  - Fixed size, may be readable or writable

Summary (thus far)

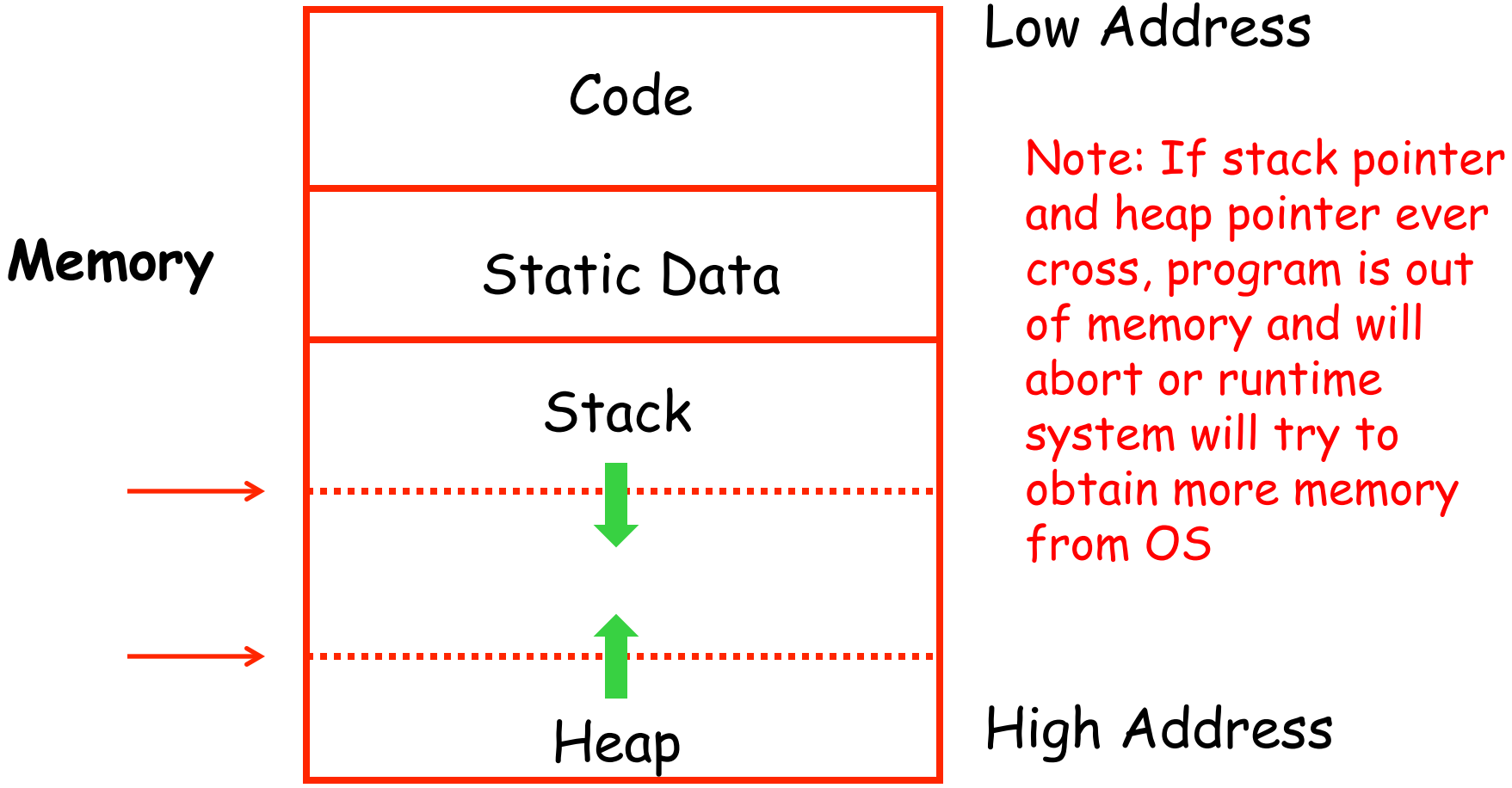
- The stack contains an AR for each currently active procedure
  - Each AR usually fixed size, contains locals

- Heap contains all other data (including dynamically allocated data)
  - In C, heap is managed by malloc and free
  - In Java, have new and garbage collection

Notes

- 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



Note: Heaps I've seen grow from low address to high!

Data Layout

- Low-level details of machine architecture are important in laying out data for correct code and maximum performance

- Chief among these concerns is *alignment*

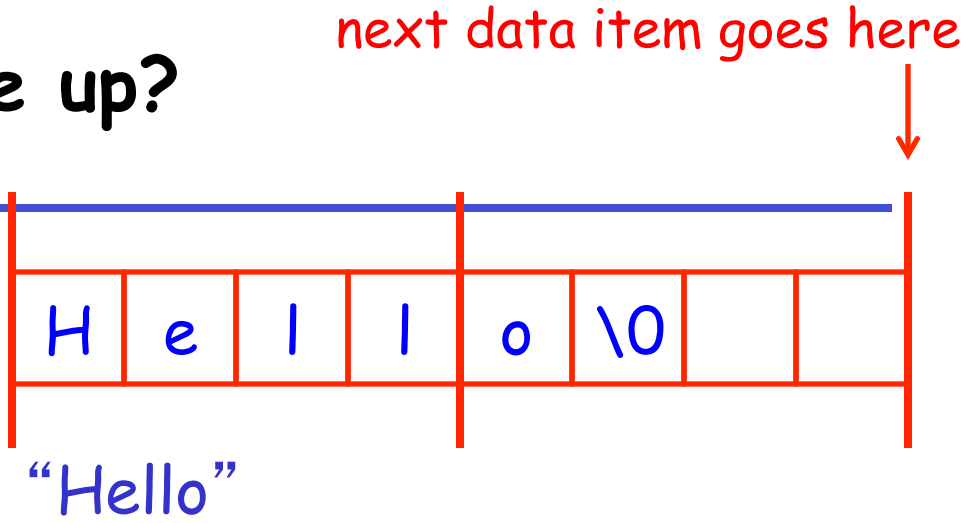
Alignment

- Most modern machines are either 32 or 64 bit
  - 8 bits in a byte
  - 4 or 8 bytes in a word
  - Machines are either byte or word addressable
    - Meaning that the native language of the machine it is possible to reference memory by individual bytes or by words
- Data is *word aligned* if it begins at a word boundary (if you don't understand, please ask)
- Most machines have some alignment restrictions
  - Or performance penalties for poor alignment

Alignment

- Most machines have some alignment restrictions
  - Or performance penalties for poor alignment
- Restrictions come in two forms
  - If data not properly aligned, machine fails to execute instruction that referenced non-aligned data
    - E.g, "seg fault" or "bus error", or machine hangs.
  - If data not properly aligned, program executes correctly, but at significant performance cost
    - Often dramatic – can be 10 times slower to access misaligned data!
    - Execution far more efficient if data properly aligned

Where does this come up?



- Example: A string

  Takes 5 characters (without a terminating \0)

- To word align next datum, add 3 "padding" characters to the string
  - Or 2, as above, if we include the '\0'

- The padding is not part of the string, it's just unused memory

So, General Strategy

- Align all data on word boundaries
- If the data does not require an exact multiple of 4 (or 8, as the case may be) of memory, just pad so that it does
- Occurs not just with strings, but with character arrays as well
  - E.g., you ask for an array large enough for 10 chars, then you get enough memory for 12 (assuming char is a single byte)

On to Code Generation

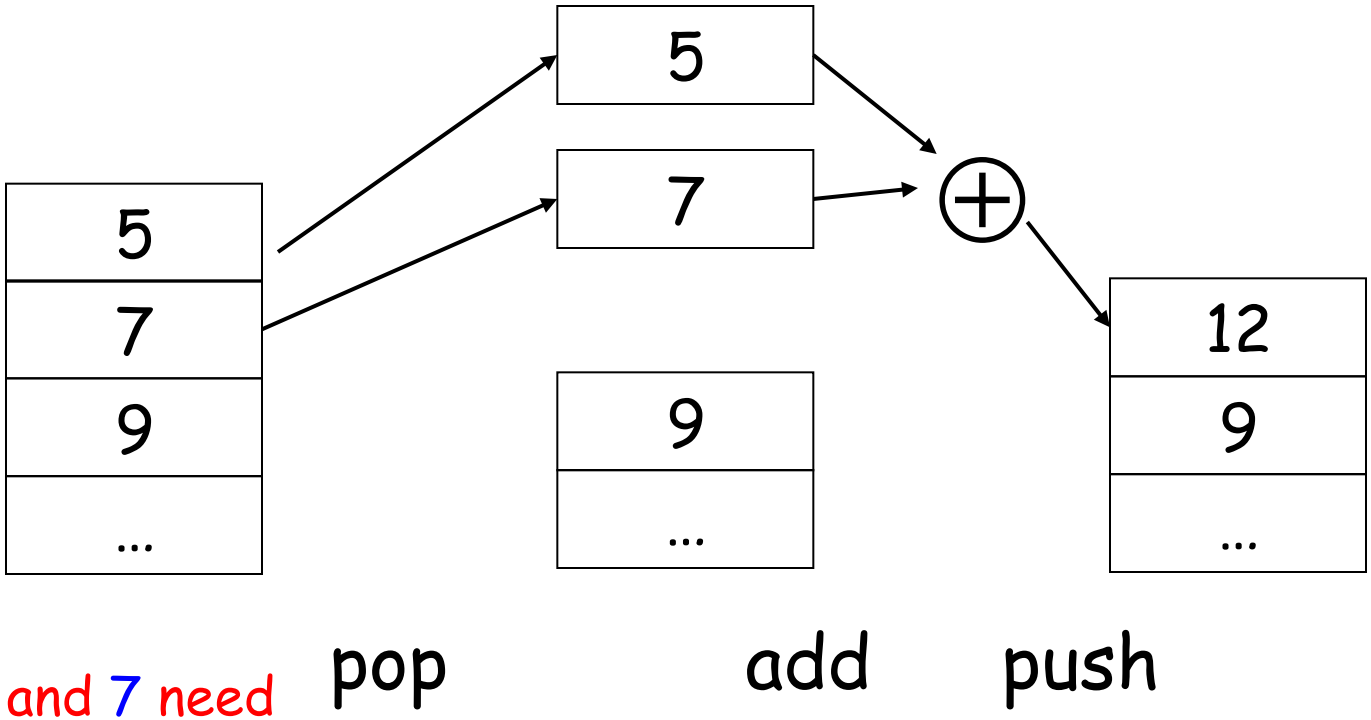
- Done with run time stuff, on to code generation
- Simplest model for code generation is called a stack machine, the subject of the next few slides

Stack Machines

- Only storage is (surprisingly) a stack
- A simple evaluation model
- No variables or registers
- A stack of values for intermediate results
- Each instruction $r = F(a_1, a_2, \ldots, a_n)$
  - Pops n operands from the top of the stack
  - Computes the operation F using those operands
  - Pushes the result r on the stack

Example of Stack Machine Operation

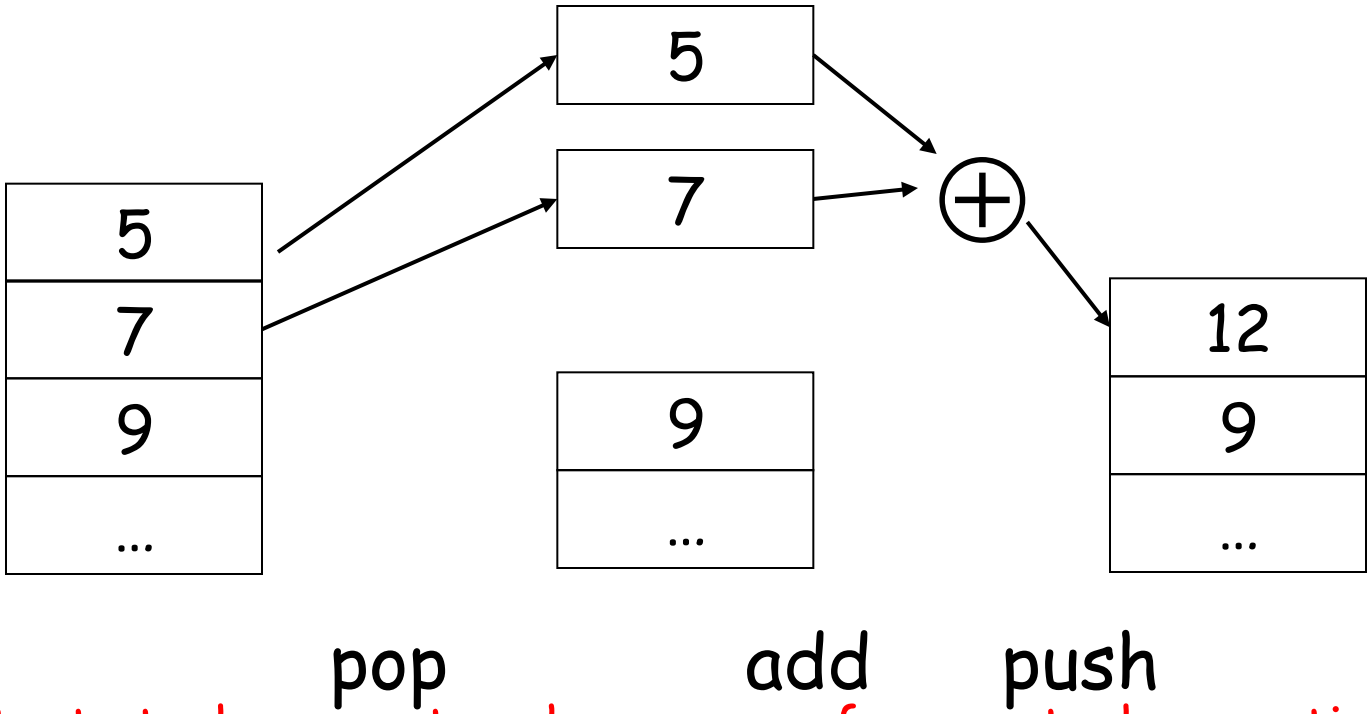
- The addition operation on a stack machine: 5+7



Note: 5 and 7 need to be pushed onto stack!

Example of Stack Machine Operation

- The addition operation on a stack machine: 5+7



Important stack property: when you perform a stack operation, the contents of the stack prior to beginning of evaluation of the expression will be preserved

Example of a Stack Machine Program

- Consider language with just 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
```

Why Use a Stack Machine ?

- Location of the operands is implicit
  - Always on the top of the stack
  - So no need to specify operands explicitly
- No need to specify the location of the result
- Instruction "add" as opposed to *register machine* instruction "add r_1, r_2, r_3"
  - ⇒ Smaller encoding of instructions
  - ⇒ More compact programs
- This is one reason why Java Bytecodes use a stack evaluation model
  - Code often transmitted via Internet, so benefit to having more compact code

Why Use a Stack Machine ?

- Because 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

Why a Register Machine ?

- Generally faster because we can place data exactly where we want it to be
  - Fewer intermediate operations (such as pushing and popping) required to get at the data

Middle Ground?

- There is an intermediate point between a pure stack machine and a pure register machine
- An n-register stack machine
  - Conceptually, keep the top n locations of the pure stack machine's stack in registers
- The particular variant of this that we are most interested in is the 1-register stack machine
  - Turns out there is significant benefit even from having just one register to store the data at top of stack
  - The register is called the accumulator (it accumulates results of operations)

Advantage of 1-Register Stack Machine

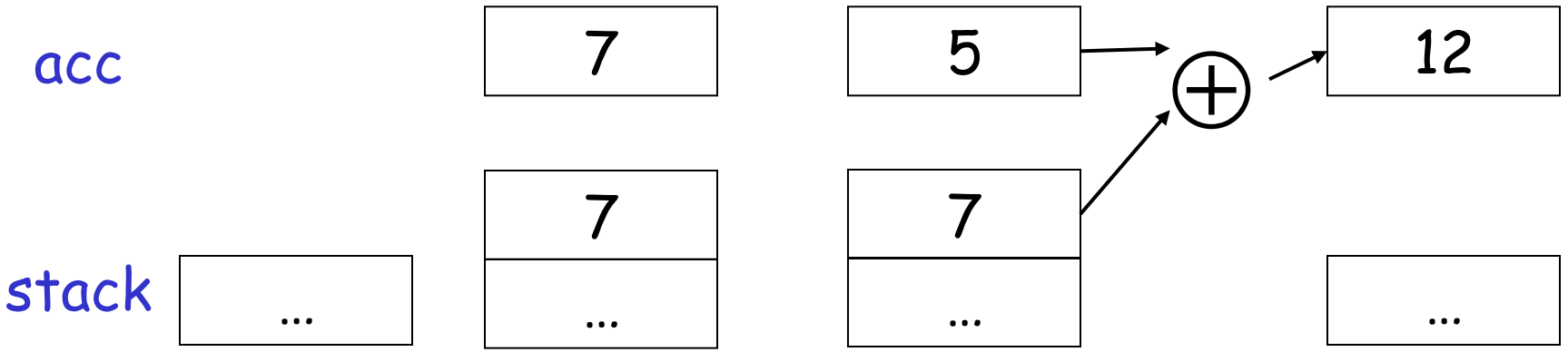
- In pure stack machine, the add instruction does 3 memory operations
  - Two reads and one write to the stack
  - The top of the stack is frequently accessed
- In 1-register stack machine, the add does a lot of the work out of the one register
  - Register accesses are faster
- The "add" instruction is now

  acc ← acc + top_of_stack

  - Only one memory operation!

Stack Machine with Accumulator. Example

- Compute 7 + 5 using an accumulator



Notes

- It is very important evaluation of a subexpression preserves the stack
  - Stack before the evaluation of 7 + 5 is 3, <init>
  - Stack after the evaluation of 7 + 5 is 3, <init>
  - The first operand is on top of the stack

General Strategy: Stack Machine with Accumulator

- Consider a COOL operation: $op(e_1,\ldots,e_n)$
  - Note $e_1,\ldots,e_n$ are subexpressions
  - Each will require (perhaps involved) evaluation
- The process:
- For each e_i ($1 < i < n$):
  - Compute e_i (recursively, using same strategy)
    - Result ends up in accumulator
  - Push result on the stack
    - For e_n, just evaluate, don't push onto stack
- Pop $n-1$ values from the stack, compute op
- Store result in accumulator

Notes:

- Invariant: After evaluating an expression e, the accumulator holds the value of e and the stack is unchanged
  - Memory portion of stack is identical to what it was before we started evaluating e

- Put another way: Expression evaluation preserves the stack

A Bigger Example: 3 + (7 + 5)

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

A Bigger Example: 3 + (7 + 5)

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

Note recursion and evaluation of subexpression 7 + 5