Lecture K:

Code Shape I — Inside A Function

CSE401/501m:

Introduction to Compiler Construction

Instructor: Gilbert Bernstein

Administrivia

- Midterm Results — Will be out very soon; exam and solution will be posted online at that time
  - Score distribution looks ok overall
  - If you did not do as well as you'd hoped, take the opportunity to reflect and figure out why. This is only a fraction of your grade. Remember we're here to help.
- Parser/AST feedback — if you have questions, feedback or regrade requests, please email cse401-staff@cs and cc your partner on all emails. It's good to fix serious problems before moving forward.
- Checking is DUE next Tuesday
  - Make sure to come (w/partner) to sections this Thursday

Plan for This Week + a bit

- Today — basics of Code Generation / Code Shape
  - Will focus on Statements and Expressions *inside of a single function/method*
- Wednesday & Friday — OOP Concepts & Whole Program
  - How do we do layout in memory of objects?
  - How do we compile function calls?
  - How do we perform dynamic dispatch?
- Next Wed & Next Thu — MiniJava Codegen Details
  - How do we get our generated code to interact with the broader host system so that we can actually run it?

Outline

Structural Invariants

Expressions & Simple Statements

Booleans and Short-Circuiting

Statement Control Flow

Arrays

Outline

Structural Invariants

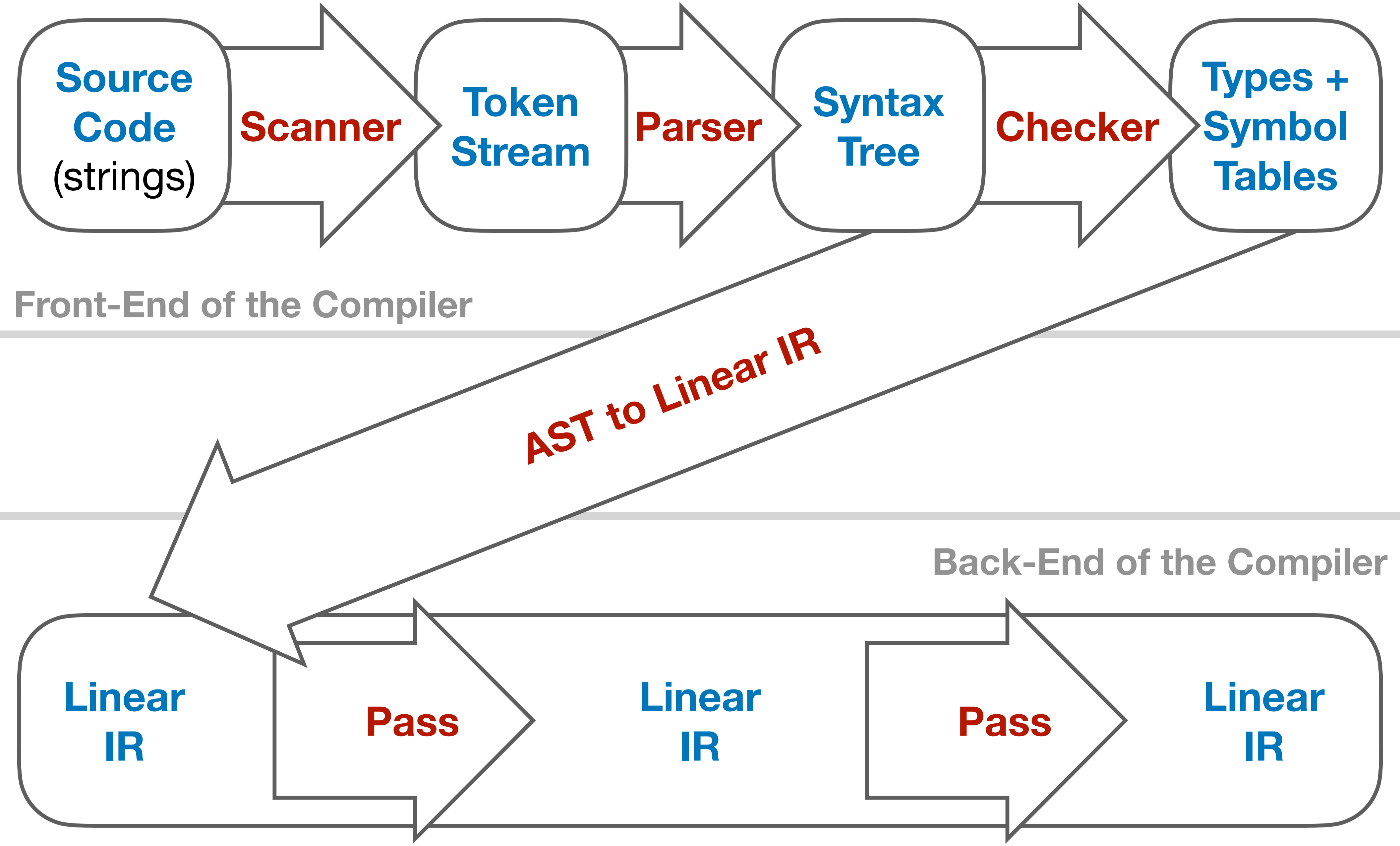
Expressions & Simple Statements

Booleans and Short-Circuiting

Statement Control Flow

Arrays

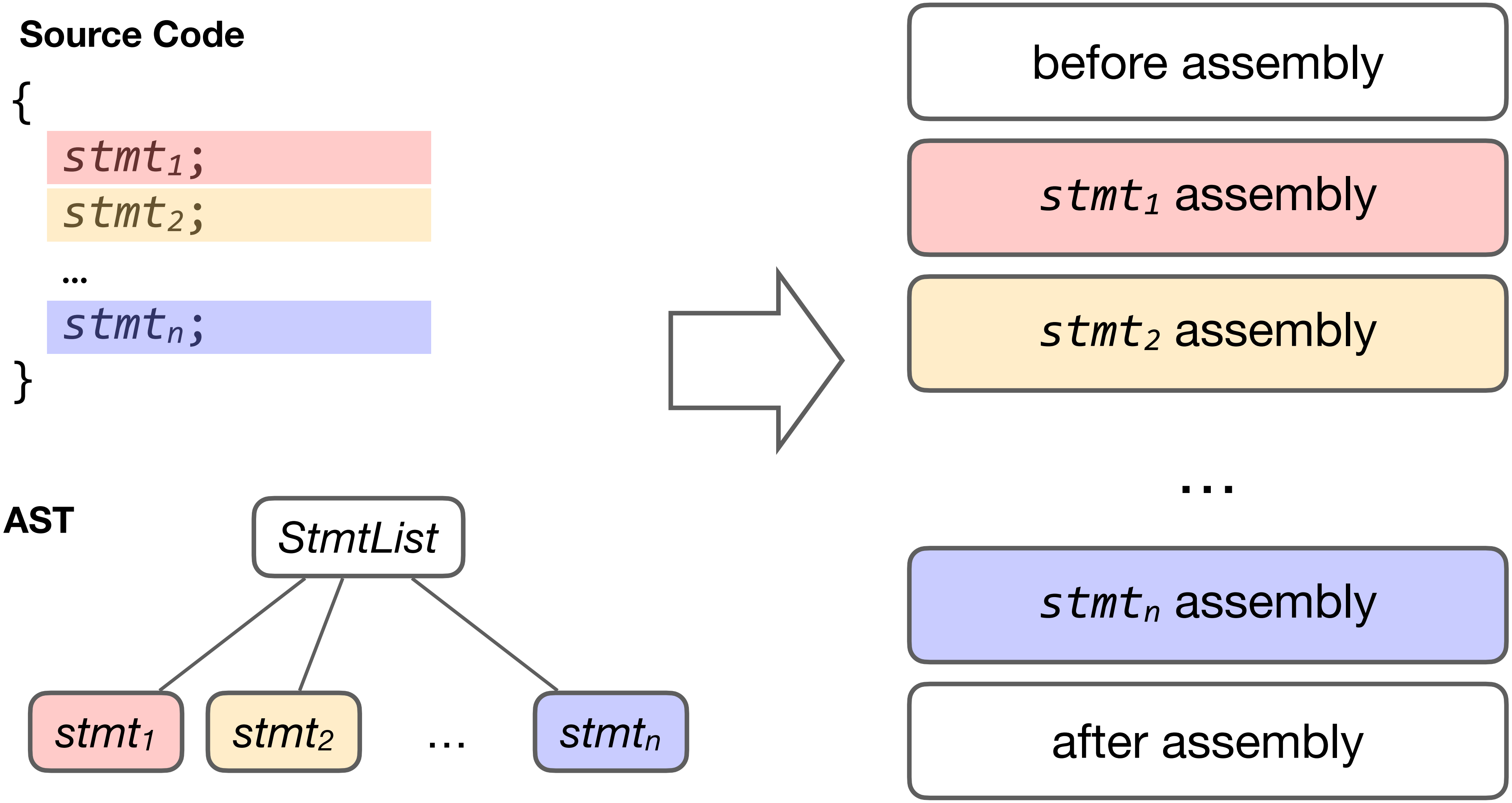
Where we are in the Compiler



From AST to Linear IR

- What are the possible options for structuring this pass?
  - ✦ Structural Recursion on the AST — Hard to come up with something else to do with an AST!
- How should we start thinking about writing a structurally recursive function?
  - ✦ In Medias Res (trans. “in the middle of events”)
- Which kind of AST Node should we think about first?
  - ✦ a generic Statement or Expression!
- Pay attention to what comes before, during, & after!

The Shape of Statements



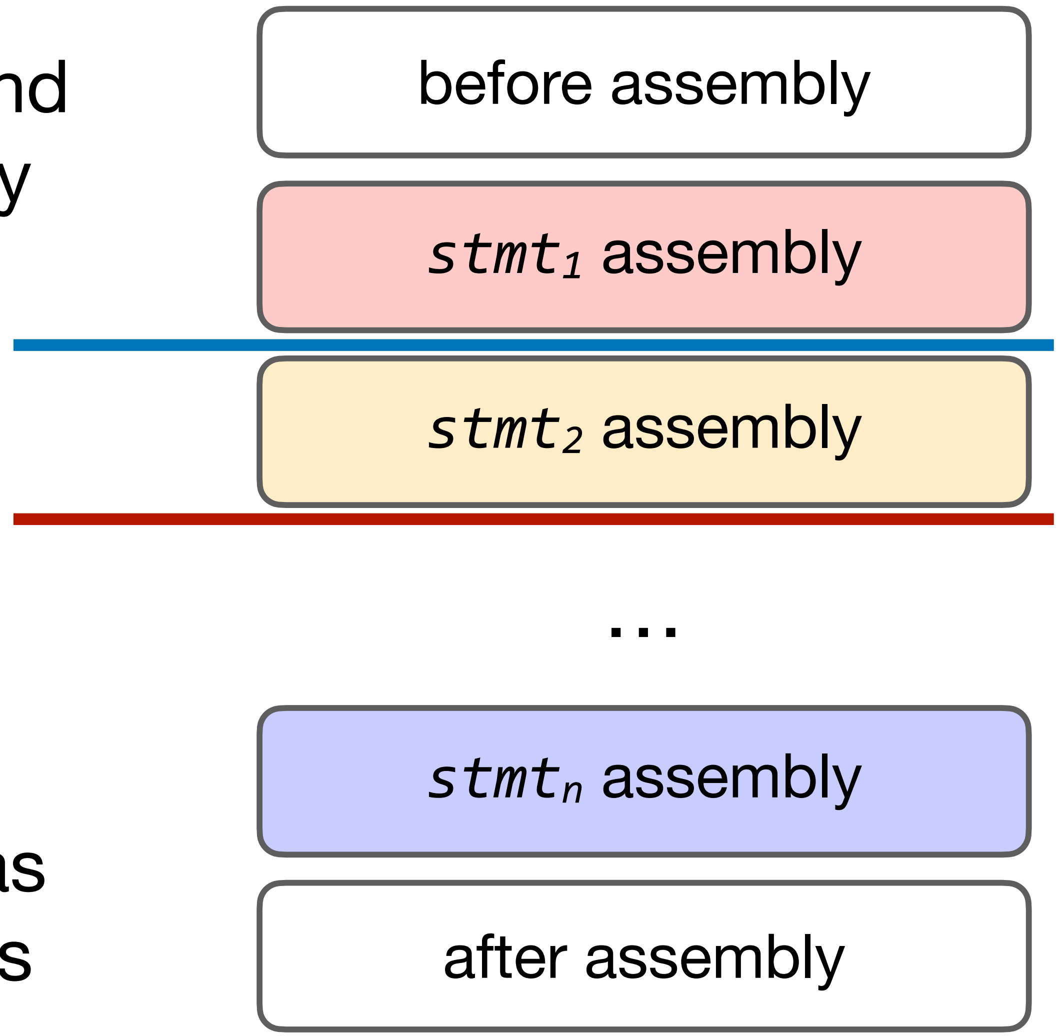
The Shape of Statements

Invariants

What must be true **before** and **after** each block of assembly code corresponding to a statement?

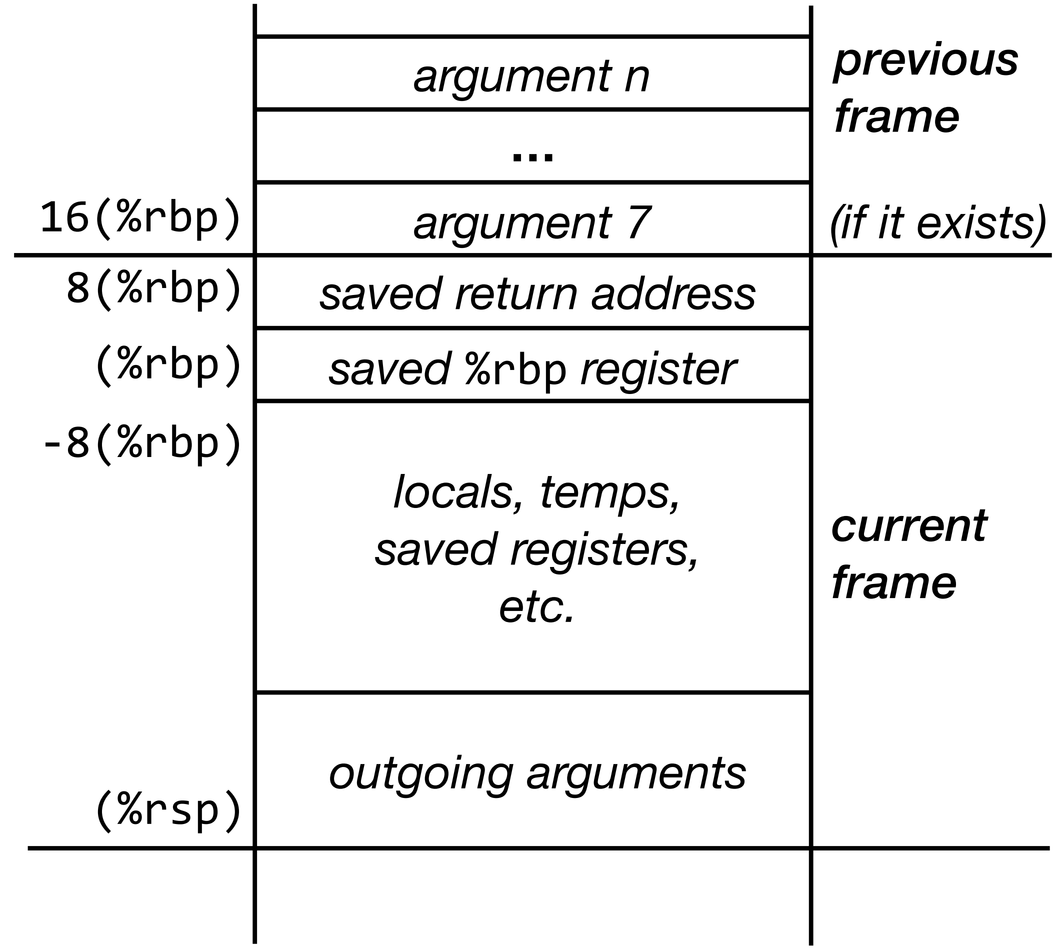
Every translation of a statement can **assume** the **invariant** is true before starting.

In return, each translation has to **guarantee** the **invariant** is true after finishing.



(Some) Invariants We Will Use

- The stack should be managed according to the ABI! (*note* **`%rsp, %rbp`**)
- All local variables should be stored on the stack frame between statements
- No guarantees on the contents of any other register



Registers

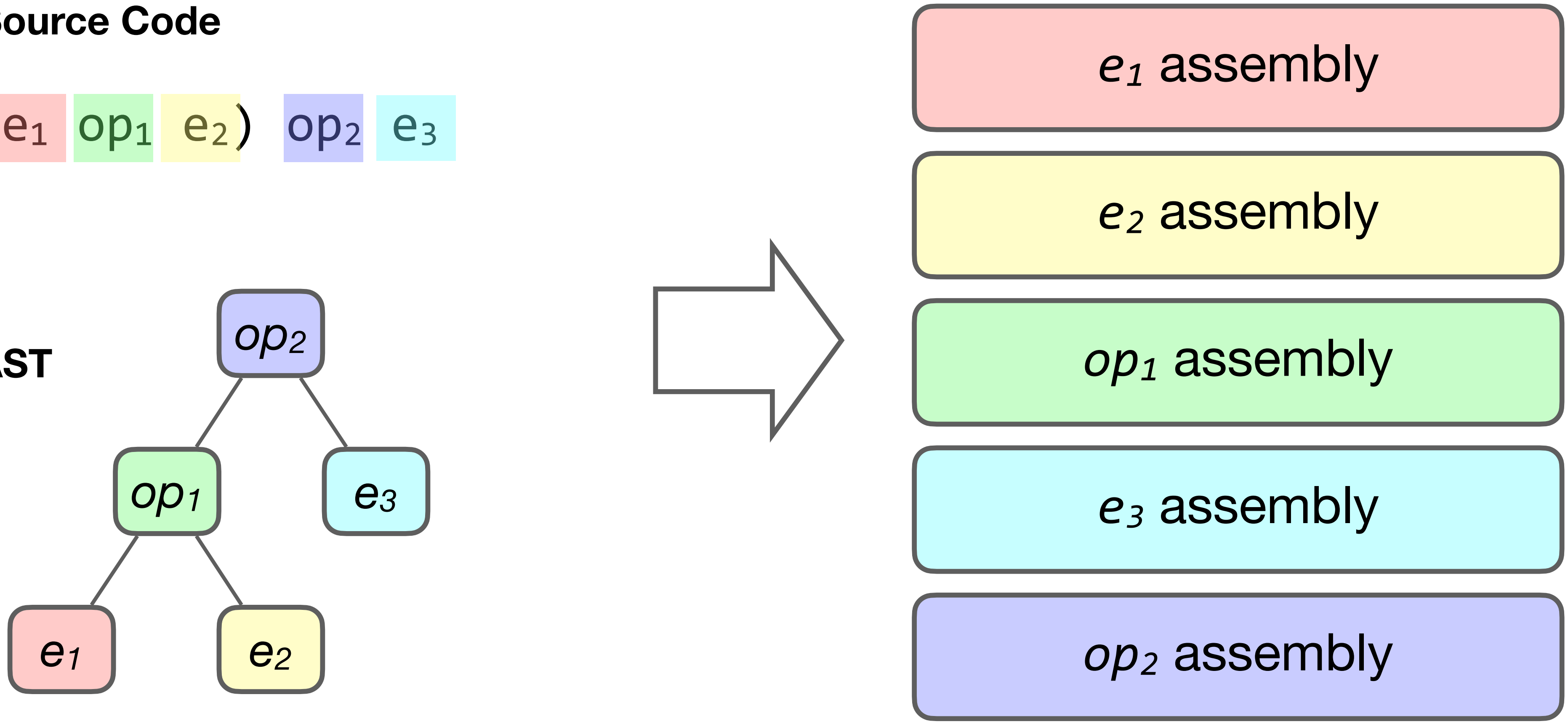
%rax	%rbx	%rcx	%rdx	%rsp	%rbp	%rsi	%rdi
%r8	%r9	%r10	%r11	%r12	%r13	%r14	%r15

The Shape of Expressions

Source Code

(e_1 op_1 e_2) op_2 e_3

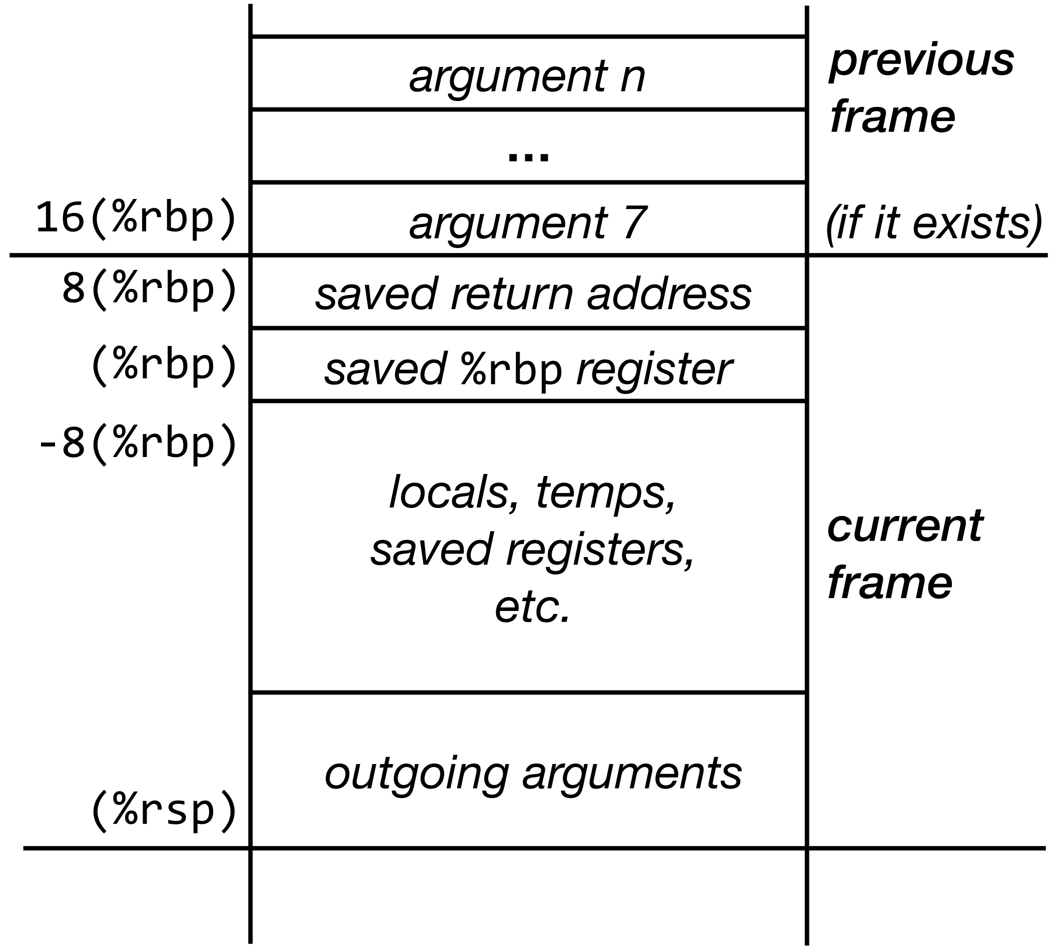
AST



What kind of traversal order is this? ***Post-Order***

(Some) Invariants We Will Use

- The stack should be managed according to the ABI! (*note* `%rsp, %rbp`)
- All local variables should be stored on the stack frame between statements
- No guarantees on the contents of any other register
- Expression Results will be held in `%rax`



Registers

%rax	%rbx	%rcx	%rdx	%rsp	%rbp	%rsi	%rdi
%r8	%r9	%r10	%r11	%r12	%r13	%r14	%r15

First Operand Location?

- consider evaluation of e_1 *op* e_2:
  - Eval e_1, then eval e_2 (result in `%rax`), then eval *op*
  - Where does the result of e_1 go?
- Idea 1: put the result of e_1 into `%rbx` — will this work?
- What if $e_2 = e_3\ op_2\ e_4$?
  - Then, we eval e_3, eval e_4 (into `%rax`), then eval op_2
  - Where does the result of e_3 go?
- A vicious cycle!

First Operand Location? (2)

- consider evaluation of e_1 *op* e_2:
  - Eval e_1, then eval e_2 (result in `%rax`), then eval *op*
  - Where does the result of e_1 go?
- Observation — the number of temporary LHS operands required may be arbitrarily large, but we have a limited number of registers
- Idea 2: put temporary intermediary values (LHS) on the stack. This has two consequences
  - Invariant “all other registers don’t matter” maintained
  - stack frame size is dynamic, not static (hence `%rbp`)
  - The cost of going to memory vs. registers

In Medias Res

- We've talked about statements and expressions *in general*, but we haven't talked about any *specific* statements or expressions!
- In reality, it's often very hard to get the general principle (i.e. choice of invariant) right on the first guess.
  - We have to work a bunch of examples, and then realize "oh no, that won't work at all"
- At that point you have to go back and *change your invariant*. (and then repeat this cycle a few times)
- **The big danger** — your code uses *different invariants in different cases*. Doing this will create complicated bugs.

Are Invariants Necessary?

- You don’t have to use these exact invariants
  - Enjoy your Freedom! 🦅
  - Or, the nice thing about standards is…
- But! you will probably get in trouble if you don’t think carefully about what invariants you want to maintain

Outline

Structural Invariants

Expressions & Simple Statements

Booleans and Short-Circuiting

Statement Control Flow

Arrays

Constants (Expressions)

- Source Code

```
17
```

- x86-64

```
movq $17,%rax
```

- Alternate form **(optimization)** when constant is 0

```
xorq %rax,%rax
```

Variables (Expressions)

- In MiniJava, all variables are either local or instance vars
- Source Code

```
x
```

- x86-64 (when variable is method-local)
  - *(stored at an offset in the stack frame, e.g. -16)*

```
movq -16(%rbp),%rax
```

- x86-64 (when dealing with an instance variable)
  - We will cover classes/objects next lecture…

Assignment Statements

- Source Code

```
var = exp;
```

- x86-64

<eval exp into %rax>

```
movq %rax,-16(%rbp)
```

- (if *var* is stored at -16 on the stack; otherwise, wherever it is stored)

Unary Minus

- Source Code

    `-exp`

- x86-64

    <eval exp into `%rax`*>*

    `negq %rax`

- **Optimization**
    - collapse `-(-exp)` to *exp*
- *(note: unary plus is a no-op)*

Binary Plus

- Source Code

  exp_1 + exp_2

- x86-64

  <eval exp_1 into %rax>

  <eval exp_2 into %rdx>

  ```
  addq %rdx,%rax
  ```

Binary Plus (Optimizations)

- If exp_2 is only a variable or constant, we don't need to load it into another register first. Instead, we can do

```
addq <exp₂>,%rax
```

- We can change exp_1 `+ (-`exp_2`)` into exp_1 `-` exp_2
- If `exp2` is `1`, we can replace this with

```
incq %rax
```

  - *Which is better? It depends on the microarchitecture. (i.e. processor implementation) So, is x86-64 code portable?*

Binary Sub., Mult.

- Same as addition (more or less)
  - Use `subq` for subtraction (but be careful of arg. order!)
  - Use `imulq` for mutliplication
- Some **optimizations**
  - Can replace `2*x` with `x << 1` or `x+x`
  - More complicated `10*x` = `(x << 3) + (x << 1)`
    - if multiplication is slow enough, maybe a good idea
  - Could use `decq` for `x-1`
  - Could use `leaq (%rax,%rax,4),%rax` to compute `5*x`

Signed Integer Division

- Source Code

  exp₁ / exp₂

- x86-64

```
<eval exp1 into %rax>   # recall: exp1 must be in %rax!
<eval exp2 into %rbx>
cqto                    # extend %rax into %rdx
idivq %rbx              # quotient in %rax, remainder in %rdx
```

- (yup, it's ugly as we talked about. It's also slow)

Optimization is…

- Very important in real systems!
  - ✦ Battery life, compute time, real-time applications (audio, games, video, flight-control)
- Best done **systematically**
  - ✦ Focus on important parts (e.g. inner-loop)
  - ✦ Based on controlled experiments, not pure theory
- Which is why **premature optimization** is…
- It can be ***tempting*** 😈 to implement lots of optimizations in your code gen, but each optimization increases complexity. Good compiler construction is about fighting complexity!

Outline

A Preview…

- Source Code

```
if (cond) stmt
```

- x86-64

<eval cond…>

jump to `skip` *if cond was false*

<code for stmt>

```
skip:
```

- Ok, but how do we compile *cond* then?

Boolean Expressions

- How do we generate code for …

  `x > y`

- e.g. How do we generate code for `if (x > y)` *`stmt`* ?
- What if it's more like `(x > y) && (y > z)` ?
- And what about `z = (x > y);` ?
- Approach 1: make code generation of Boolean expressions **depend on context**
- Approach 2: code generate for Boolean expressions in a **context-independent** way (but maybe less optimized)

And — a first attempt

- Source Code

  exp$_1$ && *exp*$_2$

- x86-64

  <eval exp$_1$ *into %rax>*

  <eval exp$_2$ *into %rdx>*

  ```
  andq %rdx,%rax
  ```

- What's wrong here?
- This code will execute *exp*$_2$ regardless of whether *exp*$_1$ is true or false — i.e. there is no **short-circuiting**!

Short-Circuiting Behavior

- Suppose we are computing the expression exp_1 `&&` exp_2 and assigning the result to a variable `res`.
  - i.e. `res =` exp_1 `&&` exp_2`;`
- The above should have the same meaning as

```
if (exp₁)
     res = exp₂;
else
     res = false;
```

- ***Encoding Note:*** *We can choose whatever encoding of true and false we want, but 1 and 0 are customary.*

And

- Source Code

exp_1 && exp_2

- x86-64

```
    <eval exp1 into %rax>
    cmpq $0,%rax
    je skip
    <eval exp2 into %rax>
skip:
```

Or

- Source Code

  exp$_1$ || *exp*$_2$

- x86-64

```
      &lt;eval exp1 into %rax&gt;
      cmpq $0,%rax
      jne skip
      &lt;eval exp2 into %rax&gt;
skip:
```

Not

- Source Code

```
!exp
```

- x86-64

```
<eval exp into %rax>
xorq $1,%rax
```

- Why doesn't this example use `notq`?
- For what encodings of Booleans does the above work?

Less Than

- Source Code

  $exp_1 < exp_2$

- x86-64

```
      <eval exp1 into %rax>
      <eval exp2 into %rbx>
      cmpq %rbx,%rax
      jnl gebranch
      movq $1,%rax
      jmp done
gebranch:
      movq $0,%rax
done:
```

Optimizing Booleans

- The examples above are definitely much less efficient than they have to be.
  - e.g. `true && x = x`, etc.
- A more efficient approach might ***fuse*** the computation of Booleans into the control flow statements they're used in
  - i.e. treat the case of assigning a Boolean to a variable as the **uncommon case**, and the case of branching on a Boolean expression as the **common case**
  - Doing this safely requires a **different invariant**; this is a nice stretch goal if you're looking for extra credit

e.g. Optimizing Less Than

- Source Code

  exp_1 < exp_2

- x86-64

```
<eval exp1 into %rax>
<eval exp2 into %rbx>
cmpq   %rbx,%rax
setl   %al          # sets low byte of %rax to 0/1
movzbq %al,%rax     # zero-extend to 64 bits
```

- Note: this uses x86 features we didn't cover, like `%al` (lowest 8-bits of `%rax`), $\texttt{set}_{cc}$, which is like $\texttt{j}_{cc}$ but sets instead of jumping, and `movzbq` which zero-extends

Outline

Structural Invariants

Expressions & Simple Statements

Booleans and Short-Circuiting

Statement Control Flow

Arrays

If

- Source Code

```
if (cond) stmt
```

- x86-64

```
        <eval cond into %rax>
        cmpq $0,%rax
        je skip
        <code for stmt>
    skip:
```

- ***note: (true for all labels) need to make sure labels get unique names when generating code!***

If-Else

- Source Code

```
if (cond) stmt₁ else stmt₂
```

- x86-64

```
        <eval cond into %rax>
        cmpq $0,%rax
        je else
        <code for stmt₁>
        jmp done
else:   <code for stmt₂>
done:
```

While

- Source Code

```
while (cond) stmt
```

- x86-64

```
test: <eval cond into %rax>
      cmpq $0,%rax
      je done
      <code for stmt>
      jmp test
done:
```

Alternate While

- Source Code

```
while (cond) stmt
```

- x86-64

```
        jmp test
loop:   <code for stmt>
test:   <eval cond into %rax>
        cmpq $0,%rax
        jne loop
```

- Why do this alternative?
- Doing this moves one jump out of the inner loop
  - ✦ usually a win, and about as simple as the first method

Do-While

- Source Code

```
do stmt while (cond)
```

- x86-64

```
loop: <code for stmt>
test: <eval cond into %rax>
      cmpq $0,%rax
      jne loop
```

Jumping All Around

```
if (cond1) {
   if(cond2)
      stmt1
   else
      stmt2
} else
   stmt3
```

We might turn this into assembly code with the following "shape" …

```
                <eval cond1 into %rax>
                cmpq $0,%rax
                je else1
                <eval cond2 into %rax>
                cmpq $0,%rax
                je else2
                <code for stmt1>
                jmp done2
else2:  <code for stmt2>
done2:
                jmp done1
else1:  <code for stmt3>
done1:
```

Jump Chaining

- Naive code generation can produce jumps to jumps when we nest control flow structures
- **Optimization** — if a jump has as its target an unconditional jump, change the target of the first jump to the target of the second jump
  - repeat until we reach fixed-point
  - This can be fixed up at different points (e.g. via CFG)

Switch

- Source Code

```
switch(exp) {
  case 0: stmt0;
  case 1: stmt1;
  case 2: stmt2;
}
```

- *(break is an unconditional jump to the end of the switch)*
- compilation strategy 1 — reduce to a set of nested if-else statements; however, this may require O(n) comparisons
- compilation strategy 2 — compile to a “jump table”

Switch (Jump Table)

- Source Code

```
switch(exp) {
  case 0: stmt0;
  case 1: stmt1;
  case 2: stmt2;
}
```

- x86-64

```
    <put exp in %rax>
    # if %rax not between 0 and 2,
    # then jmp to the default label
    movq    swtable(,%rax,8),%rax
    jmp     *%rax
      .data
swtable:
      .quad L0
      .quad L1
      .quad L2
      .text
L0: <stmt0>
L1: <stmt1>
L2: <stmt2>
```

Outline

Structural Invariants

Expressions & Simple Statements

Booleans and Short-Circuiting

Statement Control Flow

Arrays

Arrays

- In Java, arrays are
  - 0-origin — i.e. an array with n elements indexes them as `a[0]` through `a[n-1]` (`a[n]` is out of bounds)
  - 1-dimension (Java) — to get more than one dimension, we use nested arrays (i.e. `a[i][j]`, not `a[i,j]`)
- Regardless of what kind of arrays, the key step is to do an indexing calculation to guide the load/store

Arrays (0-based, 1-dimension)

- Source Code

  exp₁[*exp₂*]

- x86-64

  <eval exp₁ into %rax>

  <eval exp₂ into %rdx>

```
# if load, then
movq (%rax,%rdx,8),…    # 64-bit array elem.

# if store, then
movq …,(%rax,%rdx,8)
```

Alternatively…

```
imulq $8,%rdx
   # (or use left shift)
addq %rdx,%rax
movq (%rax),…
```

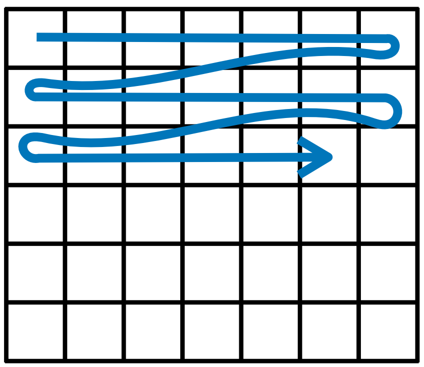
Arrays of Arrays

- If we have an array type, we can have Array(Array(T))
- What are the downsides?
  - No way to guarantee that the arrays are rectangular — we could have ragged arrays
  - Extra memory lookup / indirection cost
- So, it makes sense to have “native” multi-dimensional arrays

2-dimensional Arrays

- If the language has 2-dimensional arrays, then
  - Are they row-major or column-major?

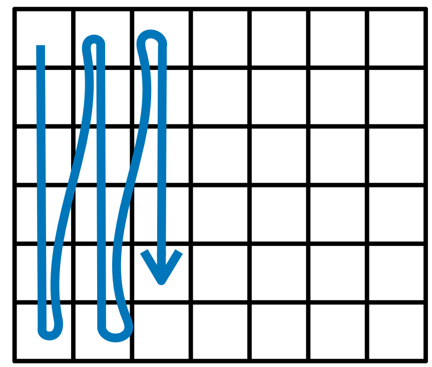
Row Major



`A[i,j]`

A is NxM

Column Major



- What is the formula for 2d array indexing?

Row Major

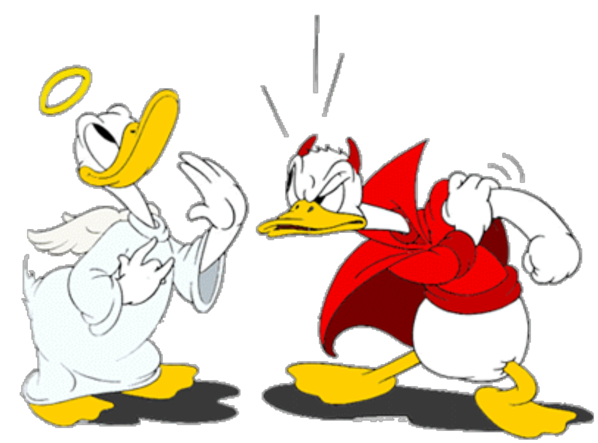
`&A + 8*(M*i + j)`

Column Major

`&A + 8*(i + N*j)`

Optimization is…

- Mighty **Tempting**!
  - "Oh, who will notice if I cut a little corner here or there?"
- Once you start violating your **invariants**, you open the door to all kinds of ***fiendish*** 😈 bugs.
- You can pick whatever invariants you want, but you have to **keep your promise** to yourself!
  - If it's an invariant, you can **assume** it
  - But if it's in an invariant, then you're responsible for **guaranteeing** it too!



Next Time…

- Code Generation for Objects (Wed/Fri)
  - How to represent objects
  - How to represent method calls
  - Inheritance and overriding
- Optimization (next Mon)
- Practical Details for Project (next week; Wed/Thu)