CSE 390B, Winter 2022

Building Academic Success Through Bottom-Up Computing

Midterm Debrief, Compilers

Midterm Debrief, Revisiting Time Management, Introduction to the Compiler, Project 6 Overview

If joining virtually, please have your camera turned on if you can!

Lecture Outline

- **Midterm Debrief**

- Introduction to the Compiler
  - Overview, Scanner, Parser

- Project 6 Overview
  - Midterm Corrections, Professor Meeting Report

Midterm Debrief

- You all put great effort into the exam!
- Challenging midterm for the 50 minutes you were allotted
- Key Takeaways:
  - Excellent job on the Hack Assembly and circuit design problems!
  - Importance of taking the time to read the problem carefully
  - Time management: Prioritizing problems you feel most confident in

Midterm Next Steps

- If you think a problem was graded unfairly or wrong, submit a regrade request in Gradescope!
  - Don't be afraid to do so; this is a great learning opportunity for both you and the course staff

- You will have a chance to get points back with midterm corrections as part of Project 6

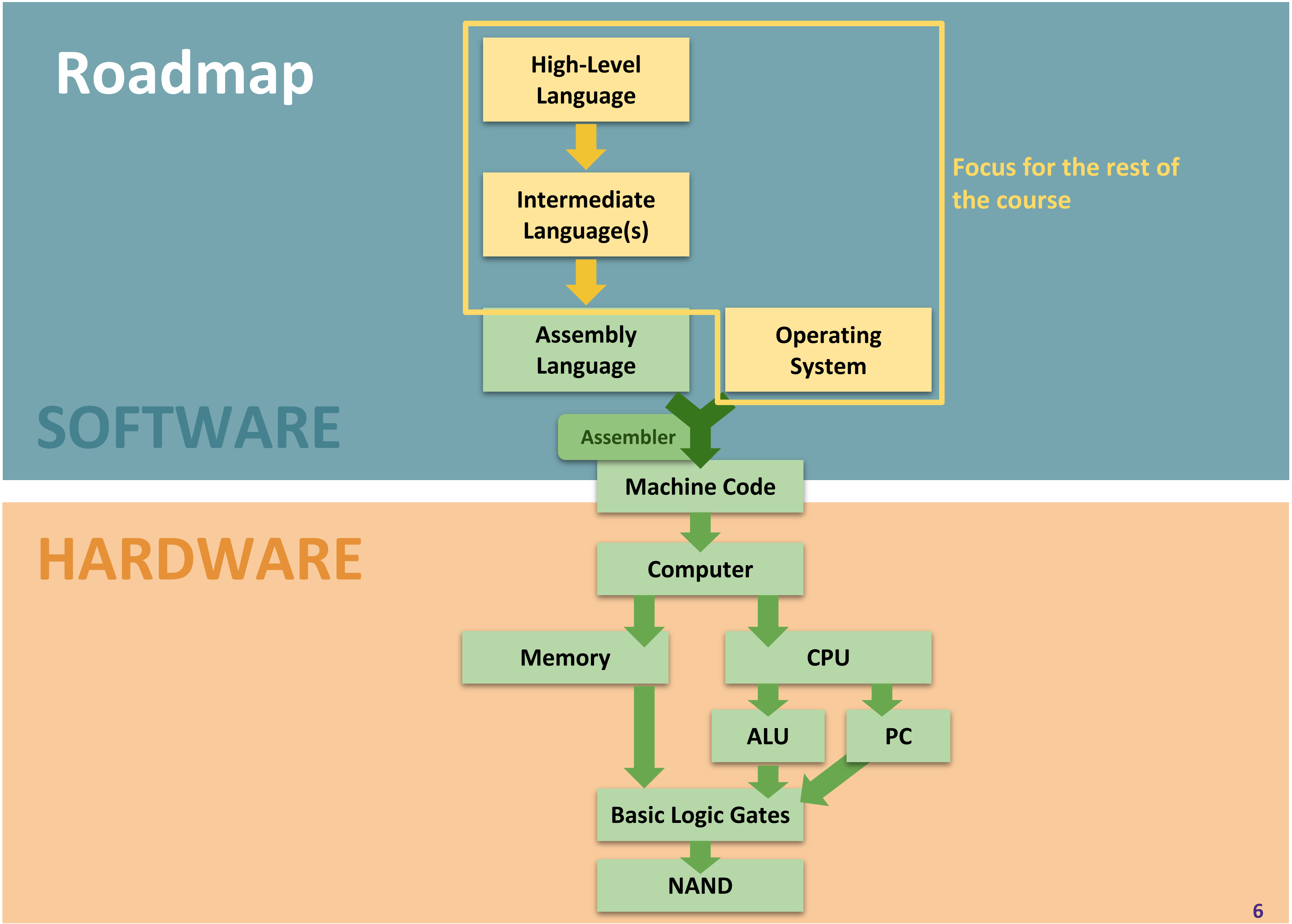
Lecture Outline

- Midterm Debrief

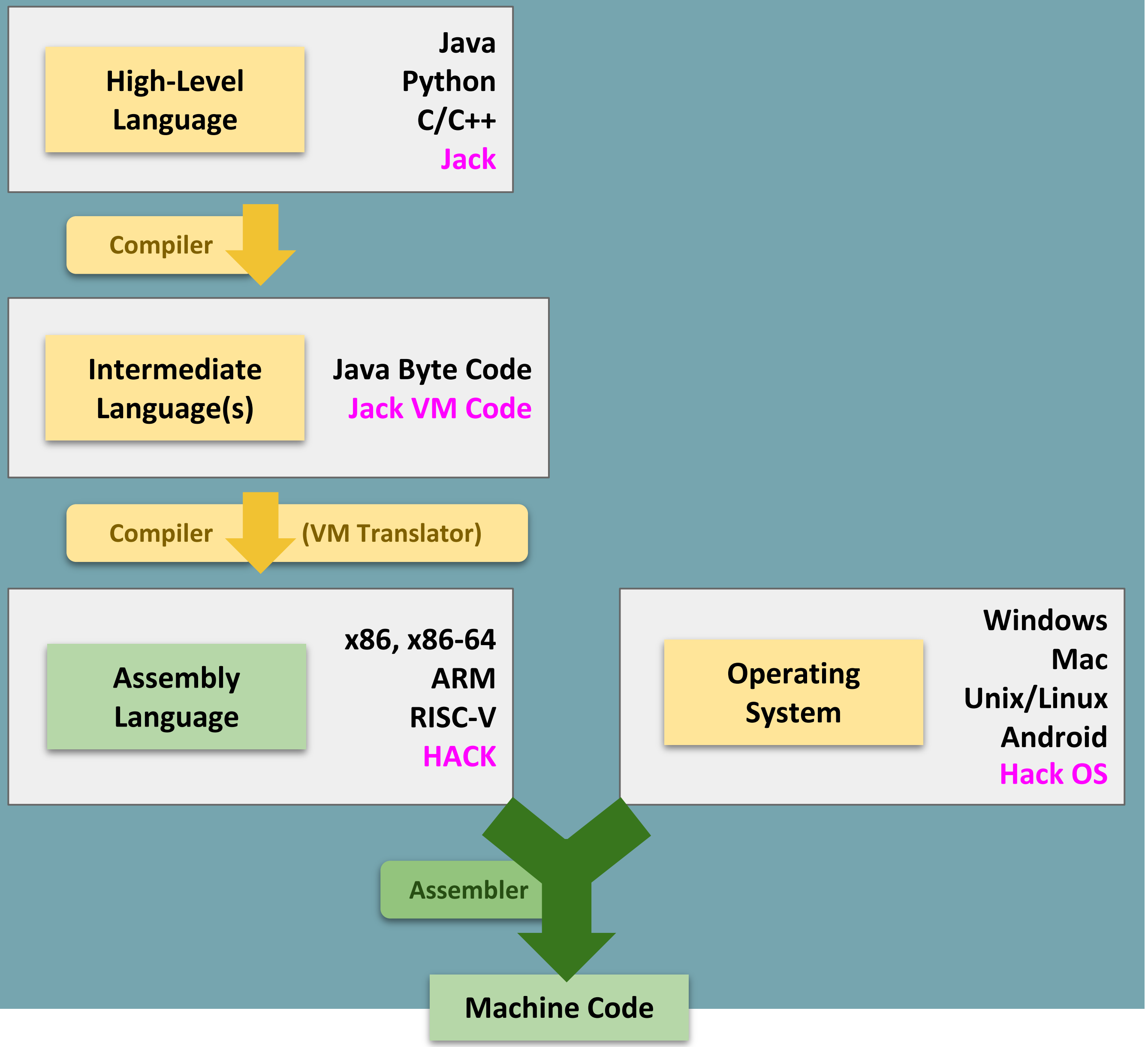
- **Introduction to the Compiler**
  - **Overview, Scanner, Parser**

- Project 6 Overview
  - Midterm Corrections, Professor Meeting Report

Roadmap

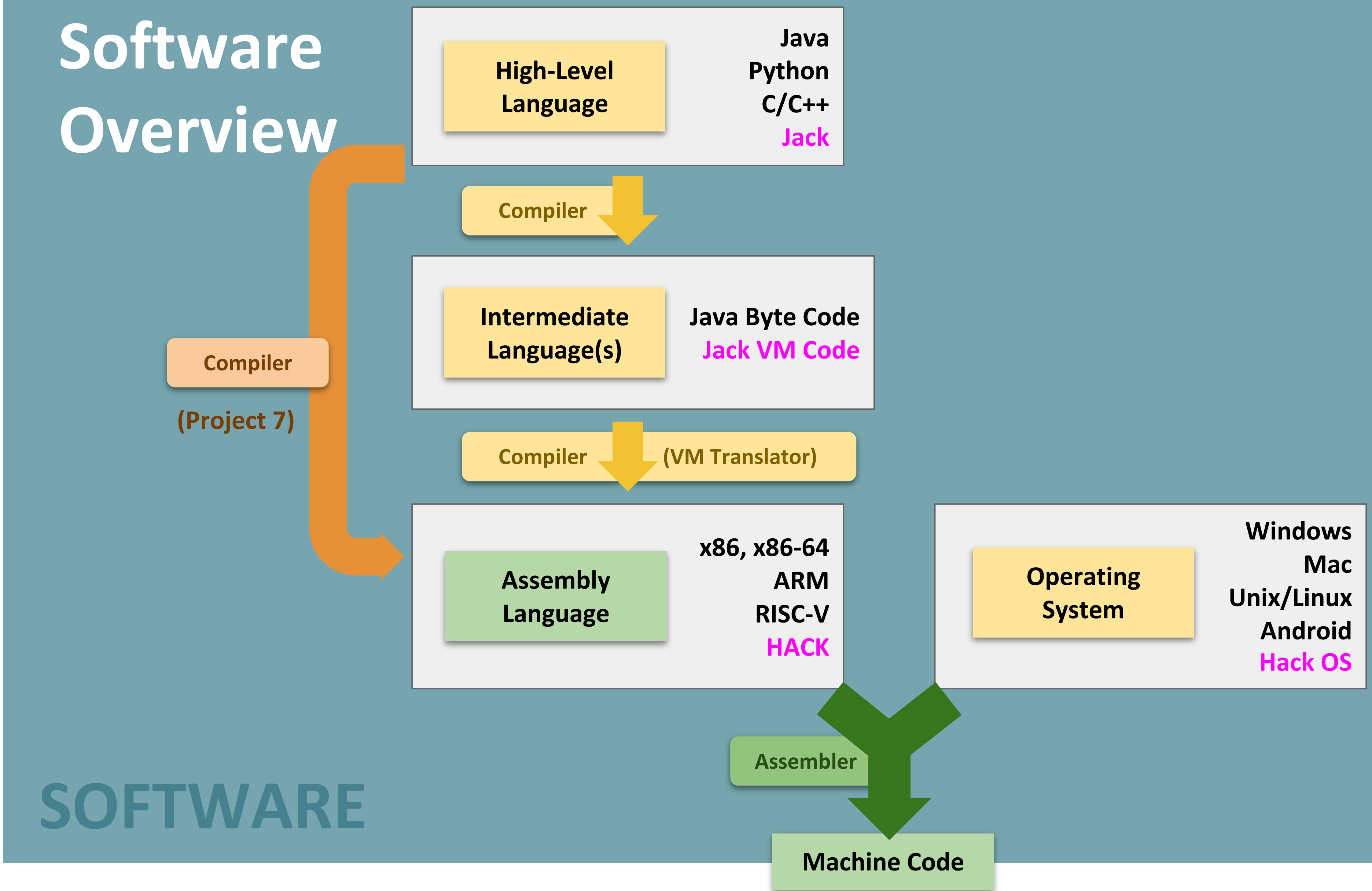

High-Level Language
Intermediate Language(s)
Assembly Language
Operating System
Focus for the rest of the course
SOFTWARE
Assembler
Machine Code
HARDWARE
Computer
Memory
CPU
ALU
PC
Basic Logic Gates
NAND

Software Overview



SOFTWARE

Software Overview



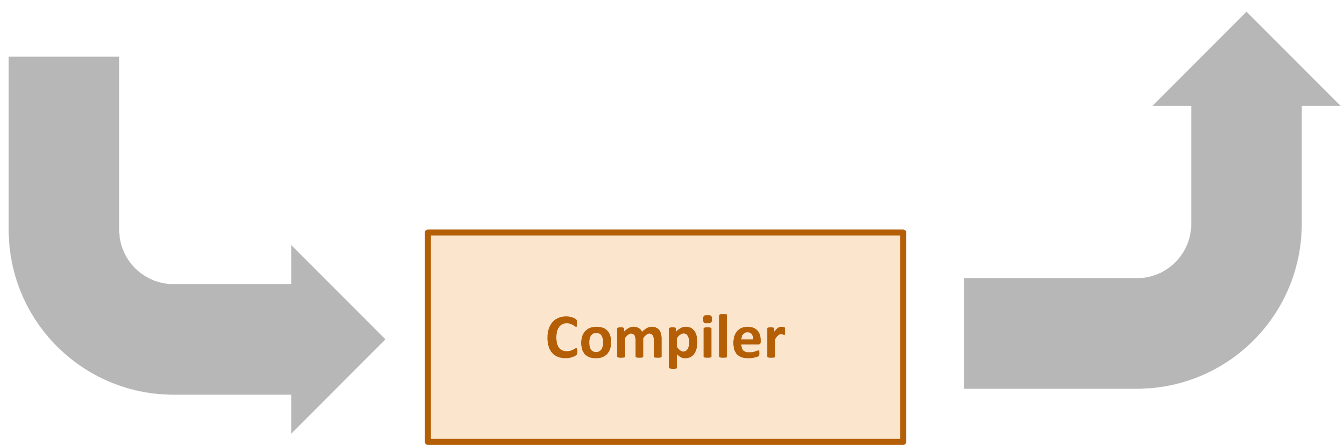
The Compiler: Goal

```
public int fact(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * fact(n - 1);
  }
}
```

High-Level Language

```
(fact)
  @R0
  M=M+1
  @R1
  D=A
  @ifbranch
  D;JEQ
```

Assembly Language



The Compiler: Goal

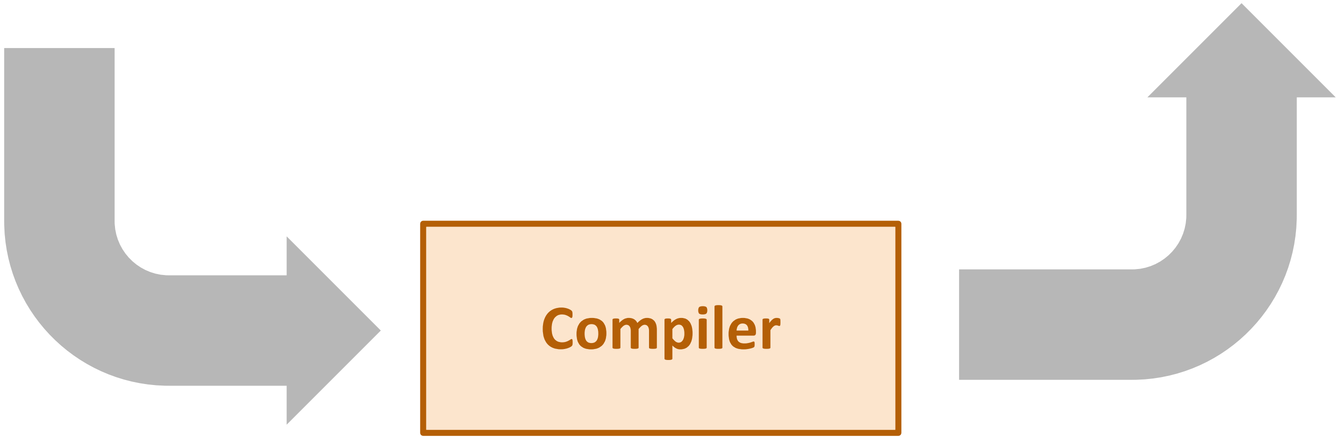
```
public int fact(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * fact(n - 1);
  }
}
```

High-Level Language

Theory Definition: a string, from the set of strings making up a language

```
(fact)
  @R0
  M=M+1
  @R1
  D=A
  @ifbranch
  D;JEQ
```

Assembly Language



The Compiler: Goal

```
public int fact(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * fact(n - 1);
  }
}
```

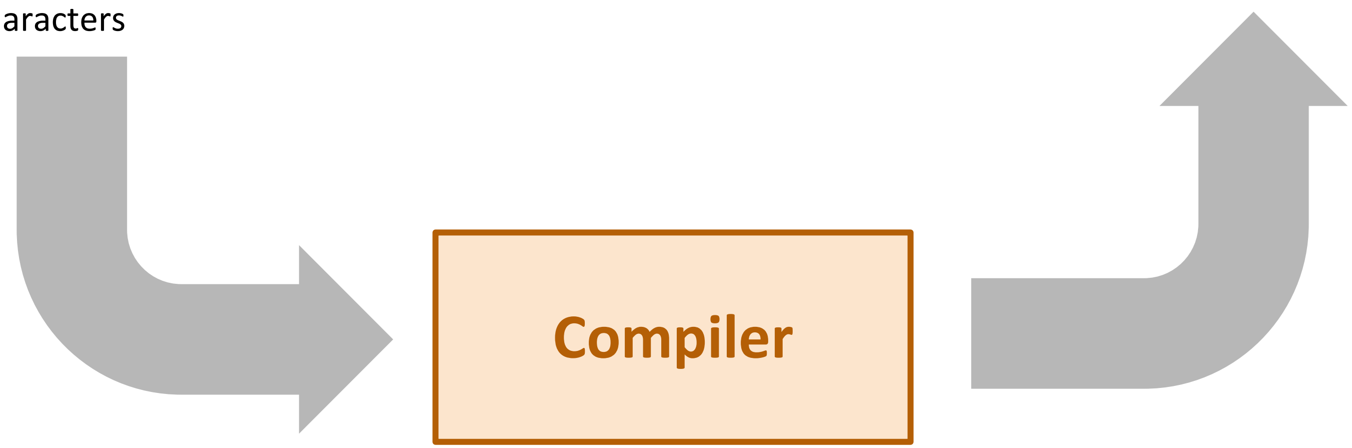
High-Level Language

Theory Definition: a string, from the set of strings making up a language

Practical Definition: a file containing a bunch of characters

```
(fact)
  @R0
  M=M+1
  @R1
  D=A
  @ifbranch
  D;JEQ
```

Assembly Language



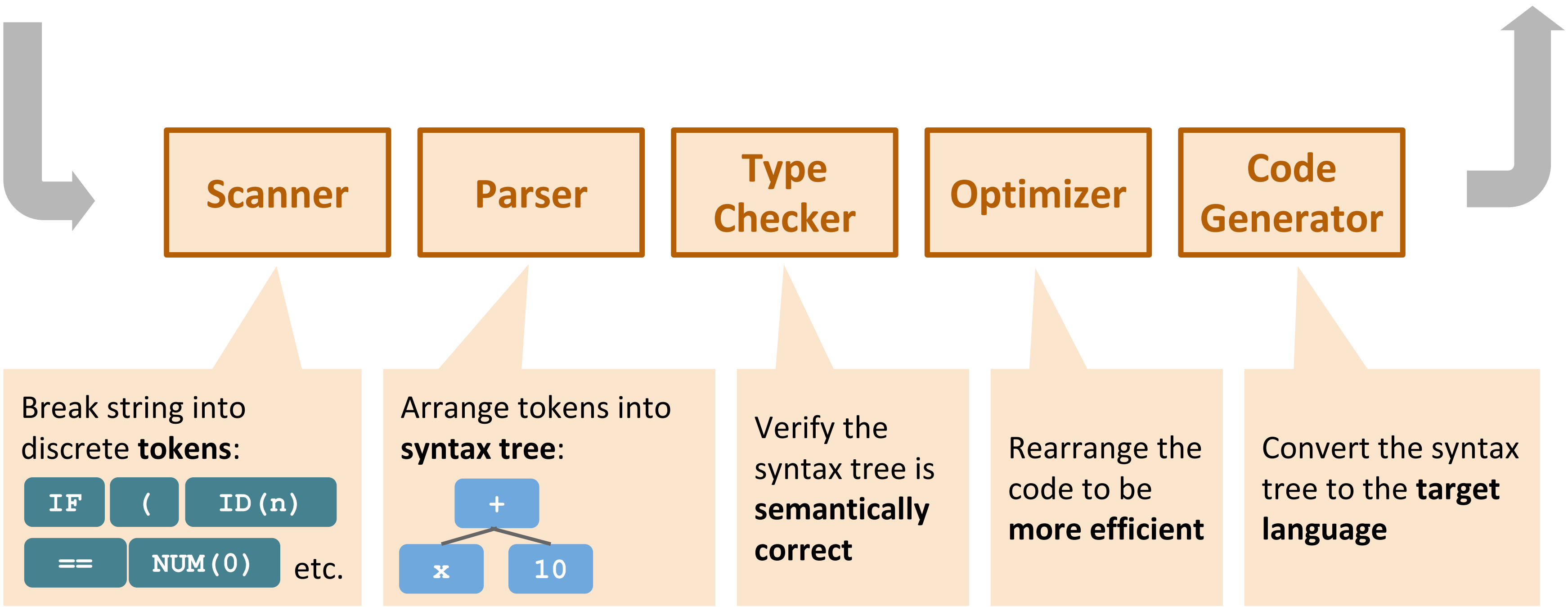
The Compiler: Implementation

```
public int fact(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * fact(n - 1);
  }
}
```

High-Level Language

```
(fact)
  @R0
  M=M+1
  @R1
  D=A
  @ifbranch
  D;JEQ
```

Assembly Language



Lecture Outline

- Midterm Debrief

- **Introduction to the Compiler**
  - **Overview, Scanner, Parser**

- Project 6 Overview
  - Midterm Corrections, Professor Meeting Report

Aside: The Jack Language

- The High-Level Language we will use to program your Hack computer
- Very similar to Java: mostly just a different set of keywords sprinkled around
  - Makes compiling easier

```
static void main() {
  int a, bar;
  bar = 10;
}

int f(int a) {
  return 2;
}
```

Java

≈

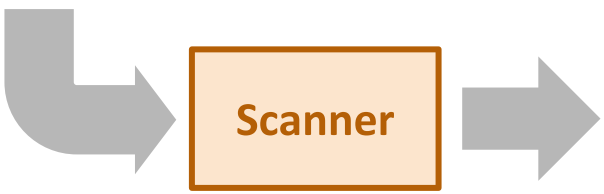
```
function void main() {
  var int a, bar;
  let bar = 10;
}

method int f(int a) {
  return 2;
}
```

Jack

The Scanner

```
function void main() {
  var int a, bar;
  let bar=10; // init
}
```

Jack



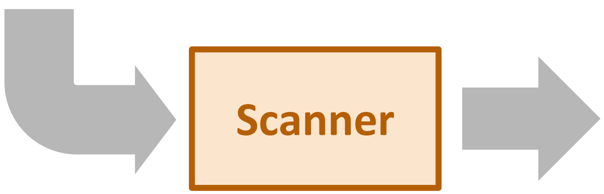
FUNCTION VOID ID(main)
LPAREN RPAREN LCURLY VAR
INT ID(a) COMMA ID(bar)
SEMICOLON LET ID(bar)
EQUALS NUM(10) SEMICOLON
RCURLY

Token Stream

The Scanner

```
function void main() {
  var int a, bar;
  let bar=10; // init
}
```

Jack

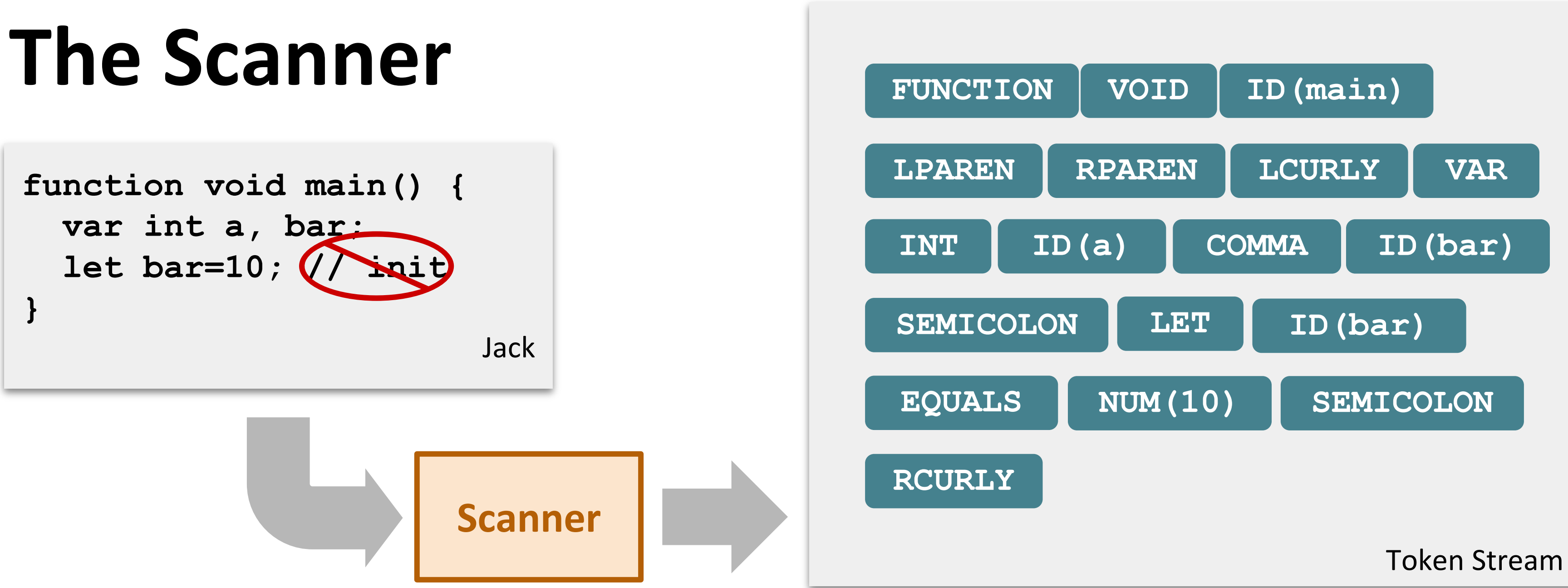


FUNCTION VOID ID(main)
LPAREN RPAREN LCURLY VAR
INT ID(a) COMMA ID(bar)
SEMICOLON LET ID(bar)
EQUALS NUM(10) SEMICOLON
RCURLY

Token Stream

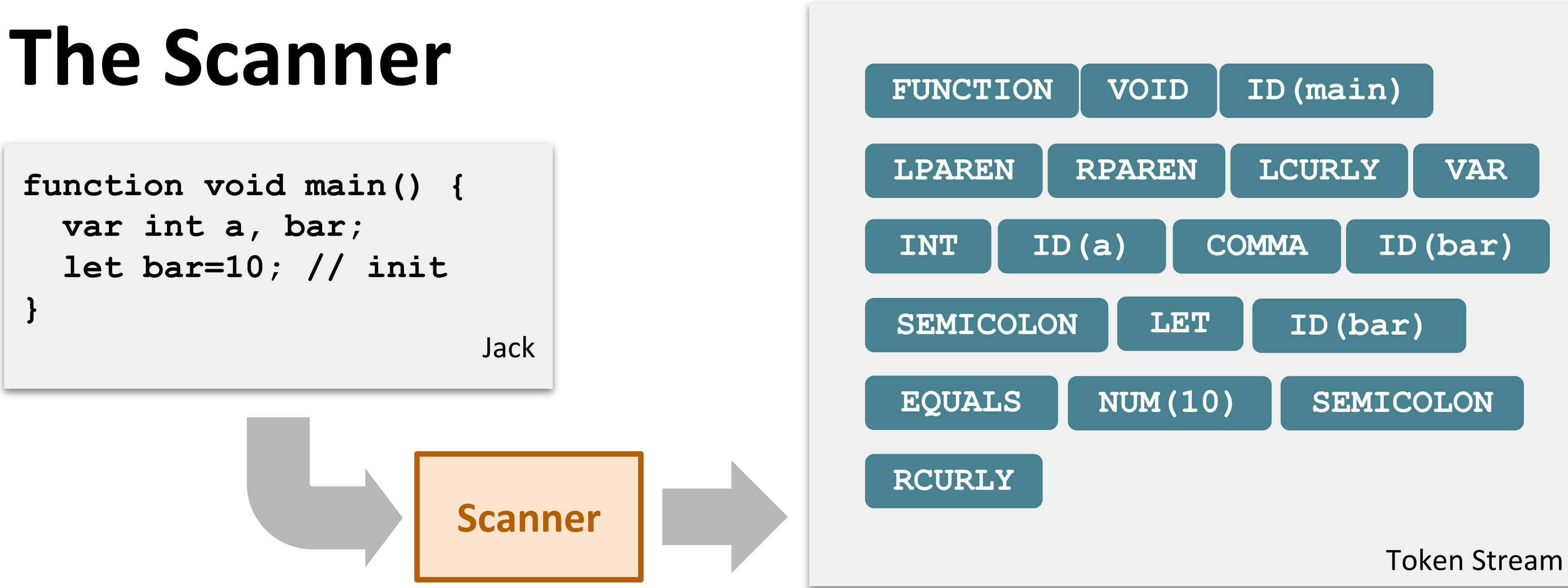
- Reads a giant string, breaks down into tokens
  - Each token has a type: what role does this token play?
    - E.g., LCURLY is a type representing an occurrence of "{"
  - What types do we care about? The "building blocks" of our programming language:
    - Keywords (e.g., FUNCTION)
    - Operators (e.g., EQUALS)
    - Punctuation (e.g., SEMICOLON COMMA)

The Scanner



- In addition to a type, some tokens carry a value:
  - Identifiers (e.g., ID(a))
  - Numbers (e.g., NUM(10))
- Scanner should present a *clean* token stream
  - No whitespace or comments: the rest of the compiler only wants to consider things that change program meaning

The Scanner

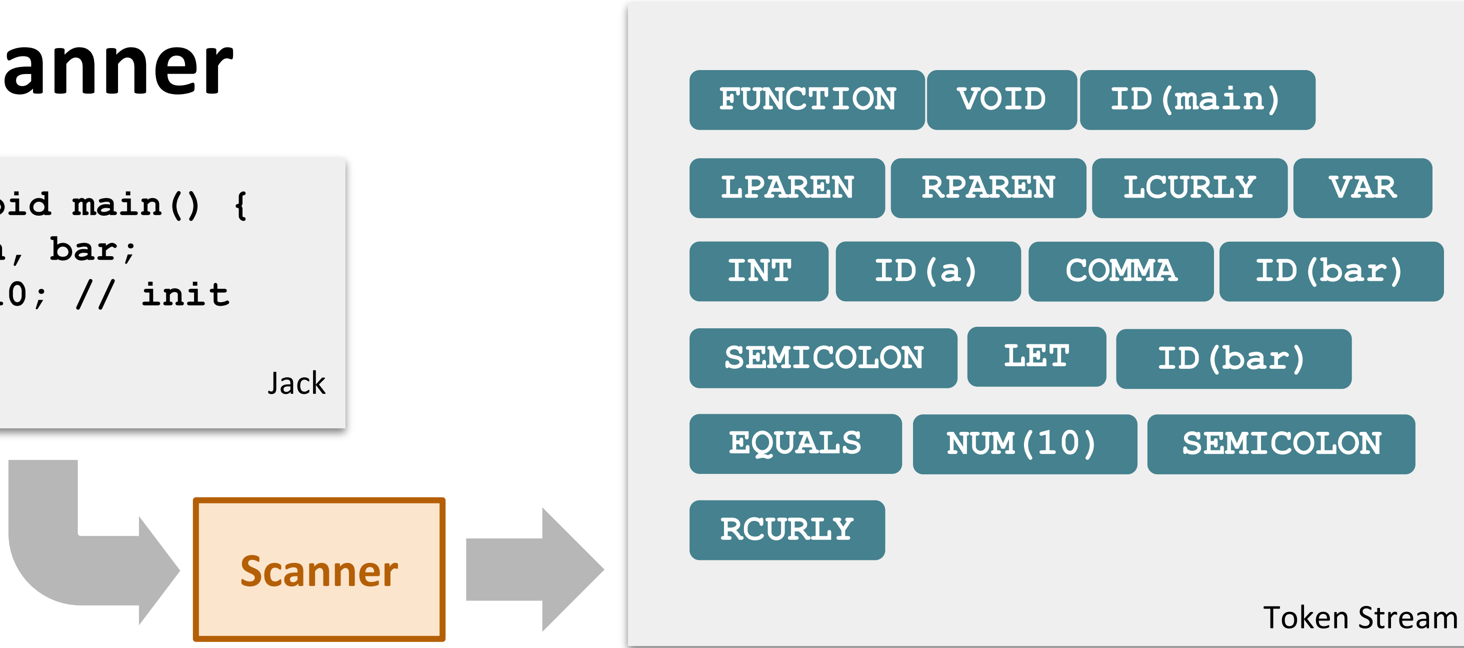


- What if we split the input program on whitespace, and match each segment to a token type? (E.g., "{" → LCURLY)

The Scanner

```
function void main() {
  var int a, bar;
  let bar=10; // init
}
```

Jack



- What if we split the input program on whitespace, and match each segment to a token type? (E.g., “{“ → LCURLY)

- Tempting, but we would end up with “a,” “bar;” “bar=10;”
  - Whitespace is tricky: generally, we want to ignore it, but we can’t count on it being there

The Scanner: How?

curr

```
; let bar=10;
```

Jack

Accumulated: ;

Token Stream

- Observation: many tokens have disjointed starting characters
- Keep cursor on current char
  - Break off a token when we complete one
  - If the next char could be part of this token, accumulate it
- How to distinguish built-in keywords (e.g., “let”) from identifiers (e.g., “bar”)?
  - Simple: when token is done, check against list of keywords

The Scanner: How?

curr

```
; let bar=10;
```

Jack

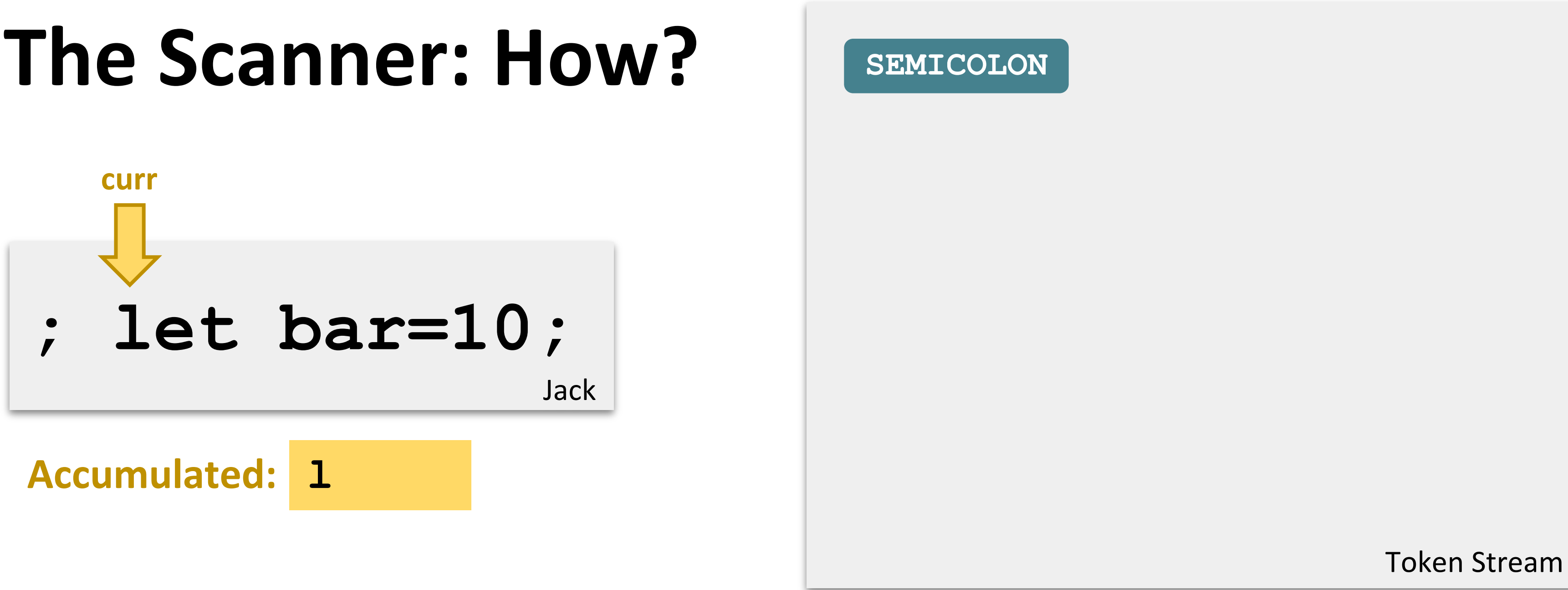
Accumulated:

SEMICOLON

Token Stream

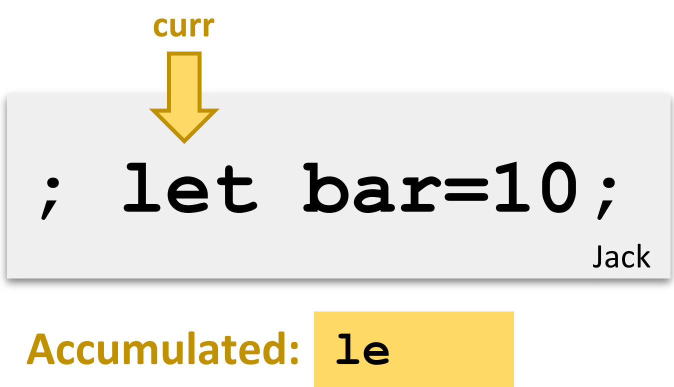
- Observation: many tokens have disjointed starting characters
- Keep cursor on current char
  - If the char *could* be part of this token, accumulate it
  - If not, complete the current token
- How to distinguish built-in keywords (e.g., “let”) from identifiers (e.g., “bar”)?
  - Simple: when token is done, check against list of keywords

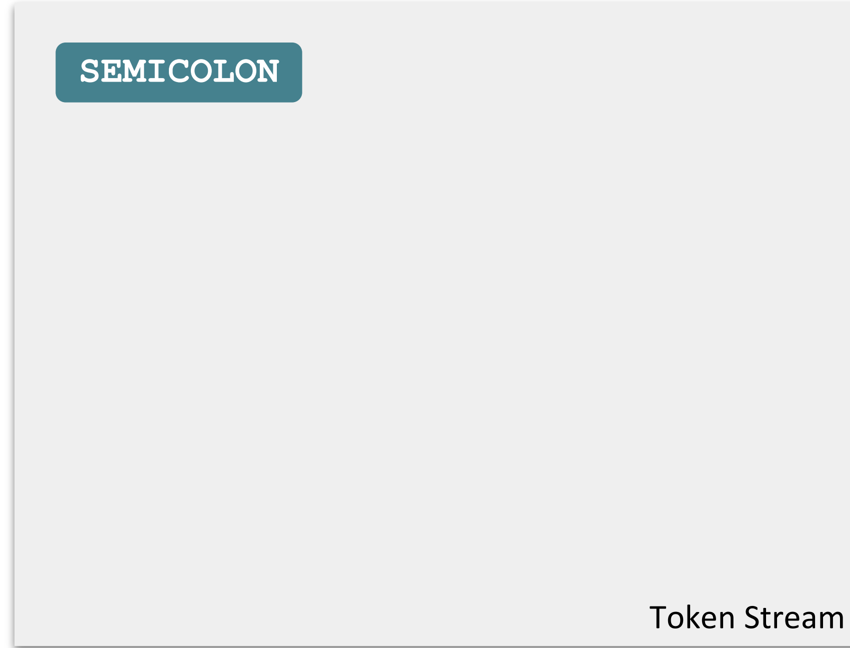
The Scanner: How?



- Observation: many tokens have disjointed starting characters
- Keep cursor on current char
  - If the char *could* be part of this token, accumulate it
  - If not, complete the current token
- How to distinguish built-in keywords (e.g., “let”) from identifiers (e.g., “bar”)?
  - Simple: when token is done, check against list of keywords

The Scanner: How?





- Observation: many tokens have disjointed starting characters
- Keep cursor on current char
  - If the char *could* be part of this token, accumulate it
  - If not, complete the current token
- How to distinguish built-in keywords (e.g., "let") from identifiers (e.g., "bar")?
  - Simple: when token is done, check against list of keywords

The Scanner: How?

curr

```
; let bar=10;
```

Jack

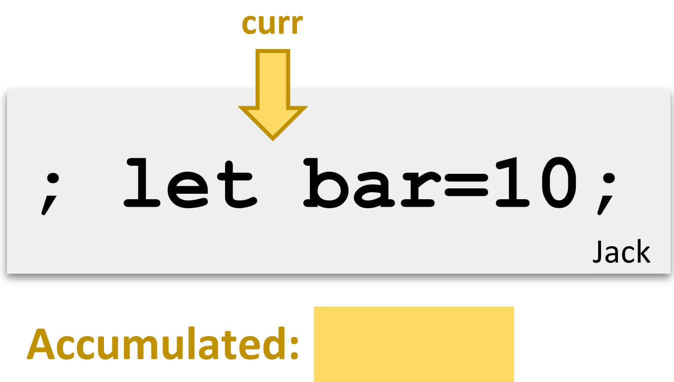
Accumulated: `let`

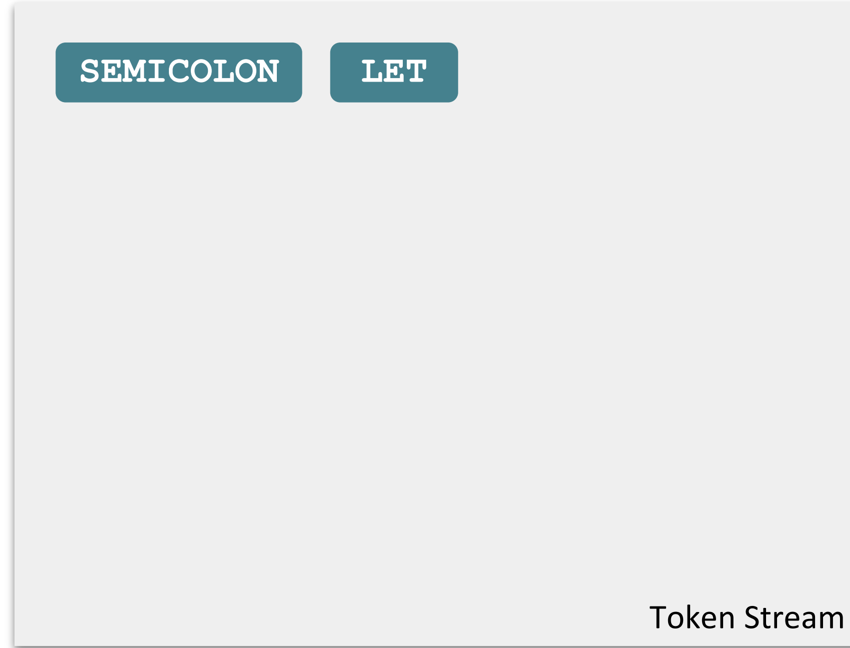
SEMICOLON

Token Stream

- Observation: many tokens have disjointed starting characters
- Keep cursor on current char
  - If the char *could* be part of this token, accumulate it
  - If not, complete the current token
- How to distinguish built-in keywords (e.g., “let”) from identifiers (e.g., “bar”)?
  - Simple: when token is done, check against list of keywords

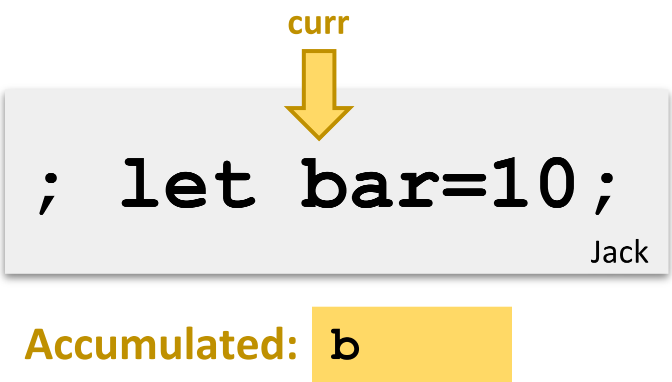
The Scanner: How?

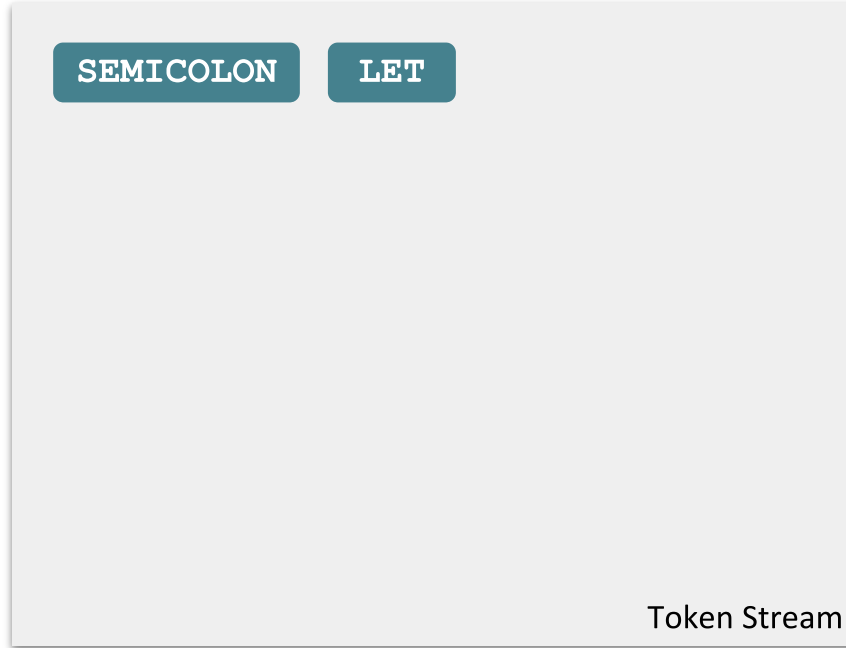




- Observation: many tokens have disjointed starting characters
- Keep cursor on current char
  - If the char *could* be part of this token, accumulate it
  - If not, complete the current token
- How to distinguish built-in keywords (e.g., “let”) from identifiers (e.g., “bar”)?
  - Simple: when token is done, check against list of keywords

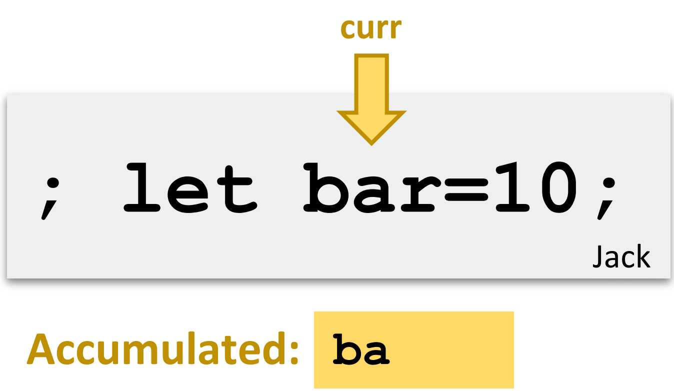
The Scanner: How?

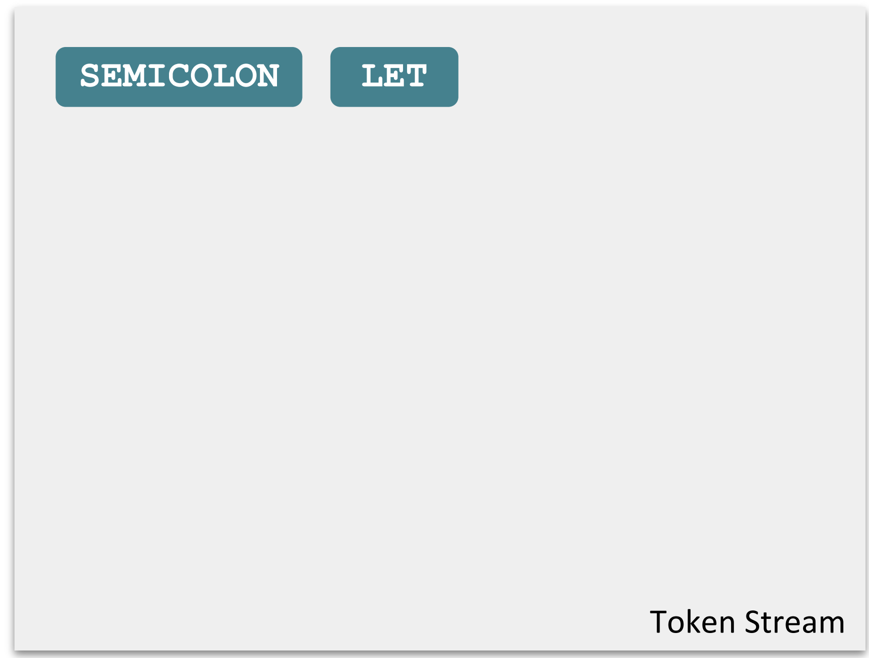




- Observation: many tokens have disjointed starting characters
- Keep cursor on current char
  - If the char *could* be part of this token, accumulate it
  - If not, complete the current token
- How to distinguish built-in keywords (e.g., “let”) from identifiers (e.g., “bar”)?
  - Simple: when token is done, check against list of keywords

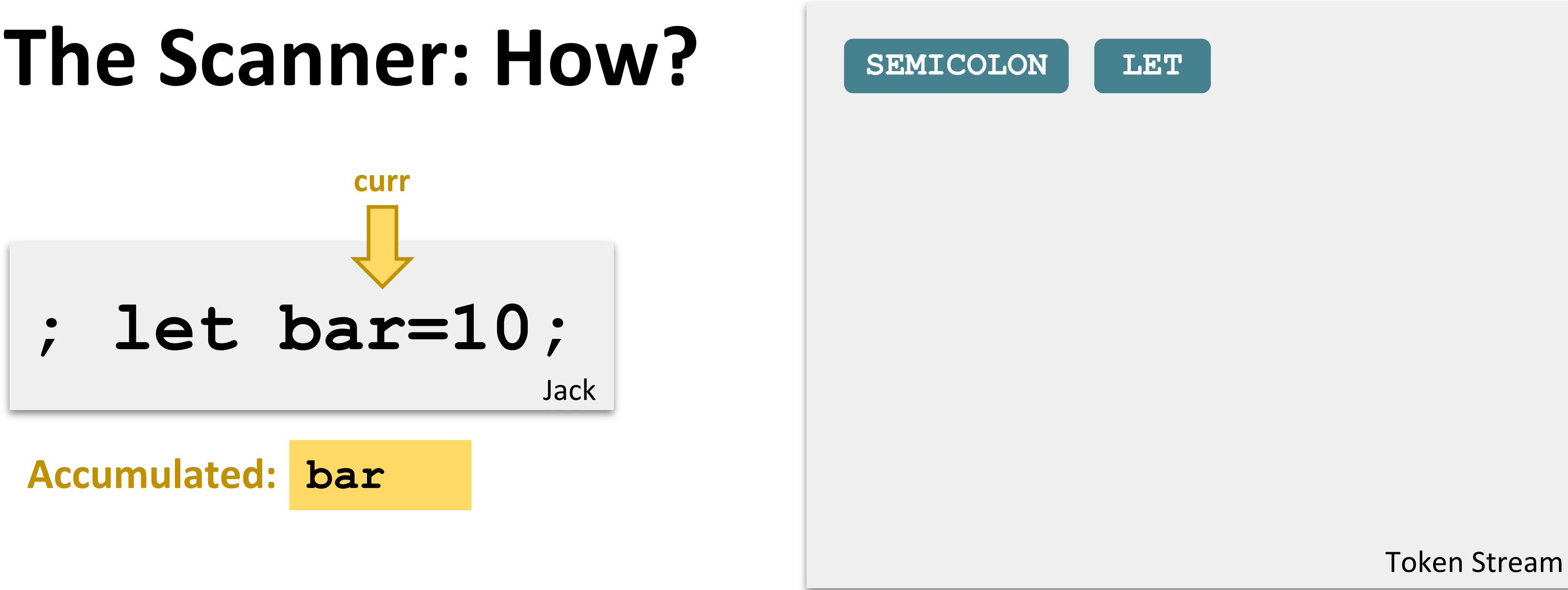
The Scanner: How?





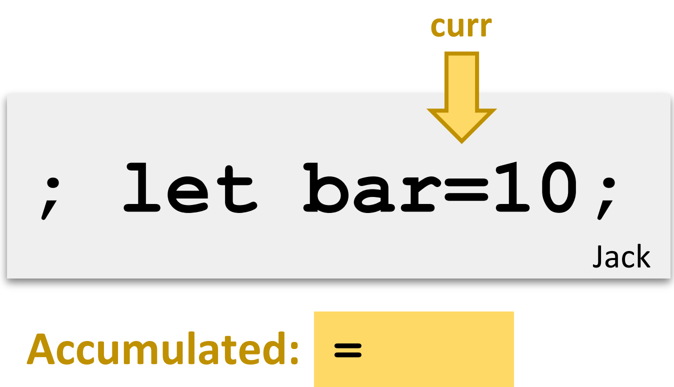
- Observation: many tokens have disjointed starting characters
- Keep cursor on current char
  - If the char *could* be part of this token, accumulate it
  - If not, complete the current token
- How to distinguish built-in keywords (e.g., “let”) from identifiers (e.g., “bar”)?
  - Simple: when token is done, check against list of keywords

The Scanner: How?



- Observation: many tokens have disjointed starting characters
- Keep cursor on current char
  - If the char *could* be part of this token, accumulate it
  - If not, complete the current token
- How to distinguish built-in keywords (e.g., "let") from identifiers (e.g., "bar")?
  - Simple: when token is done, check against list of keywords

The Scanner: How?



- Observation: many tokens have disjointed starting characters
- Keep cursor on current char
  - If the char *could* be part of this token, accumulate it
  - If not, complete the current token
- How to distinguish built-in keywords (e.g., “let”) from identifiers (e.g., “bar”)?
  - Simple: when token is done, check against list of keywords

The Scanner: How?

curr

```
; let bar=10;
```

Jack

Accumulated: 1

SEMICOLON LET ID(bar) EQUALS

Token Stream

- Observation: many tokens have disjointed starting characters
- Keep cursor on current char
  - If the char *could* be part of this token, accumulate it
  - If not, complete the current token
- How to distinguish built-in keywords (e.g., “let”) from identifiers (e.g., “bar”)?
  - Simple: when token is done, check against list of keywords

The Scanner: Why?

- Fundamentally: The compiler can't reason about a massive string, so we need to boil it down to its meaning first
  - A great place to start is grouping characters that form a "word"

- Engineering-wise: Separation of concerns
  - A stream of tokens is an important abstraction for many file-processing tasks, not just compiling
  - Cleaning away whitespace and comments makes rest of compiler simpler

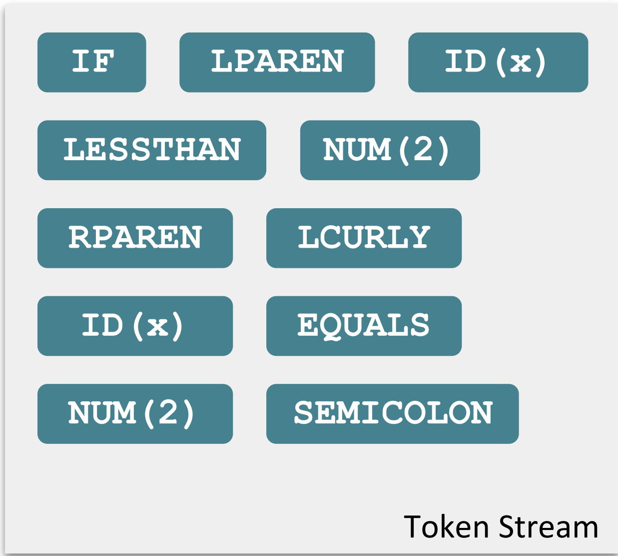
Lecture Outline

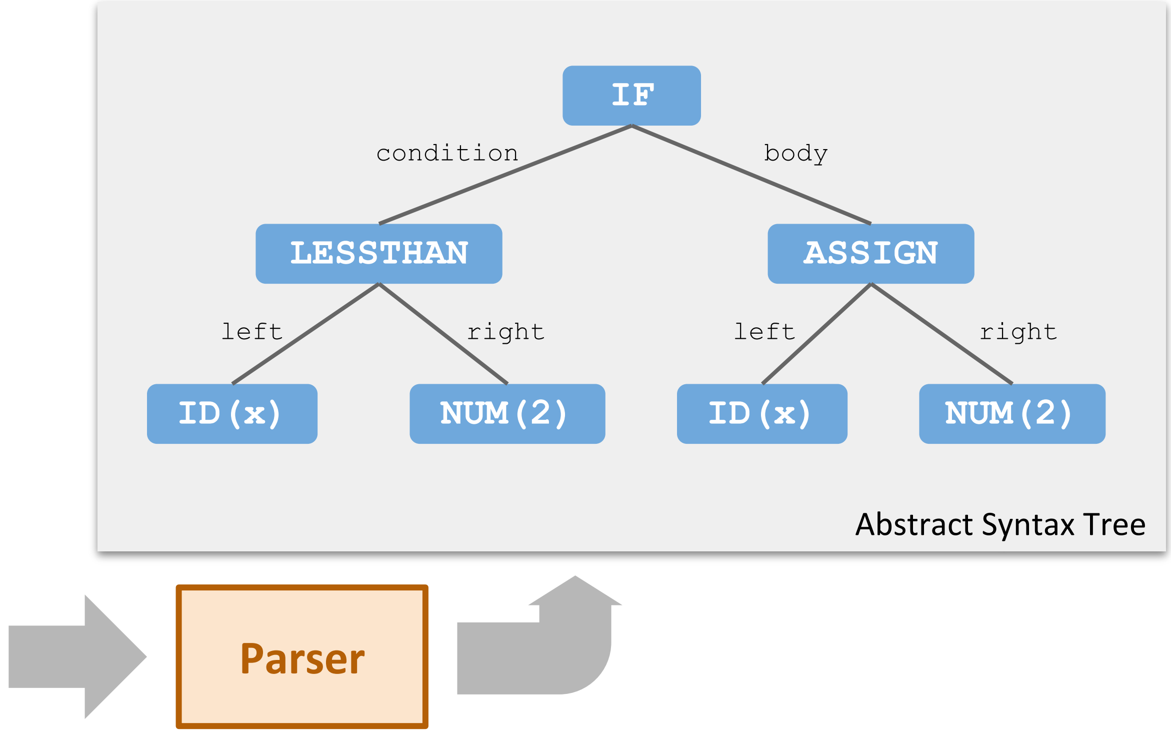
- Midterm Debrief

- **Introduction to the Compiler**
  - **Overview, Scanner, Parser**

- Project 6 Overview
  - Midterm Corrections, Professor Meeting Report

The Parser





- Takes in the *flat* token stream and outputs a *structured* tree representation of program constructs
- Result: an **Abstract Syntax Tree**
  - Captures the structural features of the program
  - Important distinction: cares about big-picture syntax (E.g., entire **`if`** statement) rather than nitty-gritty syntax (E.g., semicolons, parentheses, even word "if" used to write that **`if`** statement)

Describing a Programming Language

- Many ways to define programming languages, some formal
  - We won't cover language definition in depth
  - See CSE 341, CSE 401, CSE 402

- Example: Statements vs. Expressions

Statements *Perform an action*	**Expressions** *Evaluate to a result*

Statements — *Perform an action*

- Assignment Statement

```
x = y;
```

- If Statement

```
if (x == 0) {
  x = y;
}
```

Expressions — *Evaluate to a result*

- Operators

```
x == 0;
```

- Variable

```
x
```

- Constant

```
24
```

Describing a Programming Language

- These broad categories lend themselves well to recursive definitions
  - Easily express all possible configurations of the language constructs

Symbolic Example

```
if (x == 0) {
  x = y;
}
```

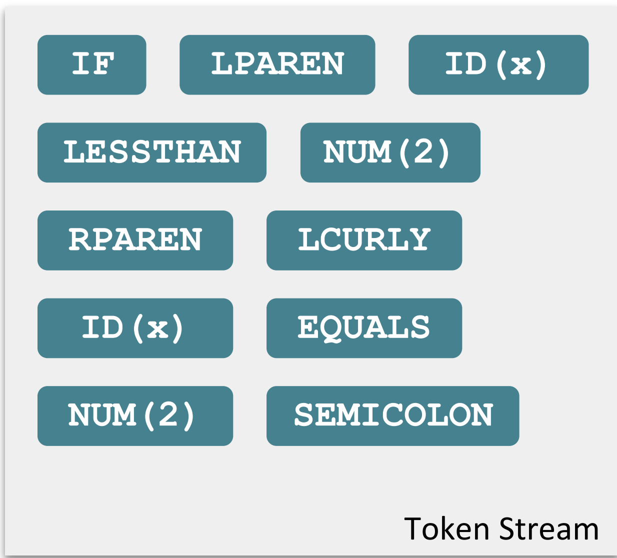
General Definition of an `if` Statement

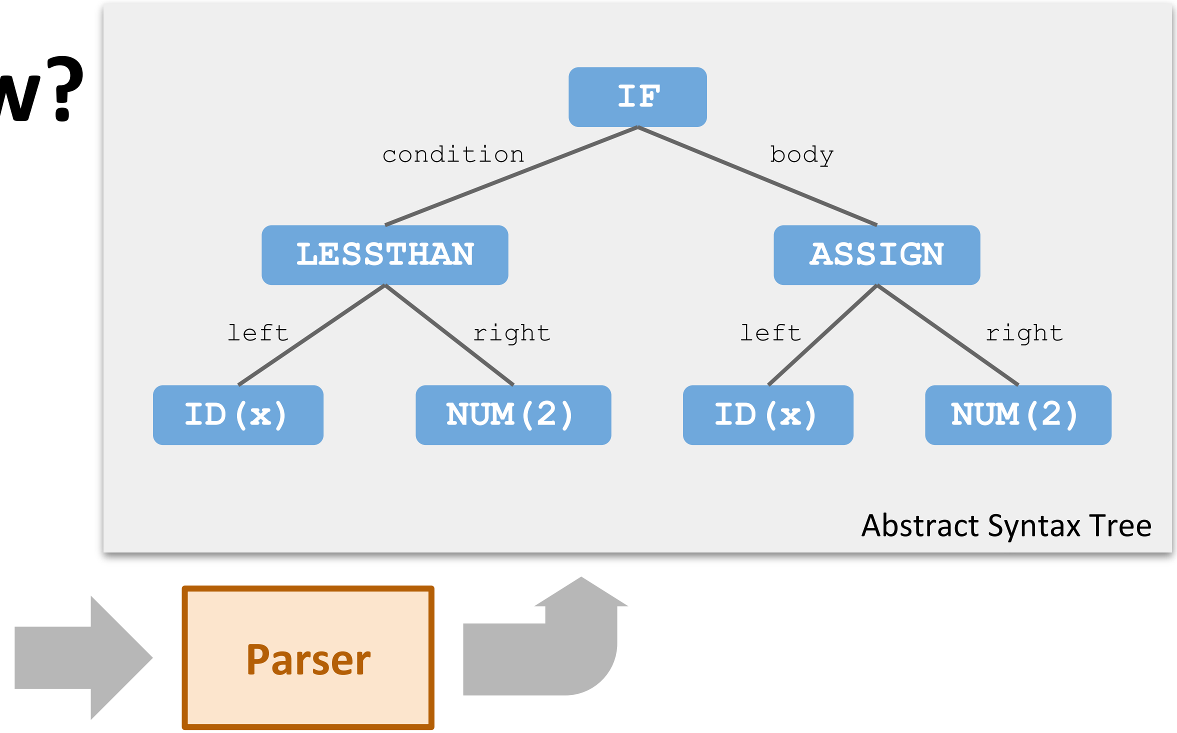
```
if ( EXPRESSION )
{
    STATEMENT
    STATEMENT
       ...
}
```

Token Stream Definition

```
IF  LPAREN
EXPRESSION  RPAREN
LCURLY  STATEMENT
STATEMENT  ...
RCURLY
```

The Parser: How?



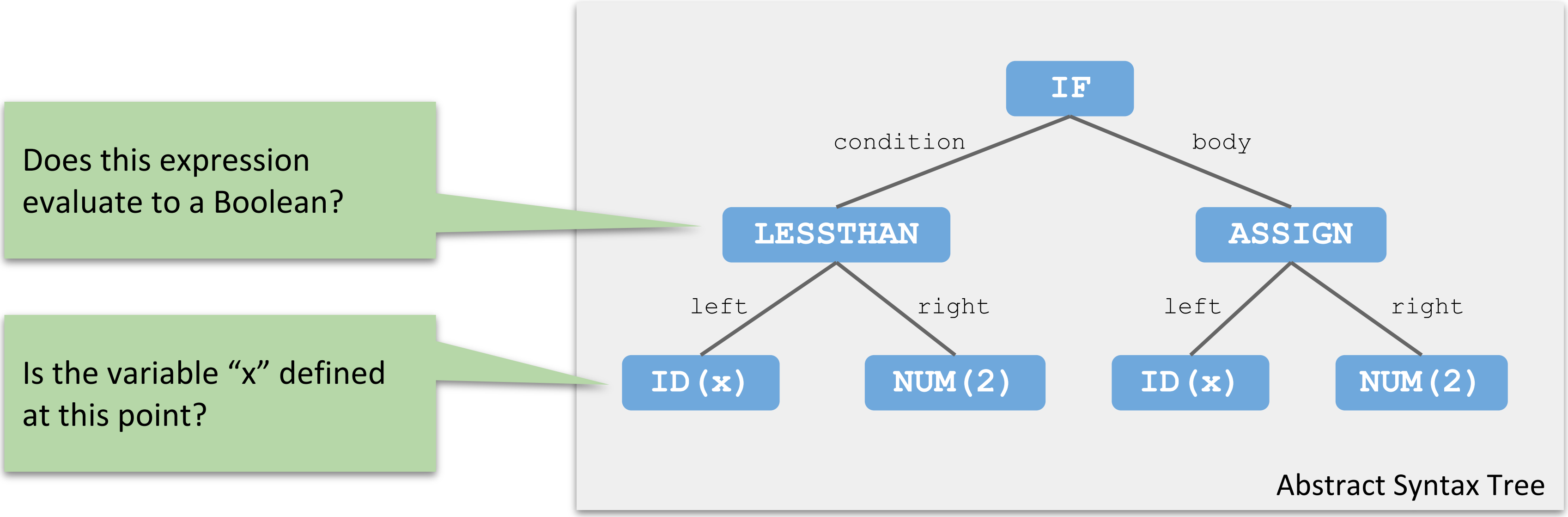


- Like scanner: single pass-through token stream, building up as we go
- Intuition: If we see IF and LPAREN , we're entering an if statement and next we must see a complete expression
  - Keep reading until we have a complete expression (recursively parse that) and attach on the condition side of the IF

Type Checking (Semantic Analysis)

- Given the abstract syntax tree, run checks over it to ensure that it fits within constraints of the language
  - Do the types match up?

- Collect additional info for code generation, such as number/type of arguments in each function



Optimization

- Code improvement: change correct code into semantically equivalent but "better" code

- Example: If something is computed every iteration of a while loop, the compiler could yank that computation out and compute it just once before entering the loop
  - Here, "better" means faster

- But requires caution: what if the value changes on each iteration of the loop?
  - "Semantically equivalent" means user sees same outcome

Code Generation

- One way to think of compiler is converting from string in source language to → its actual, abstract "meaning"

- Code generation is converting that "meaning" into a string in the destination language

- Plenty of engineering details
  - Example: if you want a stack frame/calling conventions for function calls, you have to implement them yourself via instructions generated by the compiler every time it sees a function call

Lecture Outline

- Midterm Debrief
- Introduction to the Compiler
  - Overview, Scanner, Parser
- **Project 6 Overview**
  - **Midterm Corrections, Professor Meeting Report**

Project 6 Overview

PART I:
Midterm Redo
Due in one week

- Open-note, open-tool
- Midterm grade will be the average of your score from last Thursday and your redo score
- Utilize the TAs for support!
- No late days

PART II:
Professor Meeting Report
Due in two weeks

- Cannot meet with Leslie or Eric for this assignment
- Schedule your meeting early!
- Please do not say that this is for an assignment...

Post-Lecture 14 Reminders

- What's in store for Week 8?
  - More on compilers
  - Debugging strategies
  - Project 7 Overview

- Reminders
  - Project 5 due tonight (2/17) at 11:59pm PST
  - Schedule your professor meeting ASAP!

Title

- Content