CS143: Runtime

David L. Dill

Stanford University

Runtime

- Semantic Analysis (wrap up)
  - Error Recovery
- Runtime Conventions
  - Memory Regions
  - Stack Allocation
  - Expression Evaluation on a Stack

Error Recovery

Goal: Report errors after the first one

$$\text{let } y : \text{Int} \leftarrow \overset{\text{undeclared.}}{x} + 2 \text{ in } y + 3$$

Type for undeclared x?

If $x : \text{Object}$, $x + 2$ will cause another error.

Reports too many errors.

Another Solution

Compiler stores "No_type" for erroneous expressions

No_type $\leq$ C for all types C

So No_type $\sqcup$ C is always C

All operations/assignments are ok.

let y : Int $\leftarrow \underbrace{x + 2}_{\text{No\_type} \sqcup \text{Int} \rightarrow \text{Int}}$ in y + 3 $\leftarrow$ only one error

No_type

Types no longer a tree

Not a problem unless you have code that assumes it's a tree

Not required in project

Runtime Conventions

Memory Regions

Machine architecture does not (usually) dictate how:

Memory is used

Function calls work

etc.

But these need to be handled consistently

E.g., Function calls: caller & callee need to agree on where arguments will be.

Memory Organization

Process has many different memory regions

Set up by OS, initialization code.

I will ignore dynamic loading, shared memory, etc.

Memory Regions

Code - the instructions

Data - several kinds

- Global variables & constants
- Stack - for function call data
- Heap - for other dynamically created data (e.g. from "malloc" in C)

Executable code needs to be able to find these.

Finding Data

Global Variables and constants

- Occupy fixed addresses in memory
- Compiler decides where they go.

Heap-allocated objects

- Pointers are stored/passed in global, local variables, parameters, etc.

Formals & locals: **Next**

Stack Allocation

Use of Stack

Designed for sequential code

Breaks down if

- In multithreaded code.
- If functions can access variables of containing functions AND functions are treated as first-class values (requires "closures")

Assume these are not issues for this lecture.

Program Stack

During function call, need temporary storage for:

- Actual parameter values
- Local variables
- Saved machine state (e.g. registers)
  - Return address
  - Return value
  - Pointer to previous activation record ← "control link"
  - (sometimes other stuff)

Function Calls

call f_1

call f_2

call f_3

return from f_3

return from f_2

return from f_1

Function Calls

call f_1
  call f_2
    call f_3
    return from f_3
  return from f_2
return from f_1

Nesting structure.

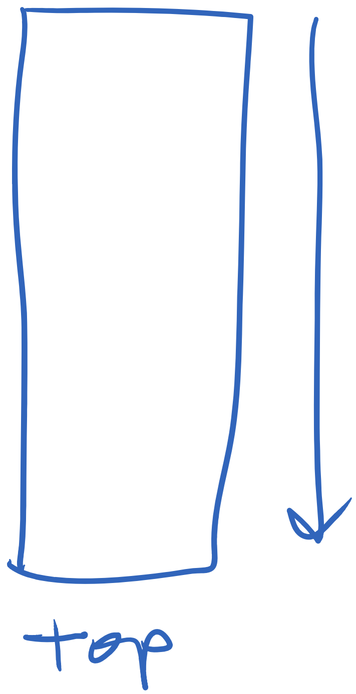
Stack allocation will work.

Def: Data to execute a function is stored in an activation record (aka "stack frame").

Under above assumptions, these can be stored in the stack.

- Actual parameter values
- Local variables
- Saved machine state (e.g. registers)
- Return address
- Return value
- Control link
- (sometimes other stuff)

Stacks Grow Down (for now)



AR organization

result
arguments
control link
return address

There are many possibilities

Compiler writer must decide and generate code to

- set up / tear down
- access data

Responsibilities must be divided between caller and callee.

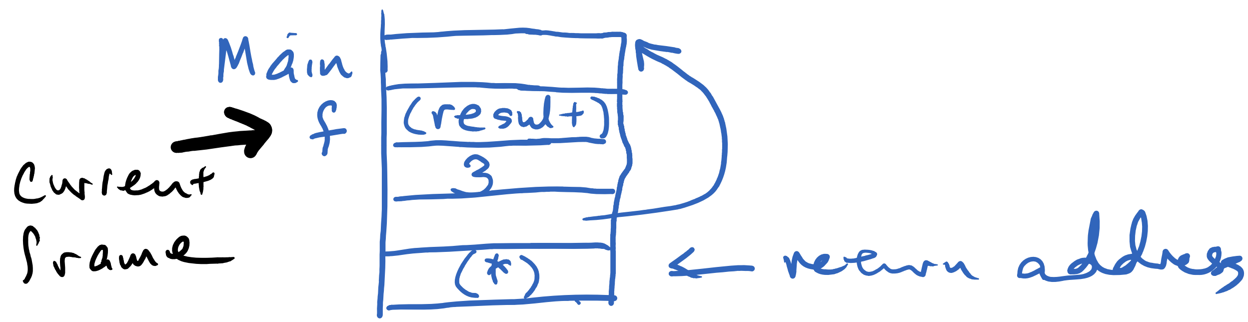
Design Considerations:

- Simplicity of codegen
- Efficiency (minimize data moves)
- Compatibility with other code.

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

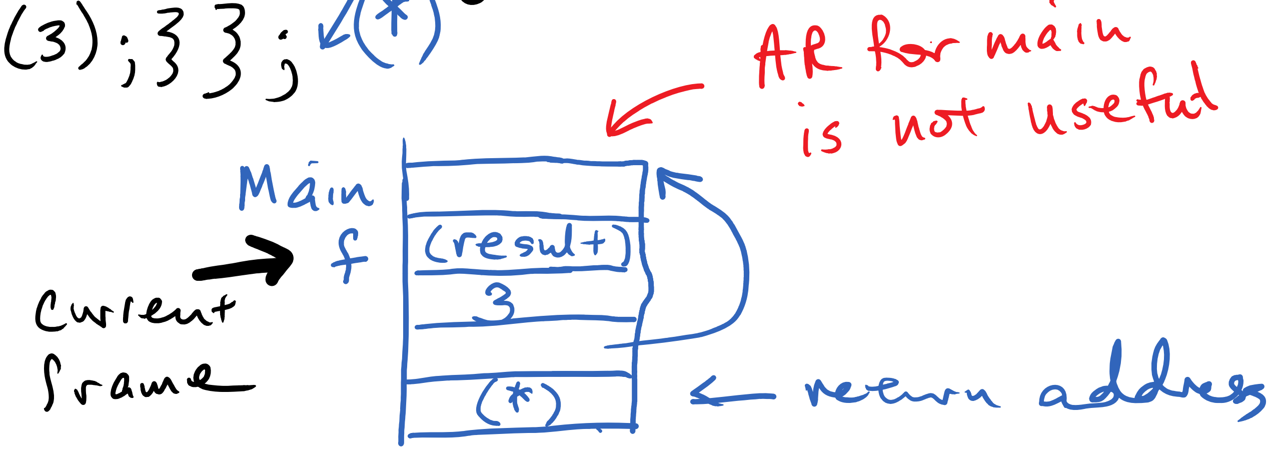
(*)

result
argument
control link
return address

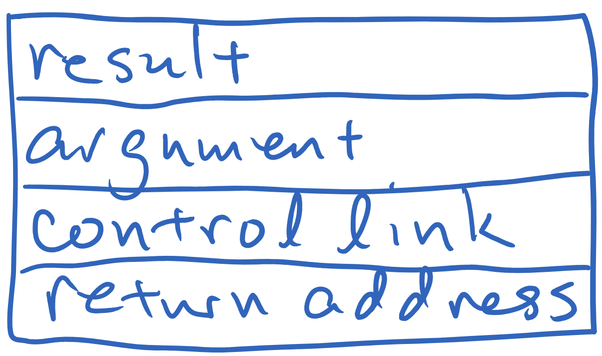


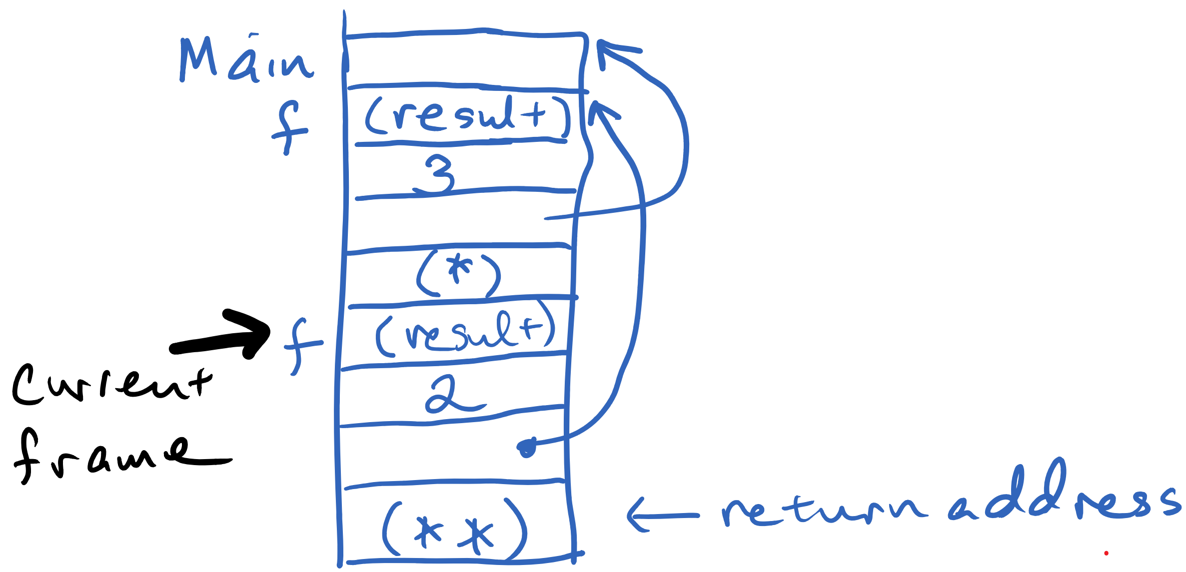
```
Class Main {
  g() : Int { 1 };
  f(x: Int) : Int { if x = 0 then g() else f(x-1) fi };
  main() : Int {{ f(3); }};   (*)
}
```

result
argument
control link
return address



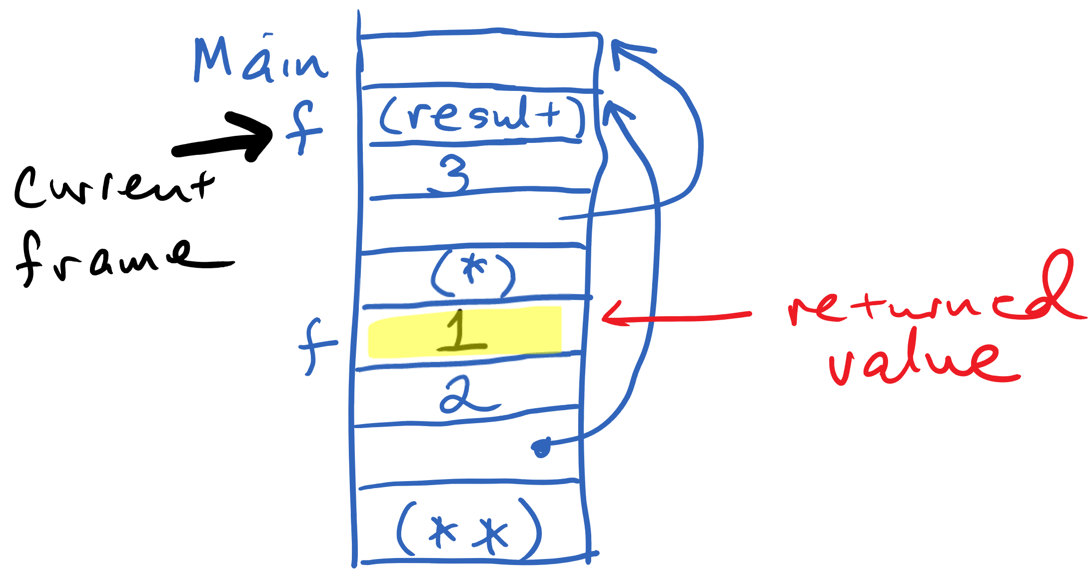
```
Class Main {
  g():Int { 1 };
  f(x:Int):Int { if x=0 then g() else f(x-1) (**) fi};
  main():Int {{ f(3); (*) }};
}
```





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

result
argument
control link
return address



Heap Allocation

Problem: Some data lives longer than the function call that creates it.

Solution: Heap allocation.

Heap: One or more reserved memory regions

C: malloc — marks free memory as used.
free — marks used memory as free

Many languages: new A allocates a new instance of A

Storage may be reclaimed automatically (e.g. garbage collection)

Memory Alignment

Almost all modern architectures are <u>byte addessable</u> (8-bit bytes)

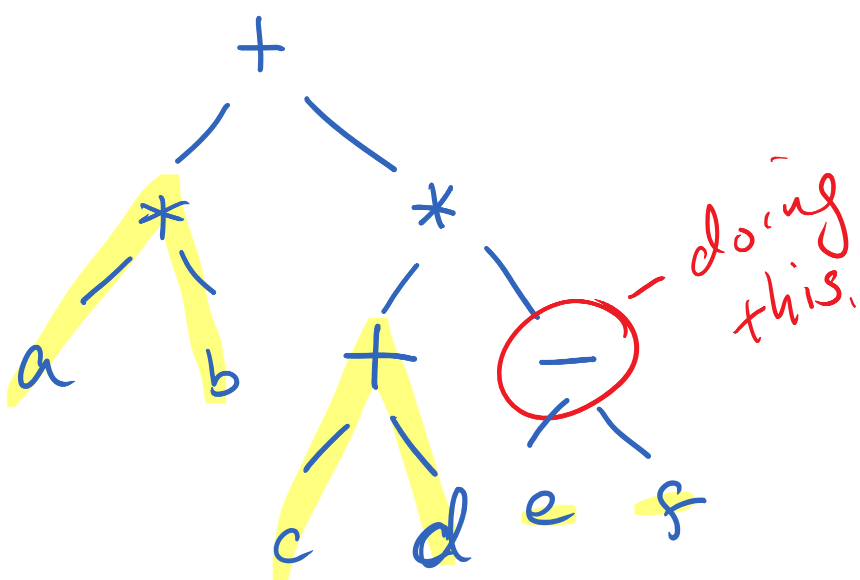
But many instructions require (or strongly reward) alignment of data to 32-bit or 64 bit boundaries.

E.g., A 32-bit integer at addess 10007 would not work well with an add instruction

"Padding" - unused bytes so next object is aligned.

Expression Evaluation with a Stack

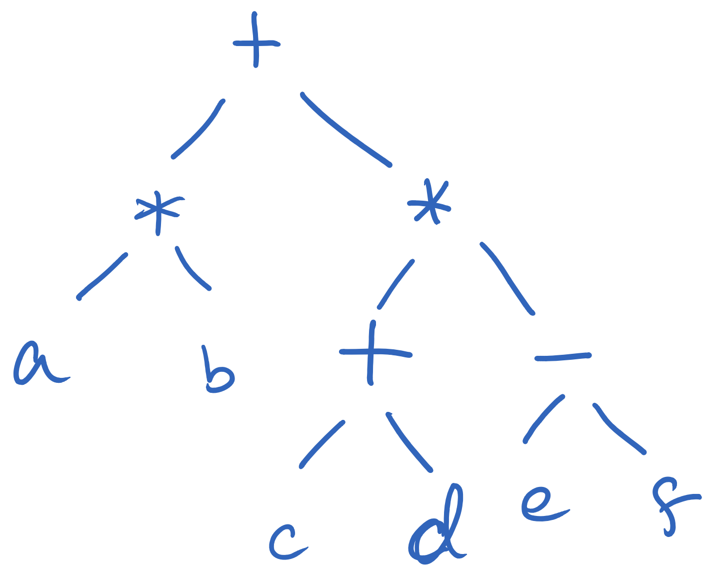
Intermediate Values



Evaluate left-to-right bottom-up.

Need to save $a*b$, $c+d$, while computing $e-f$.

Intermediate Values

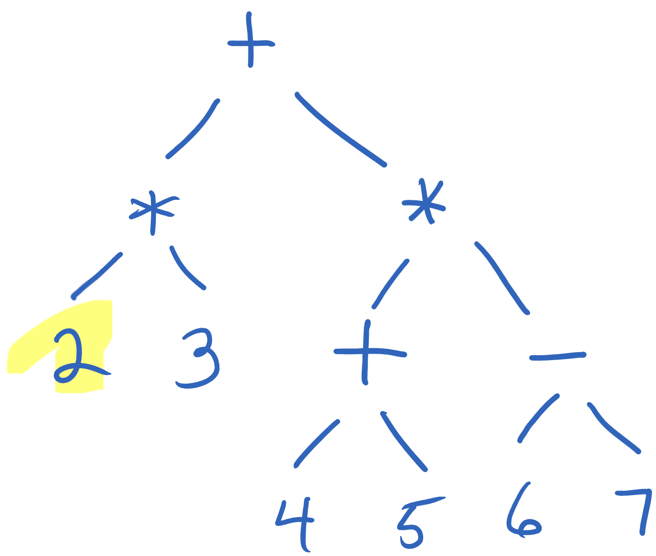


Evaluate left-to-right bottom-up.

Need to save $a*b$, $c+d$, while computing $e-f$.

This requires temporary storage.

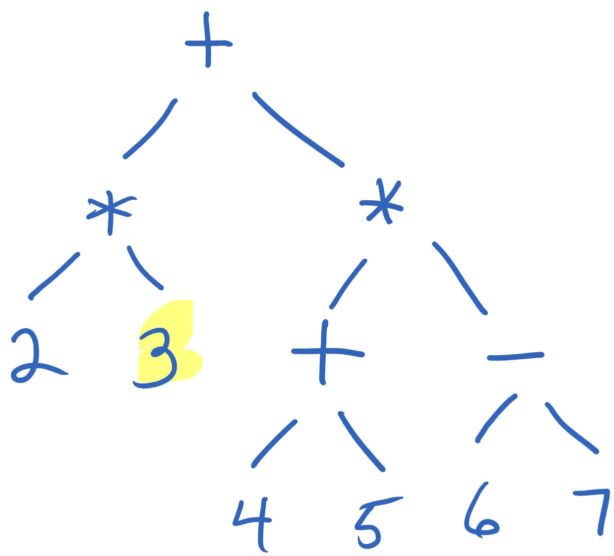
Intermediate Values



stack (grows <u>up</u>)

2

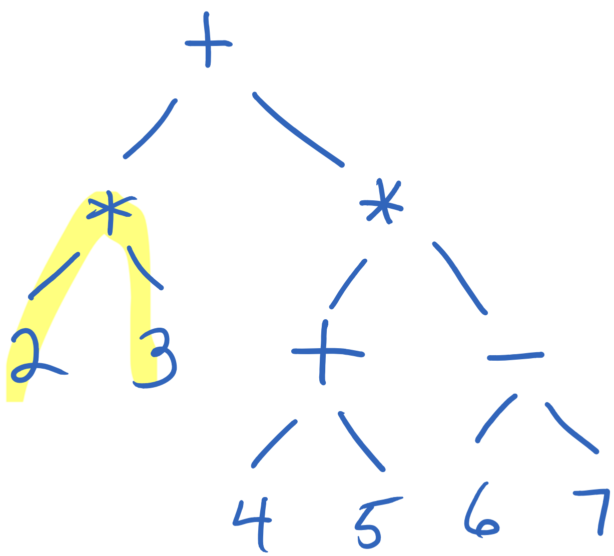
Intermediate Values



stack (grows **up**)

3
2

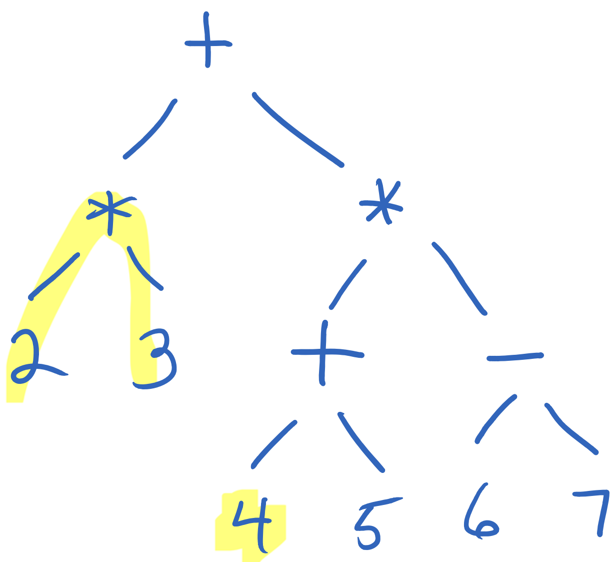
Intermediate Values



stack (grows up)

6

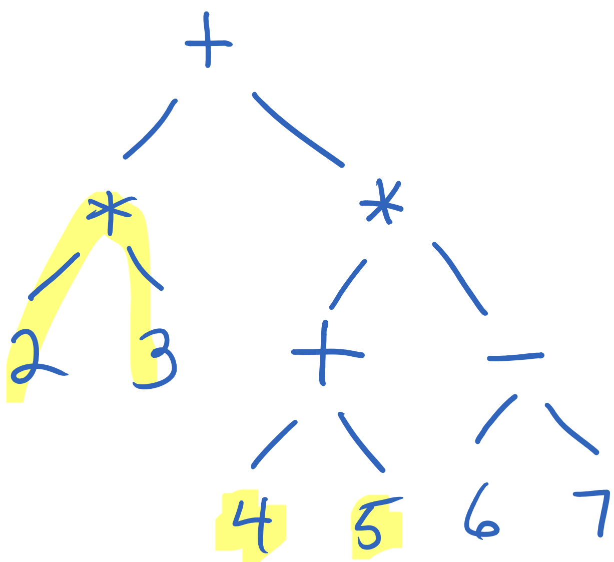
Intermediate Values



Stack (grows <u>up</u>)

4
6

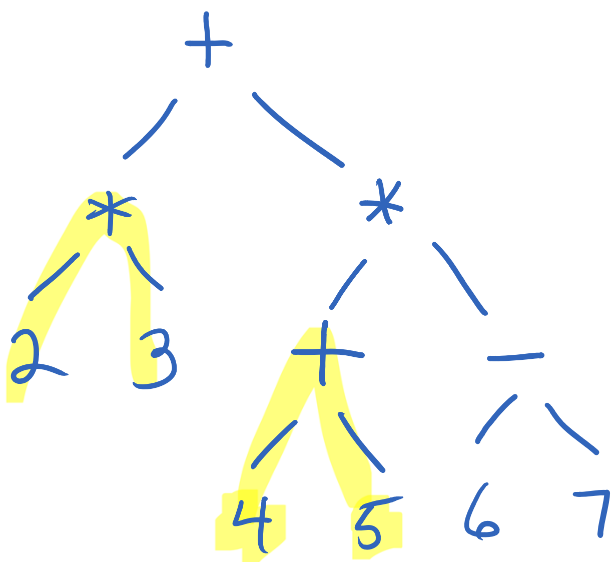
Intermediate Values



Stack (grows <u>up</u>)

5
4
6

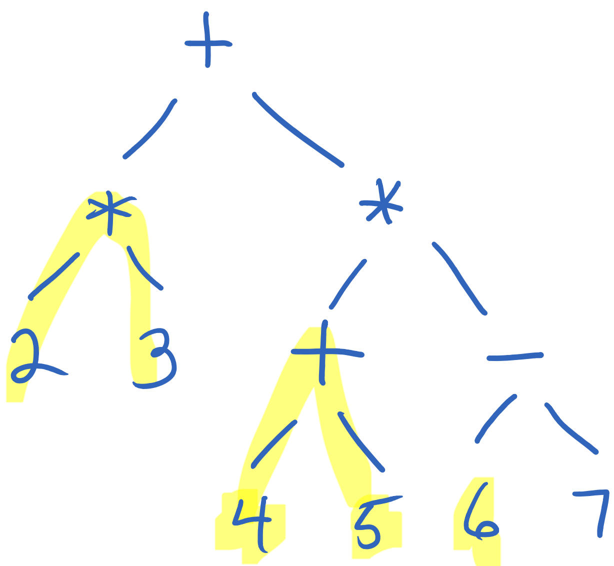
Intermediate Values



Stack (grows up)

9
6

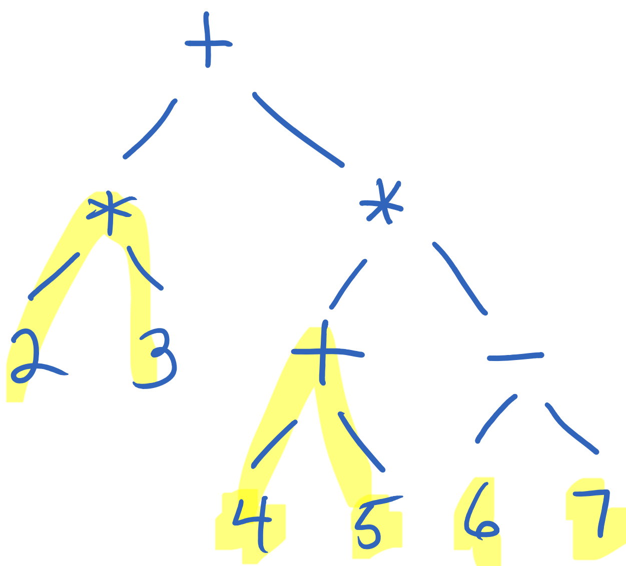
Intermediate Values



Stack (grows up)

6
9
6

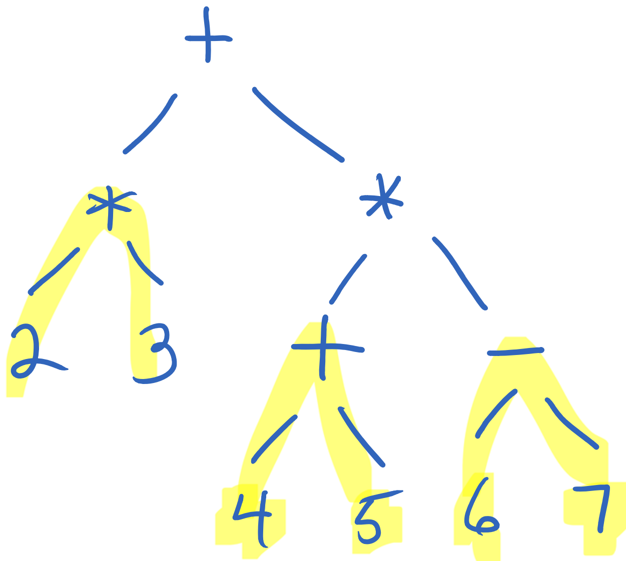
Intermediate Values



Stack (grows up)

7
6
9
6

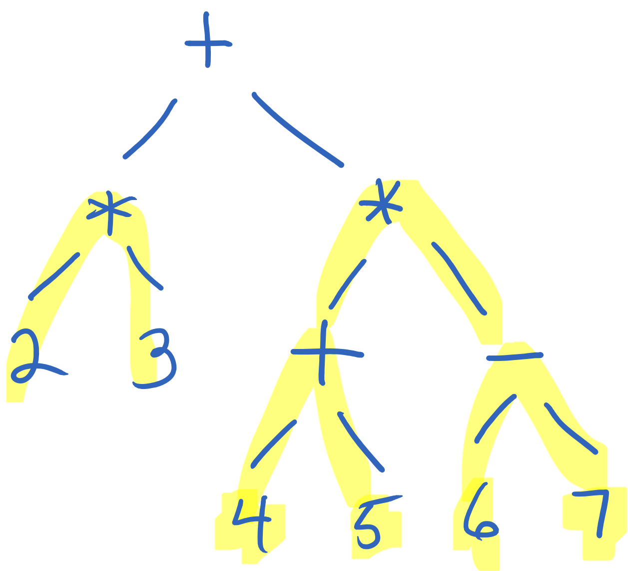
Intermediate Values



Stack (grows <u>up</u>)

−1
9
6

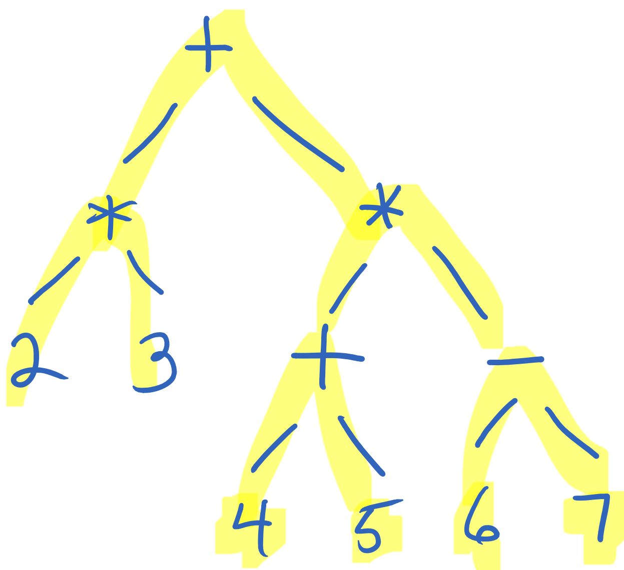
Intermediate Values



Stack (grows <u>up</u>)

−9
6

Intermediate Values



stack (grows up)

−3

Code Generation for Expressions

Instructions:

push i — push the number i on the stack

add — pop top two numbers, add them, push the sum.

$2 + 3 \implies$ push 2
push 3
add

Code Generation for Stack Machines is Easy

No need to optimize scarce registers

Don't need to specify operands

always the top n values on the stack.

No need to specify destinations

Short instructions & compact programs.

Stack + Accumulator

Idea: Top of stack is heavily used
keep it in a fast register
(called the "accumulator" – <u>acc</u>)

add: acc ← acc + top of stack
(does not pop top of stack)