Code Generation

Martin Kellogg

Course Announcements

- PA3c2 (TAC) is due later this week (“before spring break”)
  - you should already have started, or you are behind
  - if there is demand from the class, I will consider a short extension on this assignment to e.g., Monday
- I have become aware of a **bug in the reference compiler**’s x86-64 module; a fix is forthcoming. For now, don’t trust it.
- Don’t forget there is a **midterm** in this class the week after spring break!

Agenda

- In this lecture (and, unfortunately, the one on the Monday after spring break…) I am going to show you how to do code generation for a **stack machine** (“Cool ASM”/”Cool bytecode”)

Agenda

- In this lecture (and, unfortunately, the one on the Monday after spring break…) I am going to show you how to do code generation for a **stack machine** (“Cool ASM”/”Cool bytecode”)
  - Including all of the complexity of handling objects, etc.

Agenda

- In this lecture (and, unfortunately, the one on the Monday after spring break…) I am going to show you how to do code generation for a **stack machine** (“Cool ASM”/”Cool bytecode”)
  - Including all of the complexity of handling objects, etc.
- A stack machine is **simpler** than the x86-64 we saw on Monday
  - This is on purpose (where’s the fun if I make it too easy…)

Agenda

- In this lecture (and, unfortunately, the one on the Monday after spring break…) I am going to show you how to do code generation for a **stack machine** (“Cool ASM”/”Cool bytecode”)
  - Including all of the complexity of handling objects, etc.
- A stack machine is **simpler** than the x86-64 we saw on Monday
  - This is on purpose (where’s the fun if I make it too easy…)
- Hard PA3 task for you: **merge**…

Agenda

- In this lecture (and, unfortunately, the one on the Monday after spring break…) I am going to show you how to do code generation for a **stack machine** (“Cool ASM”/”Cool bytecode”)
  - Including all of the complexity of handling objects, etc.
- A stack machine is **simpler** than the x86-64 we saw on Monday
  - This is on purpose (where’s the fun if I make it too easy…)
- Hard PA3 task for you: **merge**…
  - …what you know about x86-64 (i.e., last lecture)

Agenda

- In this lecture (and, unfortunately, the one on the Monday after spring break…) I am going to show you how to do code generation for a **stack machine** (“Cool ASM”/”Cool bytecode”)
  - Including all of the complexity of handling objects, etc.
- A stack machine is **simpler** than the x86-64 we saw on Monday
  - This is on purpose (where’s the fun if I make it too easy…)
- Hard PA3 task for you: **merge**…
  - …what you know about x86-64 (i.e., last lecture)
  - …with what you know about code generation for a simpler target (this lecture)

Agenda (for real): two lectures on…

- Stack machine basics
  - accumulator, stack pointer

Agenda (for real): two lectures on…

- Stack machine basics
  - accumulator, stack pointer
- Stack discipline, calling convention for our stack machine
  - with a bit of optimization thrown in to give you a taste of the idea

Agenda (for real): two lectures on…

- Stack machine basics
  - accumulator, stack pointer
- Stack discipline, calling convention for our stack machine
  - with a bit of optimization thrown in to give you a taste of the idea
- Object layout
  - and its interactions with subtyping

Agenda (for real): two lectures on…

- Stack machine basics
  - accumulator, stack pointer
- Stack discipline, calling convention for our stack machine
  - with a bit of optimization thrown in to give you a taste of the idea
- Object layout
  - and its interactions with subtyping
- Dispatch tables/vtables
  - this gets us to dynamic dispatch

Agenda (for real): two lectures on…

- Stack machine basics
  - accumulator, stack pointer
- Stack discipline, calling convention for our stack machine
  - with a bit of optimization thrown in to give you a taste of the idea
- Object layout
  - and its interactions with s
- Dispatch tables/vtables
  - this gets us to dynamic di

Q: What's on the midterm?

Agenda (for real): two lectures on…

- Stack machine basics
  - accumulator, stack pointer
- Stack discipline, calling convention for our stack machine
  - with a bit of optimization thrown in to give you a taste of the idea
- Object layout
  - and its interactions with s
- Dispatch tables/vtables
  - this gets us to dynamic di

Q: What's on the midterm?
A: **Only what we get through today**. I probably won't ask much about it.

Stack Machines

Stack Machines

- A simple evaluation model
  - No variables or registers

Stack Machines

- A simple evaluation model
  - No variables or registers
- A **stack** of values for intermediate results
  - Imagine a calculator that uses “reverse Polish” notation

Stack Machines

- A simple evaluation model
  - No variables or registers
- A **stack** of values for intermediate results
  - Imagine a calculator that uses “reverse Polish” notation
- Think of it this way:
  - I am going to explain how you’d build “Cool javac”; your compiler is supposed to be “Cool g++”.

Stack Machines

- A simple evaluation model
  - No variables or registers
- A **stack** of values for intermediate results
  - Imagine a calculator that uses “reverse Polish” notation
- Think of it this way:
  - I am going to explain how you’d build “Cool javac”; your compiler is supposed to be “Cool g++”.
    - “Cool javac” : Cool -> stack machine bytecode
      - A “JVM” is also necessary to interpret the bytecode!

Stack Machines

- A simple evaluation model
  - No variables or registers
- A **stack** of values for intermediate results
  - Imagine a calculator that uses “reverse Polish” notation
- Think of it this way:
  - I am going to explain how you’d build “Cool javac”; your compiler is supposed to be “Cool g++”.
    - “Cool javac” : Cool -> stack machine bytecode
      - A “JVM” is also necessary to interpret the bytecode!
    - “Cool g++” : Cool -> x86-64

Stack Machines: Example Program

- Consider two instructions:

Stack Machines: Example Program

- Consider two instructions:
  - `push i`
    - place the integer i on top of the stack

Stack Machines: Example 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

Stack Machines: Example 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 Machines: Example 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
```

Quick poll: how much does this have in common with the 280 project? (determines how fast I go)

Stack Machines: Example Program

- Each instruction:

Stack Machines: Example Program

- Each instruction:
  - Takes its operands from the top of the stack

Stack Machines: Example Program

- Each instruction:
  - Takes its operands from the top of the stack
  - Removes those operands from the stack

Stack Machines: Example Program

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

Stack Machines: Example Program

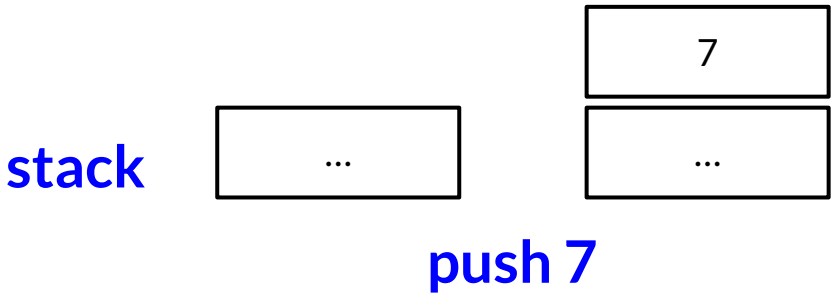
- Each instruction:
  - Takes its operands from the top of the stack
  - Removes those operands from the stack
  - Computes the required operation on them
  - Pushes the result on the stack

Stack Machines: Example Program

stack

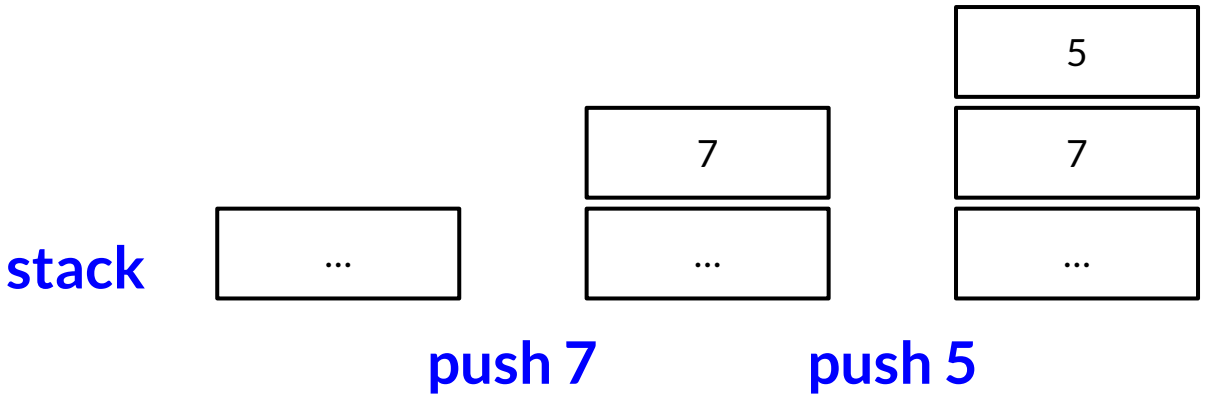
- Each instruction:
  - Takes its operands from the top of the stack
  - Removes those operands from the stack
  - Computes the required operation on them
  - Pushes the result on the stack

Stack Machines: Example Program



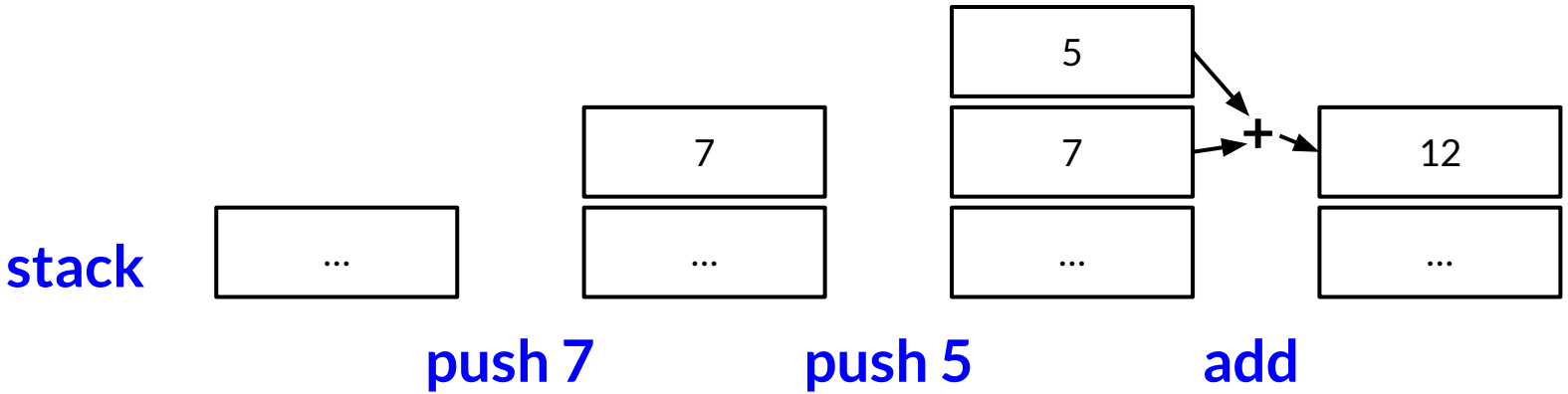
- Each instruction:
  - Takes its operands from the top of the stack
  - Removes those operands from the stack
  - Computes the required operation on them
  - Pushes the result on the stack

Stack Machines: Example Program



- 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

Stack Machines: Example Program



- 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

Stack Machines: Good For Compiler Writers

- Stack machines have a very nice property from our perspective as compiler writers:

Stack Machines: Good For Compiler Writers

- Stack machines have a very nice property from our perspective as compiler writers:
    - Each operation takes operands from the **same place** and puts results in the **same place**

Stack Machines: Good For Compiler Writers

- Stack machines have a very nice property from our perspective as compiler writers:
    - Each operation takes operands from the **same place** and puts results in the **same place**
- This means a **uniform** compilation scheme

Stack Machines: Good For Compiler Writers

- Stack machines have a very nice property from our perspective as compiler writers:
    - Each operation takes operands from the **same place** and puts results in the **same place**
- This means a **uniform** compilation scheme
    - We don't have to worry about where operands are ("they're on the stack")

Stack Machines: Good For Compiler Writers

- Stack machines have a very nice property from our perspective as compiler writers:
    - Each operation takes operands from the **same place** and puts results in the **same place**
- This means a **uniform** compilation scheme
    - We don't have to worry about where operands are ("they're on the stack")
- And therefore means we can write a **very simple compiler**

Stack Machines: Good For Compiler Writers

- Stack machines have a very nice property from our perspective as compiler writers:
    - Each operation takes operands from the **same place** and puts results in the **same place**
- This means a **uniform** compilation scheme
    - We don't have to worry about where operands are ("they're on the stack")
- And therefore means we can write a **very simple compiler**
    - Many/most compilers start by targeting a stack machine

Stack Machines: Good For Compiler Writers

- Stack machines have a very nice property from our perspective as compiler writers:
    - Each operation takes operands from the **same place** and puts results in the **same place**
- This means a **uniform** compilation scheme
    - We don't have to worry about where operands are ("they're on the stack")
- And therefore means we can write a **very simple compiler**
    - Many/most compilers start by targeting a stack machine
    - A good idea for you, too! (in PA3)

Stack Machines: Advantages

Stack Machines: Advantages

- Location of the operands is implicit
  - Always on the top of the stack

Stack Machines: Advantages

- Location of the operands is implicit
  - Always on the top of the stack
- No need to specify operands explicitly

Stack Machines: Advantages

- 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

Stack Machines: Advantages

- 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 r1, r2**"
  - Smaller encoding of instructions
  - More compact programs (great for network!)

Stack Machines: Advantages

- 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 r1, r2**"
  - Smaller encoding of instructions
  - More compact programs (great for network!)
- This is one reason why Java bytecode uses a stack evaluation model

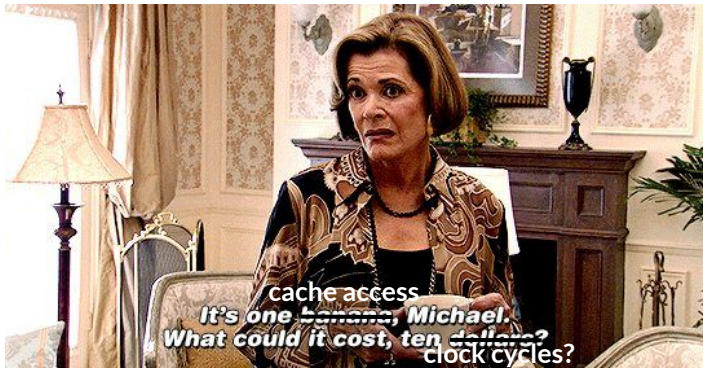
Stack Machines: A Big Disadvantage

Stack Machines: A Big Disadvantage

- The **add** instruction does 3 memory operations
  - Two reads and one write to the stack

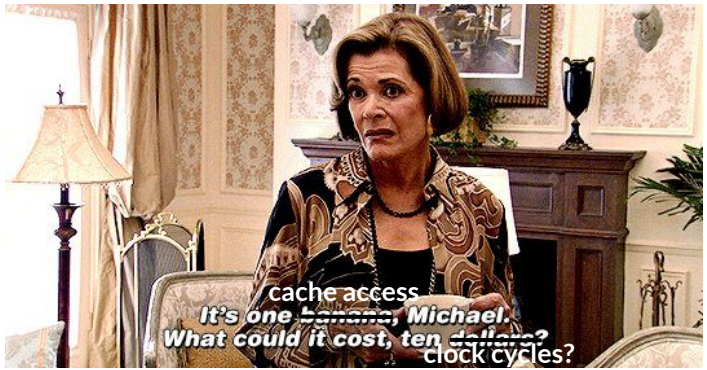
Stack Machines: A Big Disadvantage

- The **add** instruction does 3 memory operations
  - Two reads and one write to the stack
    - This is very slow!



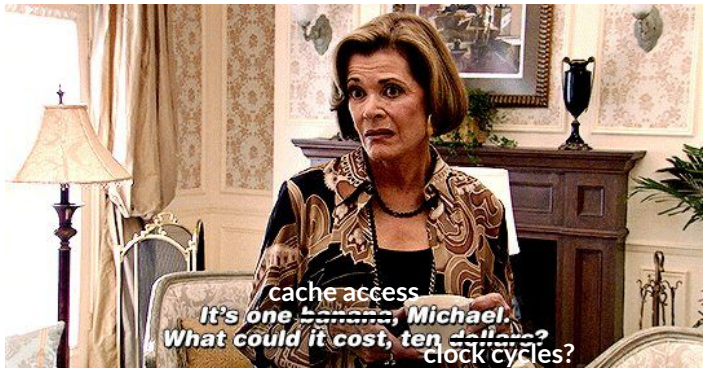
Stack Machines: A Big Disadvantage

- The **add** instruction does 3 memory operations
  - Two reads and one write to the stack
    - This is very slow!
  - The top of the stack is “frequently” accessed (*so much*)



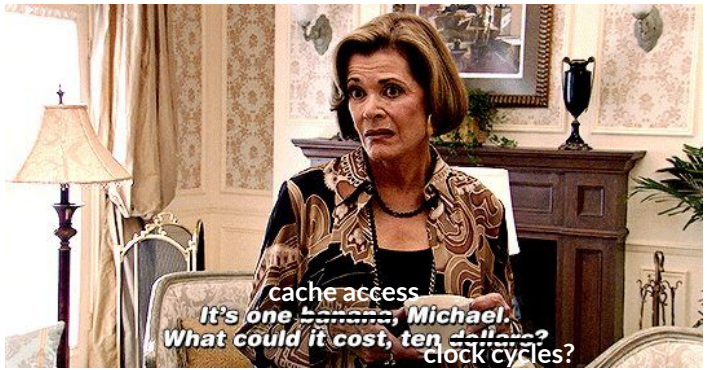
Stack Machines: A Big Disadvantage

- The **add** instruction does 3 memory operations
  - Two reads and one write to the stack
    - This is very slow!
  - The top of the stack is “frequently” accessed (*so much*)
- We can do one **simple optimization** to mitigate this disadvantage



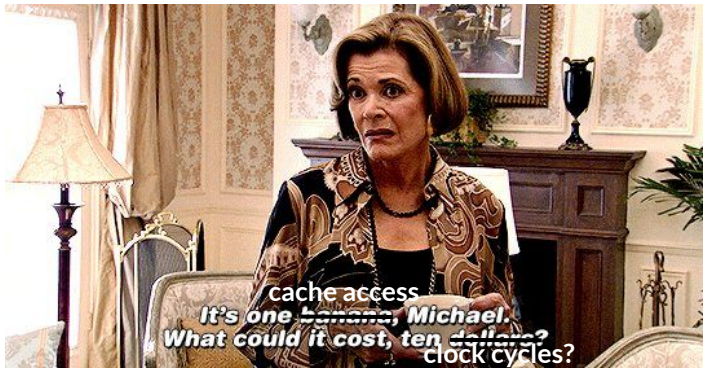
Stack Machines: A Big Disadvantage

- The **add** instruction does 3 memory operations
  - Two reads and one write to the stack
    - This is very slow!
  - The top of the stack is “frequently” accessed (*so much*)
- We can do one **simple optimization** to mitigate this disadvantage
  - Bonus pedagogical benefit: I can mention again that **you shouldn’t prematurely optimize** on PA3



Stack Machines: A Big Disadvantage

- The **add** instruction does 3 memory operations
  - Two reads and one write to the stack
    - This is very slow!
  - The top of the stack is “frequently” accessed (*so much*)
- We can do one **simple optimization** to mitigate this disadvantage
  - Bonus pedagogical benefit: I can mention again that **you shouldn’t prematurely optimize** on PA3
- Can anyone guess what it is?



Stack Machines: Accumulator Register

- Key insight: keep the top of the stack in a register (called the *accumulator*)

Stack Machines: Accumulator Register

- Key insight: keep the top of the stack in a register (called the ***accumulator***)
  - This should remind you of how **fold** works
    - It's almost like functional programming is relevant to the rest of the class…

Stack Machines: Accumulator Register

- Key insight: keep the top of the stack in a register (called the ***accumulator***)
  - This should remind you of how **fold** works
    - It's almost like functional programming is relevant to the rest of the class…
  - Register accesses are much faster

Stack Machines: Accumulator Register

- Key insight: keep the top of the stack in a register (called the *accumulator*)
  - This should remind you of how fold works
    - It's almost like functional programming is relevant to the rest of the class…
  - Register accesses are much faster
- The add instruction is now `acc <- acc + top_of_stack`
  - Only one memory operation, much faster!

Stack Machines: Accumulator Invariants

Stack Machines: Accumulator Invariants

- The result of computing an expression is **always** in the accumulator

Stack Machines: Accumulator Invariants

- The result of computing an expression is **always** in the accumulator
- For an operation $op(e_1, \ldots, e_n)$, **push** the accumulator on the stack after computing each of e_1, ..., e_{n-1}

Stack Machines: Accumulator Invariants

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

Stack Machines: Accumulator Invariants

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

Stack Machines: Accumulator Invariants

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

Stack Machines: Accumulator Invariants

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

Let's look at an example…

Stack Machines: Accumulator Example

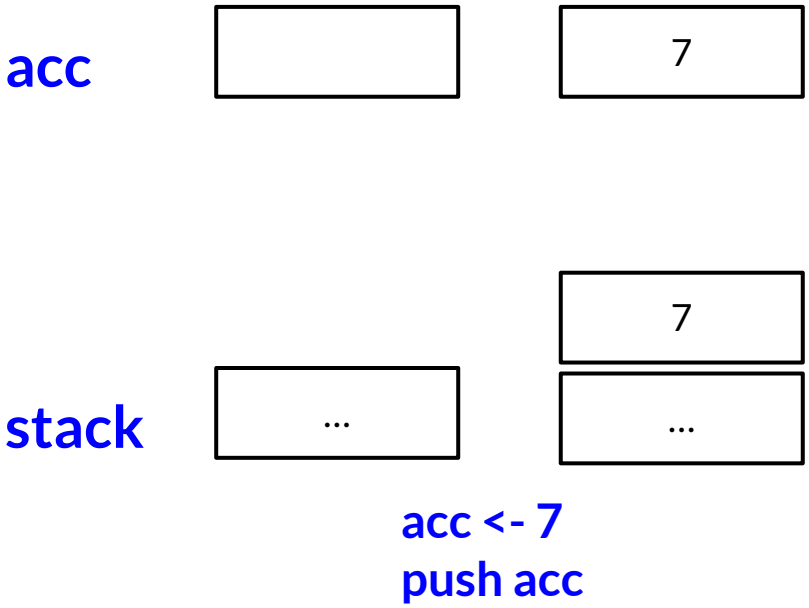
- Computing `7 + 5` with an accumulator:

acc 

stack 

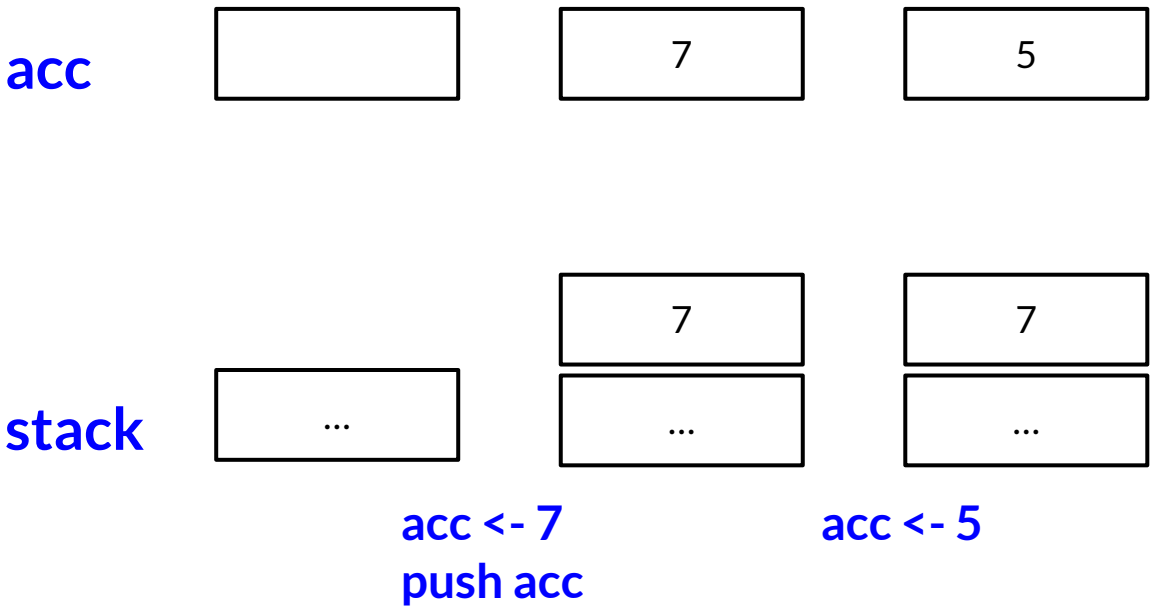
Stack Machines: Accumulator Example

- Computing 7 + 5 with an accumulator:



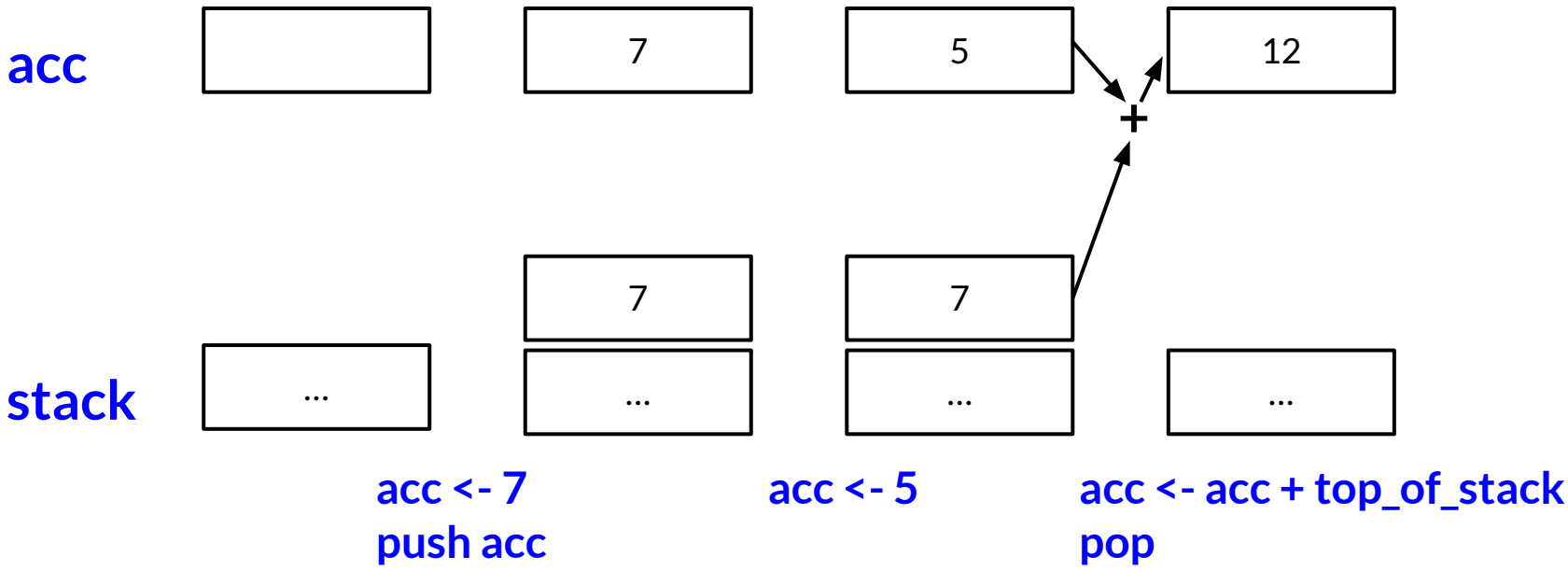
Stack Machines: Accumulator Example

- Computing 7 + 5 with an accumulator:



Stack Machines: Accumulator Example

- Computing 7 + 5 with an accumulator:



Stack Machines: Bigger Example: 3 + (7 + 5)

Code	Acc	Stack

Stack Machines: Bigger Example: 3 + (7 + 5)

Code	**Acc**	**Stack**
acc <- 3	3	<init>

Stack Machines: Bigger Example: 3 + (7 + 5)

Code	Acc	Stack
acc <- 3	3	<init>
push acc	3	3, <init>

Stack Machines: Bigger Example: 3 + (7 + 5)

Code	Acc	Stack
acc <- 3	3	<init>
push acc	3	3, <init>
acc <- 7	7	3, <init>

Stack Machines: 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>

Stack Machines: 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>

Stack Machines: 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>

Stack Machines: 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>

Stack Machines: 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>

Stack Machines: 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>

Stack Machines: Accumulator Invariants

- The result of computing an expression is **always** in the accumulator
- For an operation $op(e_1, \ldots, e_n)$, **push** the accumulator on the stack after computing each of e_1, ..., e_{n-1}
  - e_n's result is in the accumulator before op
  - After the operation **pop** n-1 values
- After computing an expression the stack is as before
  - It is **CRITICAL** that ***the stack is preserved across the evaluation of a subexpression***

Stack Machines: Accumulator Invariants

- The result of computing an expression is **always** in the accumulator
- For an operation $op(e_1, \ldots, e_n)$, **push** the accumulator on the stack after computing each of e_1, ..., e_{n-1}
  - e_n's result is in the accumulator before op
  - After the operation **pop** n-1 values
- After computing an expression the stack is as before
  - It is **CRITICAL** that ***the stack is preserved across the evaluation of a subexpression***
    - Stack *before* evaluating `7 + 5` is `3,<init>`

Stack Machines: Accumulator Invariants

- The result of computing an expression is **always** in the accumulator
- For an operation $op(e_1, \ldots, e_n)$, **push** the accumulator on the stack after computing each of e_1, ..., e_{n-1}
  - e_n's result is in the accumulator before op
  - After the operation **pop** n-1 values
- After computing an expression the stack is as before
  - It is **CRITICAL** that ***the stack is preserved across the evaluation of a subexpression***
    - Stack *before* evaluating `7 + 5` is `3,<init>`
    - Stack *after* evaluating `7 + 5` is *also* `3,<init>`

From Stack Machines to RISC

- The “compiler” we will outline in this two-lecture series will **generate code for a stack machine** with an accumulator

From Stack Machines to RISC

- The “compiler” we will outline in this two-lecture series will **generate code for a stack machine** with an accumulator
  - You might want to run the resulting code on a processor

From Stack Machines to RISC

- The “compiler” we will outline in this two-lecture series will **generate code for a stack machine** with an accumulator
  - You might want to run the resulting code on a processor
- To do so, we'll implement stack machine instructions using **Cool-ASM** instructions and registers

From Stack Machines to RISC

- The “compiler” we will outline in this two-lecture series will **generate code for a stack machine** with an accumulator
  - You might want to run the resulting code on a processor
- To do so, we'll implement stack machine instructions using **Cool-ASM** instructions and registers
  - Cool-ASM is a bytecode format described in the CRM
    - I will assume access to a Cool-ASM interpreter

From Stack Machines to RISC

- The “compiler” we will outline in this two-lecture series will **generate code for a stack machine** with an accumulator
  - You might want to run the resulting code on a processor
- To do so, we'll implement stack machine instructions using **Cool-ASM** instructions and registers
  - Cool-ASM is a bytecode format described in the CRM
    - I will assume access to a Cool-ASM interpreter
  - Fun PA3 activity for you: map this Cool-ASM to x86-64
    - effectively, this would be using Cool-ASM as an IR!

From Stack Machines to RISC

- The “compiler” we will outline in this two-lecture series will **generate code for a stack machine** with an accumulator
  - You might want to run the resulting code on a processor
- To do so, we'll implement stack machine instructions using **Cool-ASM** instructions and registers
  - Cool-ASM is a bytecode format described in the CRM
    - I will assume access to a Cool-ASM interpreter
  - Fun PA3 activity for you: map this Cool-ASM to x86-64
    - effectively, this would be using Cool-ASM as an IR!
    - you are welcome to do so in your compiler if you want
      - would it come before or after three-address code?

Cool-ASM in Brief

- Cool-ASM is a RISC-style assembly language

Cool-ASM in Brief

- Cool-ASM is a RISC-style assembly language
  - An ***assembly language*** is an untyped, unsafe, low-level, fast programming language with few-to-no primitives.

Cool-ASM in Brief

- Cool-ASM is a RISC-style assembly language
  - An ***assembly language*** is an untyped, unsafe, low-level, fast programming language with few-to-no primitives.
- A ***register*** is a fast-access untyped global variable shared by the entire assembly program.

Cool-ASM in Brief

- Cool-ASM is a RISC-style assembly language
  - An ***assembly language*** is an untyped, unsafe, low-level, fast programming language with few-to-no primitives.
- A ***register*** is a fast-access untyped global variable shared by the entire assembly program.
  - Cool-ASM: 8 general registers and 3 special ones (stack pointer, frame pointer, return address)

Cool-ASM in Brief

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

Cool-ASM in Brief

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

Cool-ASM in Brief: Sample Instructions

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

Cool-ASM in Brief: Sample Instructions

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

Details of Cool-ASM don't matter much; you're not required to implement this. See the CRM if you want full details.

Simulating a Stack Machine….

- The accumulator is kept in register r1

Simulating a Stack Machine….

- The accumulator is kept in register r1
  - This is just a **convention**. You could pick r2.

Simulating a Stack Machine….

- The accumulator is kept in register r1
  - This is just a **convention**. You could pick r2.
- The stack is kept in memory

Simulating a Stack Machine….

- The accumulator is kept in register r1
  - This is just a **convention**. You could pick r2.
- The stack is kept in memory
- The stack grows **towards lower addresses**
  - Standard architecture convention (MIPS)

Simulating a Stack Machine….

- The accumulator is kept in register r1
  - This is just a **convention**. You could pick r2.
- The stack is kept in memory
- The stack grows **towards lower addresses**
  - Standard architecture convention (MIPS)
- The address of the next unused location on the stack is kept in register sp

Simulating a Stack Machine….

- The accumulator is kept in register r1
  - This is just a **convention**. You could pick r2.
- The stack is kept in memory
- The stack grows **towards lower addresses**
  - Standard architecture convention (MIPS)
- The address of the next unused location on the stack is kept in register sp
  - The top of the stack is always at address sp + 1

Simulating a Stack Machine….

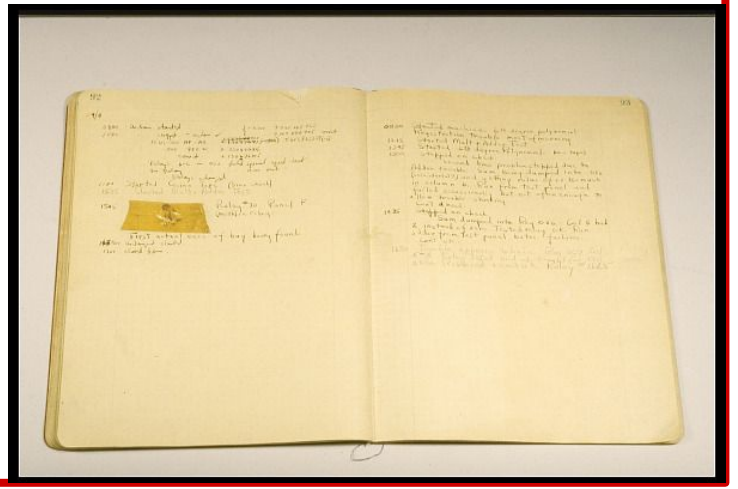
- The accumulator is kept in register r1
  - This is just a **convention**. You could pick r2.
- The stack is kept in memory
- The stack grows **towards lower addresses**
  - Standard architecture convention (MIPS)
- The address of the next unused location on the stack is kept in register sp
  - The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

Simulating a Stack Machine….

- The accumulator is kept in register r1
  - This is just a **convention**. You could pick r2.
- The stack is kept in memory
- The stack grows **towards lower addresses**
  - Standard architecture convention (MIPS)
- The address of the next unused location on the stack is kept in register sp
  - The top of the stack is always at address sp + 1
- Cool-ASM “Word Size” = 1 = number of memory cells taken up by one integer/pointer/string
  - Note **intentional simplification** vs x86! This is what makes a good IR

Trivia Break: Computer Science

This US Navy Rear Admiral is credited with inventing one of the first compilers for a computer programming language, as well as popularizing machine-independent languages (particularly COBOL). The term “debugging” was also popularized when a logbook that is sometimes erroneously credited to this person (who was one of the associated project’s leaders) physically recorded a moth that had gotten into the relays:



Cool Assembly Example

Stack-machine code for `7 + 5`:

Equivalent Cool-ASM:

Cool Assembly Example

Stack-machine code for 7 + 5:

```
acc <- 7
```

Equivalent Cool-ASM:

```
li r1 7
```

Cool Assembly Example

Stack-machine code for 7 + 5:

```
acc <- 7
push acc
```

Equivalent Cool-ASM:

```
li r1 7
sw sp[0] <- r1
sub sp <- sp 1
```

Cool Assembly Example

Stack-machine code for 7 + 5:

```
acc <- 7
push acc

acc <- 5
```

Equivalent Cool-ASM:

```
li r1 7
sw sp[0] <- r1
sub sp <- sp 1
li r1 5
```

Cool Assembly Example

Stack-machine code for 7 + 5:

```
acc <- 7
push acc

acc <- 5
acc <- acc + top_of_stack
```

Equivalent Cool-ASM:

```
li r1 7
sw sp[0] <- r1
sub sp <- sp 1
li r1 5
lw r2 <- sp[1]
add r1 <- r1 r2
```

Cool Assembly Example

Stack-machine code for 7 + 5:

```
acc <- 7
push acc

acc <- 5
acc <- acc + top_of_stack

pop
```

Equivalent Cool-ASM:

```
li r1 7
sw sp[0] <- r1
sub sp <- sp 1
li r1 5
lw r2 <- sp[1]
add r1 <- r1 r2
add sp <- sp 1
```

Cool Assembly Example

Stack-machine code for 7 + 5:

```
acc <- 7
push acc

acc <- 5
acc <- acc + top_of_stack

pop
```

Equivalent Cool-ASM:

```
li r1 7
sw sp[0] <- r1
sub sp <- sp 1
li r1 5
lw r2 <- sp[1]
add r1 <- r1 r2
add sp <- sp 1
```

We now generalize this to a simple language…

Stack Instructions

- We have these Cool-ASM instructions:

Stack Instructions

- We have these Cool-ASM instructions:

push r*X*

st sp[0] <- r*X*
sub sp <- sp 1

Stack Instructions

- We have these Cool-ASM instructions:

push r*X*	st sp[0] <- r*X* sub sp <- sp 1
pop r*X*	ld r*X* <- sp[1] add sp <- sp 1

Stack Instructions

- We have these Cool-ASM instructions:

push r*X*	st sp[0] <- r*X* sub sp <- sp 1
pop r*X*	ld r*X* <- sp[1] add sp <- sp 1
r*X* <- top	ld r*X* <- sp[1]

A Small Language

- We will consider a source language with integers and integer operations (only, for now)

A Small Language

- We will consider a source language with integers and integer operations (only, for now)
- Full grammar:

P -> D ; P | D
D -> def *id*(ARGS) = E
ARGS -> *id*, ARGS | *id*
E -> *int* | *id* | if E_1 = E_2 then E_3 else E_4
| E_1 + E_2 | E_1 - E_2 | *id*(E_1, …, E_n)

Reminder of how to read this:

- capital letters are non-terminals
- italics = lexemes
- "|" separates options

A Small Language (continued)

- The first function definition f is the “main” routine

A Small Language (continued)

- The first function definition f is the “main” routine
- Running the program on input i means computing f(i)

A Small Language (continued)

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

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

Code Generation Strategy

Code Generation Strategy

- For each expression e we generate Cool- ASM code that:

Code Generation Strategy

- For each expression `e` we generate Cool- ASM code that:
  - Computes the value of `e` in `r1` (our accumulator)

Code Generation Strategy

- For each expression e we generate Cool- ASM code that:
  - Computes the value of e in r1 (our accumulator)
  - Preserves sp and the contents of the stack

Code Generation Strategy

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

Code Generation Strategy

- For each expression e we generate Cool- ASM code that:
  - Computes the value of e in r1 (our accumulator)
  - Preserves sp and the contents of the stack
- We define a ***code generation function*** cgen(e) whose result is the code generated for e
- Like type rules, our code generation function is ***syntax-directed***

Code Generation Strategy

- For each expression e we generate Cool- ASM code that:
  - Computes the value of e in r1 (our accumulator)
  - Preserves sp and the contents of the stack
- We define a ***code generation function*** cgen(e) whose result is the code generated for e
- Like type rules, our code generation function is ***syntax-directed***
  - in other words, it is a **big case statement**, based on the structure of the input

Code Generation Strategy

- For each expression e we generate Cool- ASM code that:
  - Computes the value of e in r1 (our accumulator)
  - Preserves sp and the contents of the stack
- We define a ***code generation function*** cgen(e) whose result is the code generated for e
- Like type rules, our code generation function is ***syntax-directed***
  - in other words, it is a **big case statement**, based on the structure of the input
    - we will have one code generation rule for constants, one for addition, one for subtraction, etc.
      - one for each kind of expression!

Code Generation: Constants

Code Generation: Constants

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

  cgen(123) = li r1 123

Code Generation: Constants

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

  cgen(123) = li r1 123

- Note that this also preserves the stack, as required

Code Generation: Addition

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

Code Generation: Addition

$cgen(e_1 + e_2)$ =

$cgen(e_1)$

Code Generation: Addition

$cgen(e_1 + e_2)$ =

    $cgen(e_1)$
    push r1

Code Generation: Addition

```
cgen(e1 + e2) =
    cgen(e1)
    push r1
    cgen(e2) ;; e2 now in r1
```

Code Generation: Addition

```
cgen(e1 + e2) =
    cgen(e1)
    push r1
    cgen(e2) ;; e2 now in r1
    pop t1
```

Code Generation: Addition

cgen(e_1 + e_2) =
    cgen(e_1)
    push r1
    cgen(e_2) ;; *e_2 now in r1*
    pop t1

t1 is some unused "temporary" register

Code Generation: Addition

```
cgen(e1 + e2) =
    cgen(e1)
    push r1
    cgen(e2) ;; e2 now in r1
    pop t1
    add r1 <- t1 r1
```

Code Generation: Addition

```
cgen(e1 + e2) =
    cgen(e1)
    push r1
    cgen(e2) ;; e2 now in r1
    pop t1
    add r1 <- t1 r1
```

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

Code Generation Mistake: Addition Alternative

- Unsafe optimization: put the result of e_1 directly in register t1?

Code Generation Mistake: Addition Alternative

- Unsafe optimization: put the result of e_1 directly in register t1?

```
cgen(e1 + e2) =
    cgen(e1)
    mov t1 <- r1
    cgen(e2) ;; e2 now in r1
    add r1 <- t1 r1
```

Code Generation Mistake: Addition Alternative

- Unsafe optimization: put the result of e_1 directly in register t1?

```
cgen(e1 + e2) =
    cgen(e1)
    mov t1 <- r1
    cgen(e2) ;; e2 now in r1
    add r1 <- t1 r1
```

- To see why this is incorrect, try to generate code for : 3 + (7 + 5)

Code Generation Notes

Code Generation Notes

- The code that we generated for addition is a template with “holes” for code for evaluating e_1 and e_2

Code Generation Notes

- The code that we generated for addition is a template with “holes” for code for evaluating e_1 and e_2
- Stack-machine code generation is **recursive**

Code Generation Notes

- The code that we generated for addition 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
    - with a bit of code to actually do the addition

Code Generation Notes

- The code that we generated for addition 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
    - with a bit of code to actually do the addition
- Implication: code generation can be written as a **recursive-descent tree walk** of the AST
  - At least for expressions

Code Generation: Subtraction

$cgen(e_1 - e_2)$ =

Code Generation: Subtraction

```
cgen(e1 - e2) =
    cgen(e1)
    push r1
    cgen(e2) ;; e2 now in r1
    pop t1
    sub r1 <- t1 r1
```

Code Generation: Subtraction

```
cgen(e1 - e2) =
    cgen(e1)
    push r1
    cgen(e2) ;; e2 now in r1
    pop t1
    sub r1 <- t1 r1
```

- Almost identical to addition!
  - only difference is the **sub** instruction, which does what you'd expect (subtraction)

Code Generation: If

- To generate code for `if`, we need control flow instructions

Code Generation: If

- To generate code for `if`, we need control flow instructions
- Cool-ASM has two that will be useful:

Code Generation: If

- To generate code for `if`, we need control flow instructions
- Cool-ASM has two that will be useful:

  beq r1 r2 label

  conditional branch to label if r1 = r2

Code Generation: If

- To generate code for `if`, we need control flow instructions
- Cool-ASM has two that will be useful:

beq r1 r2 label

conditional branch to label if r1 = r2

jmp label

unconditional jump to label

Code Generation: If

- To generate code for `if`, we need control flow instructions
- Cool-ASM has two that will be useful:

beq r1 r2 label

conditional branch to label if r1 = r2

jmp label

unconditional jump to label

Note the similarity to what x86 provides! How easy would it be to convert these Cool-ASM instructions into x86?

Code Generation: If

cgen(if e1 = e2 then e3 else e4) =

Code Generation: If

cgen(if e1 = e2 then e3 else e4) =

cgen(e1)

Code Generation: If

cgen(if e1 = e2 then e3 else e4) =

    cgen(e1)

    push r1

Code Generation: If

cgen(if e1 = e2 then e3 else e4) =

```
    cgen(e1)
    push r1
    cgen(e2)
```

Code Generation: If

cgen(if e1 = e2 then e3 else e4) =

```
    cgen(e1)
    push r1
    cgen(e2)
    pop t1
```

Code Generation: If

```
cgen(if e1 = e2 then e3 else e4) =
        cgen(e1)
        push r1
        cgen(e2)
        pop t1
        beq r1 t1 true_branch ;; else fall through
```

Code Generation: If

```
cgen(if e1 = e2 then e3 else e4) =
        cgen(e1)
        push r1
        cgen(e2)
        pop t1
        beq r1 t1 true_branch ;; else fall through
        cgen(e4)
```

Code Generation: If

```
cgen(if e1 = e2 then e3 else e4) =
        cgen(e1)
        push r1
        cgen(e2)
        pop t1
        beq r1 t1 true_branch ;; else fall through
        cgen(e4)
        jmp end_if
```

Code Generation: If

```
cgen(if e1 = e2 then e3 else e4) =
        cgen(e1)
        push r1
        cgen(e2)
        pop t1
        beq r1 t1 true_branch ;; else fall through
        cgen(e4)
        jmp end_if
    true_branch:
```

Code Generation: If

```
cgen(if e1 = e2 then e3 else e4) =
        cgen(e1)
        push r1
        cgen(e2)
        pop t1
        beq r1 t1 true_branch ;; else fall through
        cgen(e4)
        jmp end_if
    true_branch:
        cgen(e3)
```

Code Generation: If

```
cgen(if e1 = e2 then e3 else e4) =
        cgen(e1)
        push r1
        cgen(e2)
        pop t1
        beq r1 t1 true_branch ;; else fall through
        cgen(e4)
        jmp end_if
    true_branch:
        cgen(e3)
    end_if:
```

Trivia Break: Art History

This style of visual arts, architecture, and product design first appeared in Paris in the early 1910s and flourished in the US and Europe during the 1920s. It is characterized by rare and expensive materials, such as ebony and ivory, exquisite craftsmanship, and the inclusion of “modern” materials like chrome plating, stainless steel, and plastic. In New York City, the Empire State Building, Chrysler Building, and other buildings from the 1920s and 1930s are monuments to the style.

Trivia Break: Geography

This lake in Russia is the world's seventh-largest by surface area (it is just larger than Belgium). However, unlike most other freshwater lakes in the world, it is a *rift lake*: a geologically active rift in the Earth's crust at the bottom of the lake is being pulled apart at a rate of about 4mm per year. This has two important implications for this lake: it is both the oldest (~25-30 million years old) and deepest (avg. 744m, deepest 1,642m) lake in the world. Because of its depth, it contains about 22% of the world's freshwater.

Activation Records

- Recall that an ***activation record*** (or ***stack frame***) stores calling context information on the stack during a function call.

Activation Records

- Recall that an ***activation record*** (or ***stack frame***) stores calling context information on the stack during a function call.
  - Code for function calls/definitions depends on the layout of the activation record

Activation Records

- Recall that an ***activation record*** (or ***stack frame***) stores calling context information on the stack during a function call.
  - Code for function calls/definitions depends on the layout of the activation record
- A very simple AR suffices for this language:

Activation Records

- Recall that an ***activation record*** (or ***stack frame***) stores calling context information on the stack during a function call.
  - Code for function calls/definitions depends on the layout of the activation record
- A very simple AR suffices for this language:
  - The result is always in the accumulator
    - Thus, no need to store the result in the AR

Activation Records

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

Calling Convention

- The ***calling convention*** (or ***stack discipline***) guarantees that on function exit sp is the same as it was on entry

Calling Convention

- The ***calling convention*** (or ***stack discipline***) guarantees that on function exit sp is the same as it was on entry
  - So, no need to save sp explicitly

Calling Convention

- The ***calling convention*** (or ***stack discipline***) guarantees that on function exit sp is the same as it was on entry
  - So, no need to save sp explicitly
  - Our simple AR maintains stack discipline (why?)

Calling Convention

- The ***calling convention*** (or ***stack discipline***) guarantees that on function exit sp is the same as it was on entry
  - So, no need to save sp explicitly
  - Our simple AR maintains stack discipline (why?)
- We do need the return address

Calling Convention

- The ***calling convention*** (or ***stack discipline***) guarantees that on function exit sp is the same as it was on entry
  - So, no need to save sp explicitly
  - Our simple AR maintains stack discipline (why?)
- We do need the return address
- Also, it's handy to have a pointer to the start of the current activation

Calling Convention

- The ***calling convention*** (or ***stack discipline***) guarantees that on function exit sp is the same as it was on entry
  - So, no need to save sp explicitly
  - Our simple AR maintains stack discipline (why?)
- We do need the return address
- Also, it's handy to have a pointer to the start of the current activation
  - This pointer lives in register fp ("frame pointer")

Calling Convention

- The ***calling convention*** (or ***stack discipline***) guarantees that on function exit sp is the same as it was on entry
  - So, no need to save sp explicitly
  - Our simple AR maintains stack discipline (why?)
- We do need the return address
- Also, it's handy to have a pointer to the start of the current activation
  - This pointer lives in register fp ("frame pointer")
  - Reason for frame pointer will be clear shortly

The Activation Record (for our little language)

- Summary: for this language, an AR with

suffices.

The Activation Record (for our little language)

- Summary: for this language, an AR with
  1. the caller's frame pointer

suffices.

The Activation Record (for our little language)

- Summary: for this language, an AR with
  1. the caller's frame pointer
  2. the actual parameters, and

suffices.

The Activation Record (for our little language)

- Summary: for this language, an AR with
    1. the caller's frame pointer
    2. the actual parameters, and
    3. the return address

suffices.

The Activation Record (for our little language)

- Summary: for this language, an AR with
  1. the caller's frame pointer
  2. the actual parameters, and
  3. the return address

suffices.

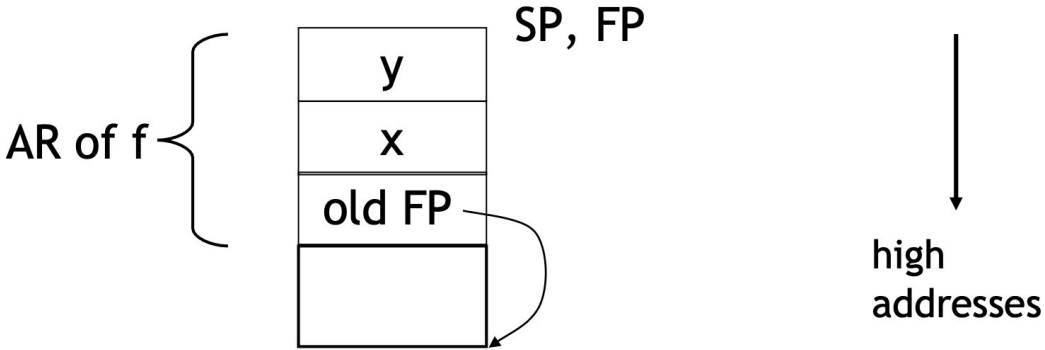
- Visual learners: consider a call to f(x,y). The AR will be:

The Activation Record (for our little language)

- Summary: for this language, an AR with
  1. the caller's frame pointer
  2. the actual parameters, and
  3. the return address

suffices.

- Visual learners: consider a call to f(x,y). The AR will be:



Code Generation: Function Calls

Code Generation: Function Calls

- The ***calling sequence*** is the instructions (of both caller and callee) to set up a function invocation

Code Generation: Function Calls

- The ***calling sequence*** is the instructions (of both caller and callee) to set up a function invocation
- Requires one new instruction: **call label**

Code Generation: Function Calls

- The ***calling sequence*** is the instructions (of both caller and callee) to set up a function invocation
- Requires one new instruction: **call label**
  - Jump to label, save address of next instruction in the special-purpose register ra
    - On other architectures (like x86!) the return address is stored on the stack by the “call” instruction
    - (This is also called “***branch and link***”.)

Code Generation: Function Calls

cgen(f(e_1,...,e_n)) =

Code Generation: Function Calls

cgen(f(e_1,...,e_n)) =
push fp

- The caller saves its value of the frame pointer

Code Generation: Function Calls

```
cgen(f(e1,...,en)) =
    push fp
    cgen(e1)
    push r1
```

- The caller saves its value of the frame pointer
- Then it saves the actual arguments in order

Code Generation: Function Calls

```
cgen(f(e1,...,en)) =
    push fp
    cgen(e1)
    push r1
    ...
```

- The caller saves its value of the frame pointer
- Then it saves the actual arguments in order

Code Generation: Function Calls

cgen(f(e_1,...,e_n)) =

```
    push fp
    cgen(e1)
    push r1
    ...
    cgen(en)
    push r1
```

- The caller saves its value of the frame pointer
- Then it saves the actual arguments in order

Code Generation: Function Calls

```
cgen(f(e1,...,en)) =
    push fp
    cgen(e1)
    push r1
    ...
    cgen(en)
    push r1
    call f_entry
```

- The caller saves its value of the frame pointer
- Then it saves the actual arguments in order
- The caller saves the return address in register ra (via the call instruction)
- The AR so far is n+1 bytes long

Code Generation: Function Calls

```
cgen(f(e1,...,en)) =
    push fp
    cgen(e1)
    push r1
    ...
    cgen(en)
    push r1
    call f_entry
    pop fp
```

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

Code Generation: Function Definition

cgen(def f(x_1,...,x_n) = e) =

Code Generation: Function Definition

cgen(def f(x_1,...,x_n) = e) =

f_entry:

Code Generation: Function Definition

cgen(def f(x_1,...,x_n) = e) =

f_entry:

mov fp<-sp

Code Generation: Function Definition

cgen(def f(x_1,...,x_n) = e) =

f_entry:

mov fp<-sp

push ra

Code Generation: Function Definition

cgen(def f(x_1,...,x_n) = e) =

f_entry:

mov fp<-sp

push ra

cgen(e)

Code Generation: Function Definition

cgen(def f(x_1,...,x_n) = e) =

f_entry:

mov fp<-sp

push ra

cgen(e)

ra <- top

Code Generation: Function Definition

cgen(def f(x_1,...,x_n) = e) =

f_entry:

mov fp<-sp

push ra

cgen(e)

ra <- top

add sp <- sp z

Code Generation: Function Definition

cgen(def f(x_1,...,x_n) = e) =

f_entry:

mov fp<-sp

push ra

cgen(e)

ra <- top

add sp <- sp z

return

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

Code Generation: Function Definition

cgen(def f(x_1,...,x_n) = e) =

***f_entry*:**

mov fp<-sp

push ra

cgen(e)

ra <- top

add sp <- sp z

return

- New instruction: **return**
  - Jump to address in register ra
- Note: the frame pointer points to the top, not bottom of the frame

Code Generation: Function Definition

cgen(def f(x_1,...,x_n) = e) =

***f_entry*:**

mov fp<-sp

push ra

cgen(e)

ra <- top

add sp <- sp z

return

- New instruction: **return**
  - Jump to address in register ra
- Note: the frame pointer points to the top, not bottom of the frame
- The callee pops the return address, the actual arguments and the saved value of the frame pointer

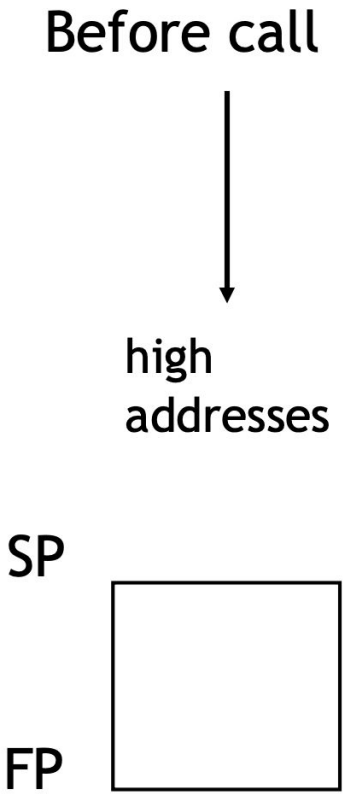
Code Generation: Function Definition

cgen(def f(x_1,...,x_n) = e) =
  ***f_entry*:**
    mov fp<-sp
    push ra
    cgen(e)
    ra <- top
    add sp <- sp z
    return

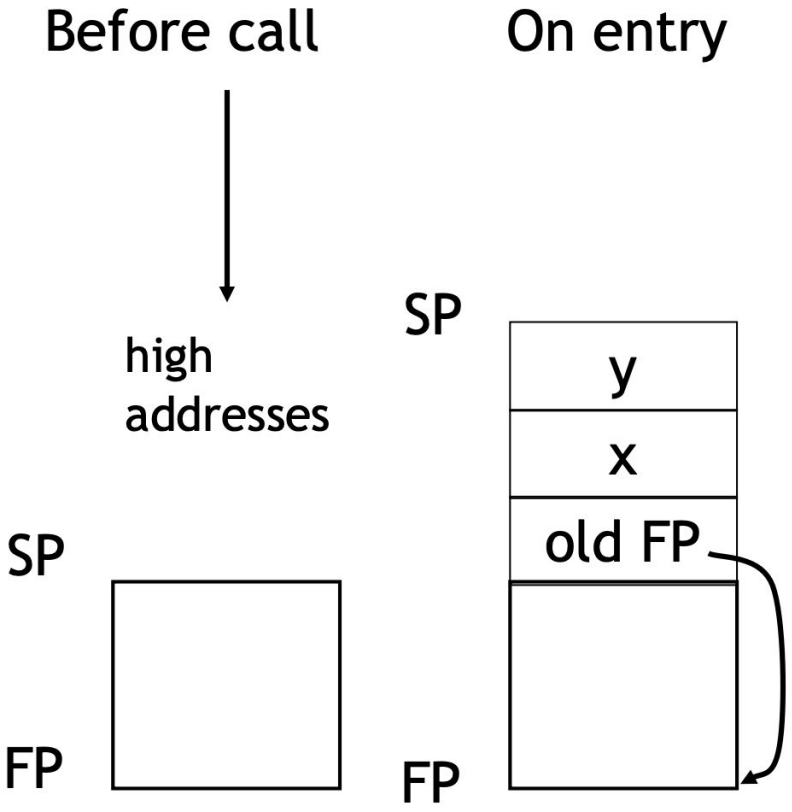
- New instruction: **return**
  - Jump to address in register ra
- Note: the frame pointer points to the top, not bottom of the frame
- The callee pops the return address, the actual arguments and the saved value of the frame pointer
- z = n + 2 (so far)

Calling Sequence for f(x, y)

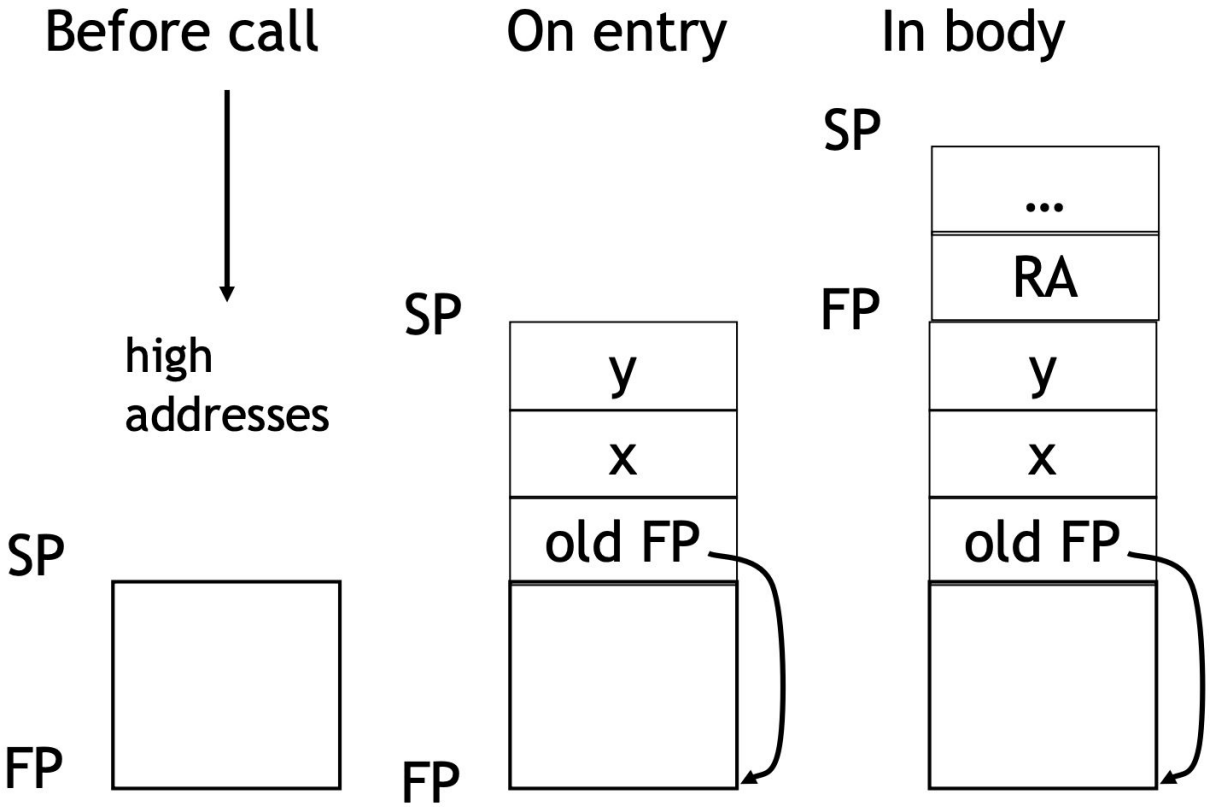
Calling Sequence for f(x, y)



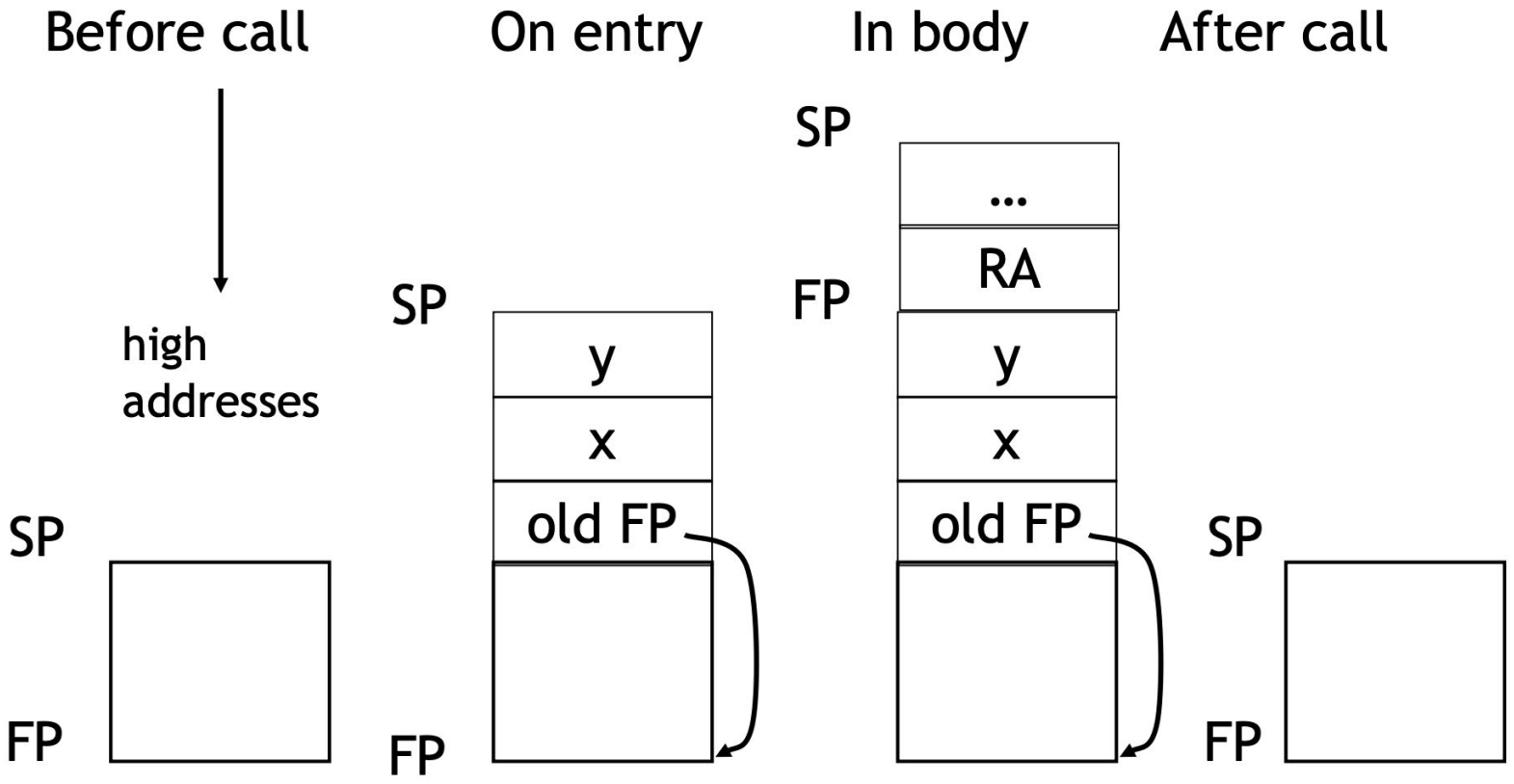
Calling Sequence for f(x, y)



Calling Sequence for f(x, y)



Calling Sequence for f(x, y)



Code Generation: Variables

- **Variable references** are the last construct

Code Generation: Variables

- **Variable references** are the last construct
- The “variables” of a function are just its **parameters**

Code Generation: Variables

- **Variable references** are the last construct
- The “variables” of a function are just its **parameters**
  - They are all in the AR

Code Generation: Variables

- **Variable references** are the last construct
- The “variables” of a function are just its **parameters**
  - They are all in the AR
  - Pushed by the caller

Code Generation: Variables

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

Code Generation: Variables

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

Code Generation: Variables

- **Variable references** are the last construct
- The “variables” of a function are just its **parameters**
  - They are all in the AR
  - Pushed by the caller
- Problem: because the stack grows when intermediate results are saved, the variables are **not at a fixed offset** from sp
  - Challenge question: what are they at a fixed offset from?
  - Answer: the **frame pointer**

Code Generation: Variables

- **Variable references** are the last construct
- The “variables” of a function are just its **parameters**
  - They are all in the AR
  - Pushed by the caller
- Problem: because the stack grows when intermediate results are saved, the variables are **not at a fixed offset** from sp
  - Challenge question: what are they at a fixed offset from?
  - Answer: the **frame pointer**
    - Always points to the return address on the stack
      - = the value of sp on function entry

Code Generation: Variables

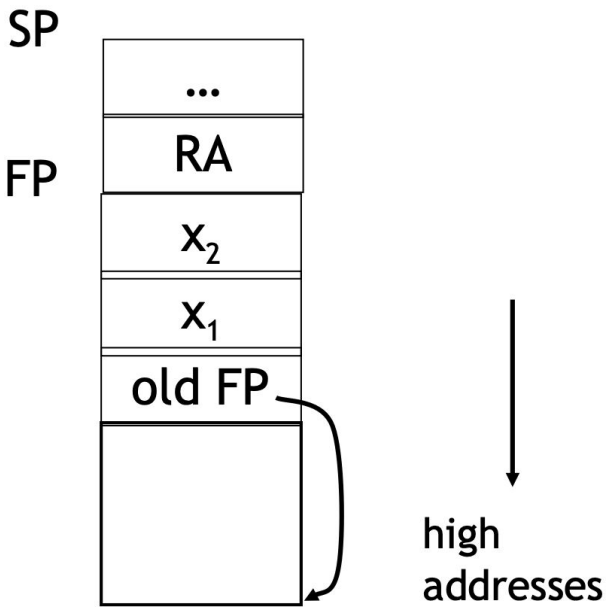
- **Variable references** are the last construct
- The “variables” of a function are just its **parameters**
  - They are all in the AR
  - Pushed by the caller
- Problem: because the stack grows when intermediate results are saved, the variables are **not at a fixed offset** from sp
  - Challenge question: what are they at a fixed offset from?
  - Answer: the **frame pointer**
    - Always points to the return address on the stack
      - = the value of sp on function entry
    - It doesn’t move => args on the stack are at a fixed offset

Code Generation: Variables

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

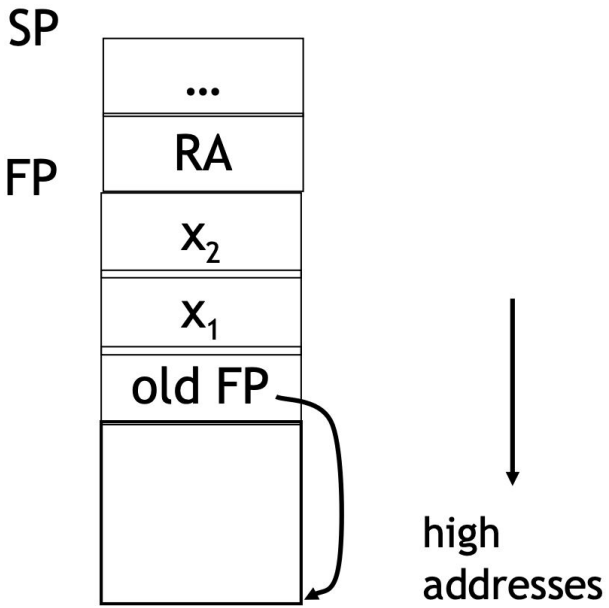
Code Generation: Variables

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



Code Generation: Variables

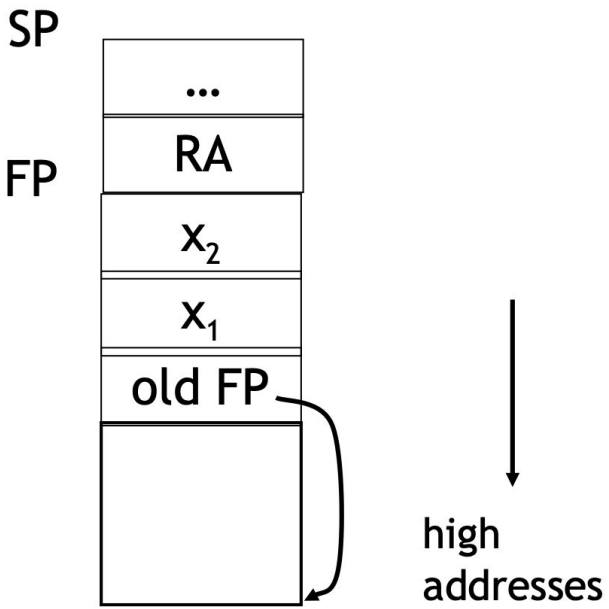
- Example: For a function def f(x_1, x_2) = e the activation and frame pointer are set up as follows:



- x_1 (first parameter) is at **fp + 2**

Code Generation: Variables

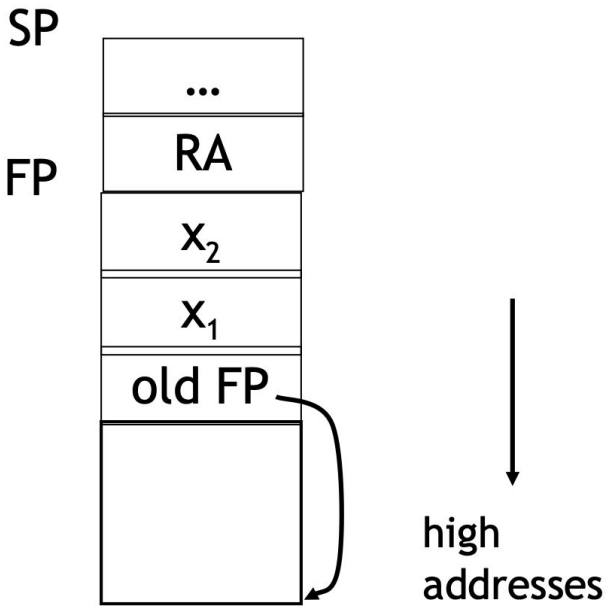
- Example: For a function def f(x_1, x_2) = e the activation and frame pointer are set up as follows:



- x_1 (first parameter) is at **fp + 2**
- x_2 (second parameter) is at **fp + 1**

Code Generation: Variables

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

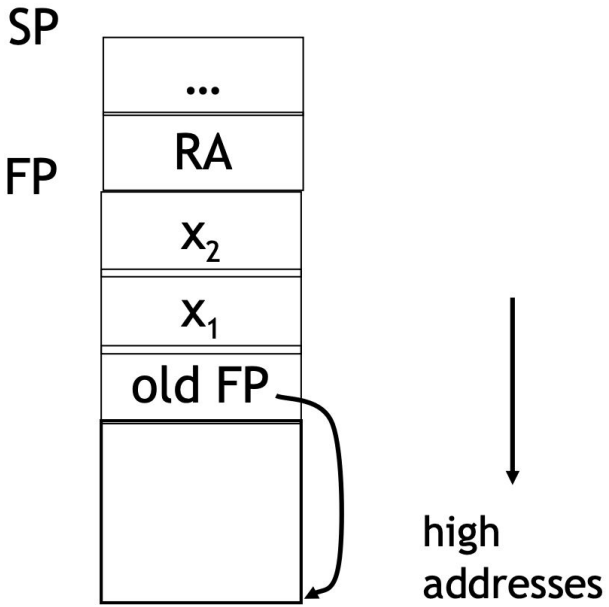


- x_1 (first parameter) is at **fp + 2**
- x_2 (second parameter) is at **fp + 1**
- Thus:

cgen(x_i) =
ld r1 <- fp[z]

Code Generation: Variables

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



- x_1 (first parameter) is at **fp + 2**
- x_2 (second parameter) is at **fp + 1**
- Thus:

cgen(x_i) =
ld r1 <- fp[z]

- where $z \approx n+1 - i$

Summary

Summary

- The activation record must be **designed together** with the code generator

Summary

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

Summary

- The activation record must be **designed together** with the code generator
- Code generation can be done by **recursive traversal** of the AST
- As you write your compiler, we recommend starting with a **stack machine** (simpler!)

Summary

- The activation record must be **designed together** with the code generator
- Code generation can be done by **recursive traversal** of the AST
- As you write your compiler, we recommend starting with a **stack machine** (simpler!)
  - `./cool --asm` generates Cool-ASM stack machine code for Cool
    - use this to help you with PA3!

Summary

- The activation record must be **designed together** with the code generator
- Code generation can be done by **recursive traversal** of the AST
- As you write your compiler, we recommend starting with a **stack machine** (simpler!)
  - `./cool --asm` generates Cool-ASM stack machine code for Cool
    - use this to help you with PA3!
- Production compilers do different things:

Summary

- The activation record must be **designed together** with the code generator
- Code generation can be done by **recursive traversal** of the AST
- As you write your compiler, we recommend starting with a **stack machine** (simpler!)
  - `./cool --asm` generates Cool-ASM stack machine code for Cool
    - use this to help you with PA3!
- Production compilers do different things:
  - keep as many values as possible in registers, etc

Summary

- The activation record must be **designed together** with the code generator
- Code generation can be done by **recursive traversal** of the AST
- As you write your compiler, we recommend starting with a **stack machine** (simpler!)
  - `./cool --asm` generates Cool-ASM stack machine code for Cool
    - use this to help you with PA3!
- Production compilers do different things:
  - keep as many values as possible in registers, etc
  - save this stuff for PA4

After Break

- We will continue our discussion of this stack machine language by discussing the **“hard” features of Cool**
  - object layout for OOP
  - dynamic dispatch

After Break

- We will continue our discussion of this stack machine language by discussing the **“hard” features of Cool**
  - object layout for OOP
  - dynamic dispatch
- You should **start PA3c3 over break**
  - I assure you that you will regret it if you do not

After Break

- We will continue our discussion of this stack machine language by discussing the **“hard” features of Cool**
  - object layout for OOP
  - dynamic dispatch
- You should **start PA3c3 over break**
  - I assure you that you will regret it if you do not
- Don’t forget about the **midterm** on 3/26
  - I will hold a review session on 3/24 or 3/25 in the evening, keep an eye on Discord for a poll