CSE 390B, 2024 Winter

Building Academic Success Through Bottom-Up Computing

Professor Meeting & Compiler Phases

Meeting with a Professor, Exploring the Compiler Phases, Project 7 Overview

Lecture Outline

- **Meeting with a Professor**
  - **How to Connect with Professors**
  - **How Connection with Professors Benefit Us**

- Exploring the Compiler Phases
  - Scanner: Process of Tokenizing an Input File
  - Parser: Making Meaning From Tokens Through ASTs
  - Type Checking, Optimization, and Code Generation

- Project 7 Overview
  - Midterm Corrections, Professor Meeting Report

Connecting with Professors

- Professors are busy but generally enthusiastic about being available to meet with students

- Channels to connect with professors:
  - Send professor an email with a request to meet
  - Meet during professor’s office hours
  - Chat with professors from community events, panels, talks, etc.

- Have questions prepared before meeting with a professor
  - Ask questions about their journey in the field, what they’ve enjoyed most, hardships they’ve faced, etc.
  - Inquiry how you may get involved with research, teaching, etc.

Benefits of Connecting with Professors

- Reaching out to your professors, TAs, and peers is a great way to discover opportunities
- Taking the time to connect with these people can open several doors
- Excellent opportunity for new perspectives and ideas for those who have been in your shoes before
- Connecting with others helps you find inspiration and build your knowledge and experience

Discussion on Professor Meeting

Take some time to think about and discuss these questions:

- Which professors are you thinking about reaching out to? Why did you choose them?
- How can you specifically benefit from connecting with professors? In your academics? Career? Personal life?
- What questions might you ask a professor if you had an upcoming meeting scheduled with one?

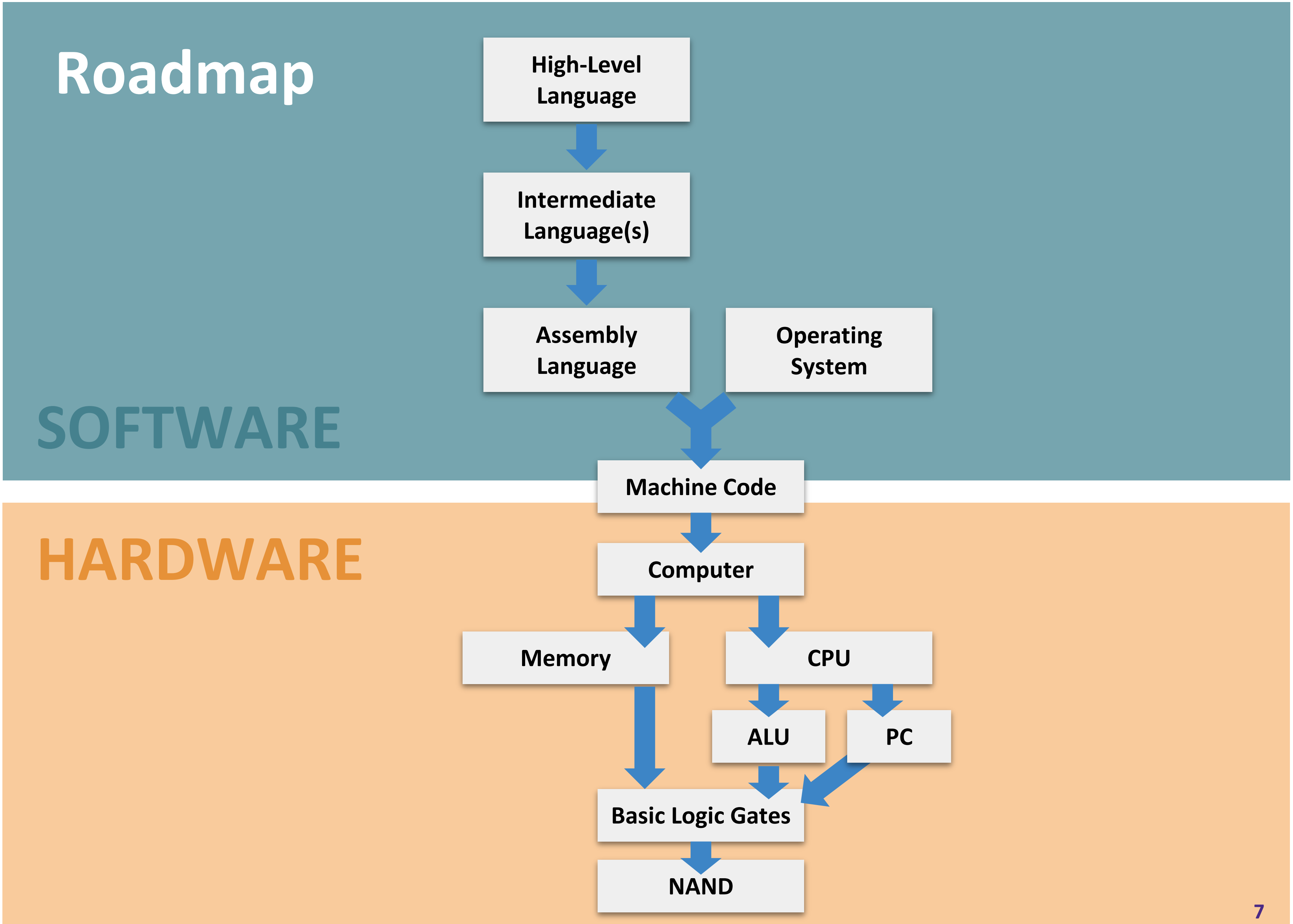
Lecture Outline

- Meeting with a Professor
  - How to Connect with Professors
  - How Connection with Professors Benefit Us

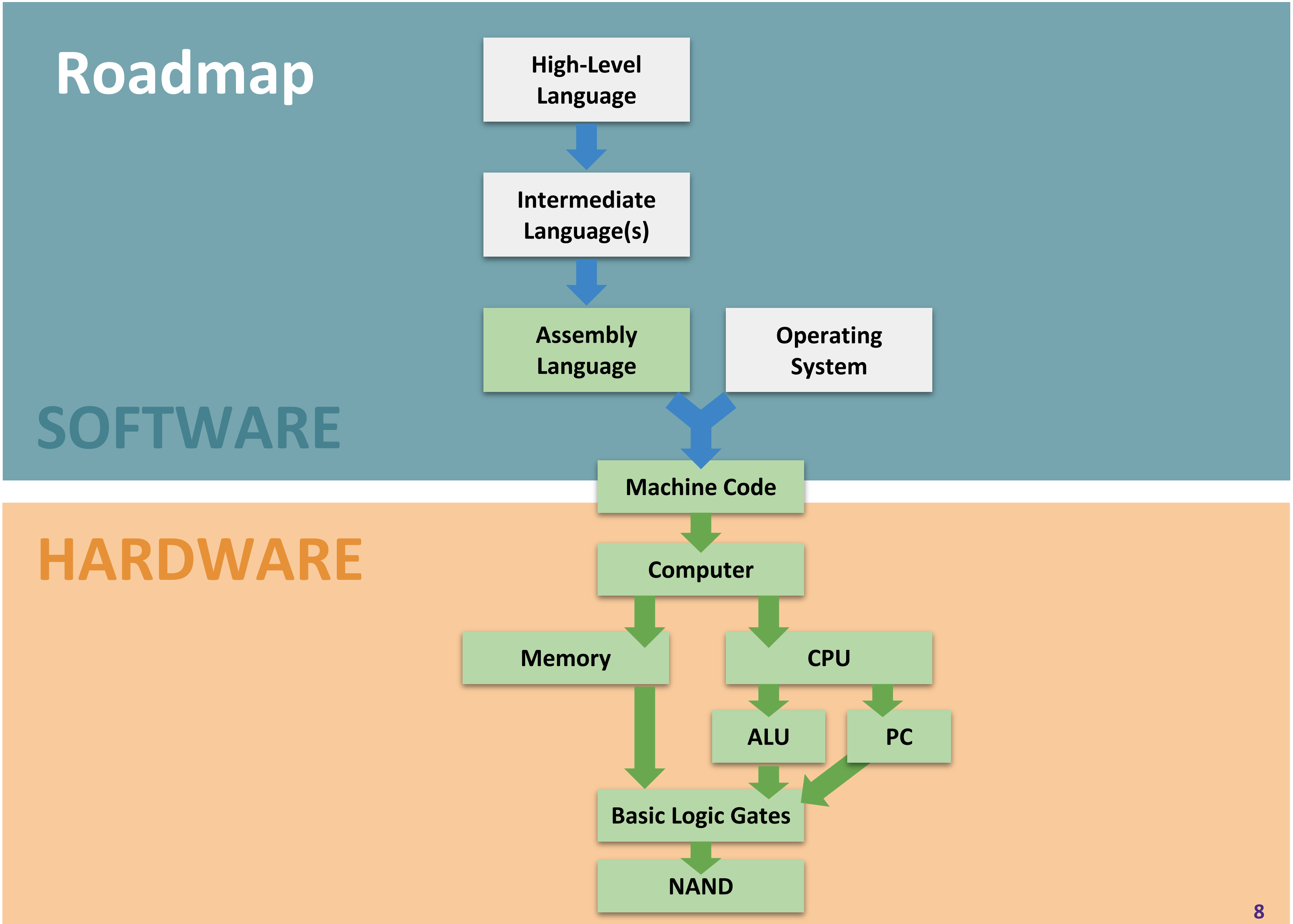
- **Exploring the Compiler Phases**
  - **Scanner: Process of Tokenizing an Input File**
  - Parser: Making Meaning From Tokens Through ASTs
  - Type Checking, Optimization, and Code Generation

- Project 7 Overview
  - Midterm Corrections, Professor Meeting Report

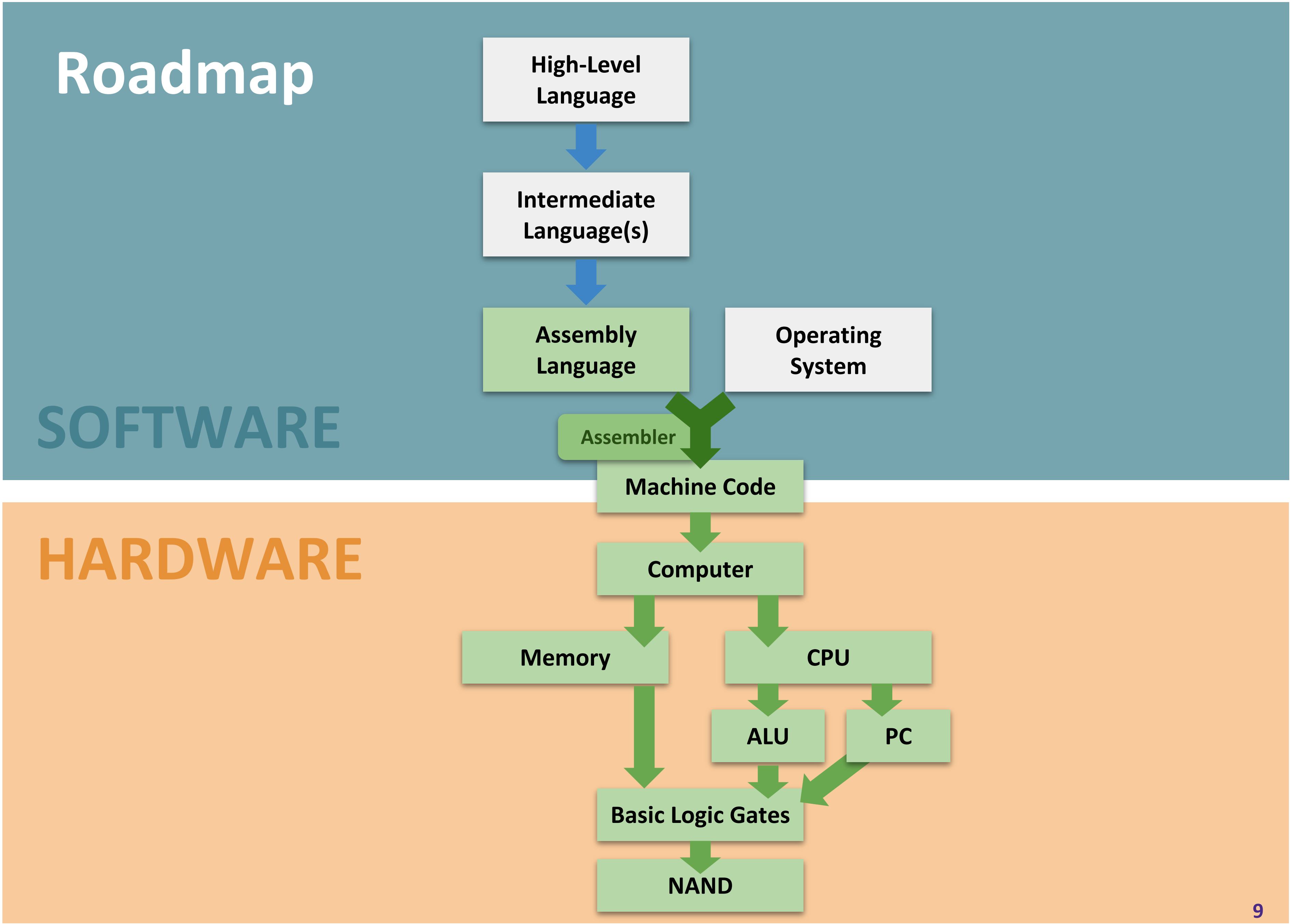
Roadmap


SOFTWARE
High-Level Language
Intermediate Language(s)
Assembly Language
Operating System
Machine Code
HARDWARE
Computer
Memory
CPU
ALU
PC
Basic Logic Gates
NAND

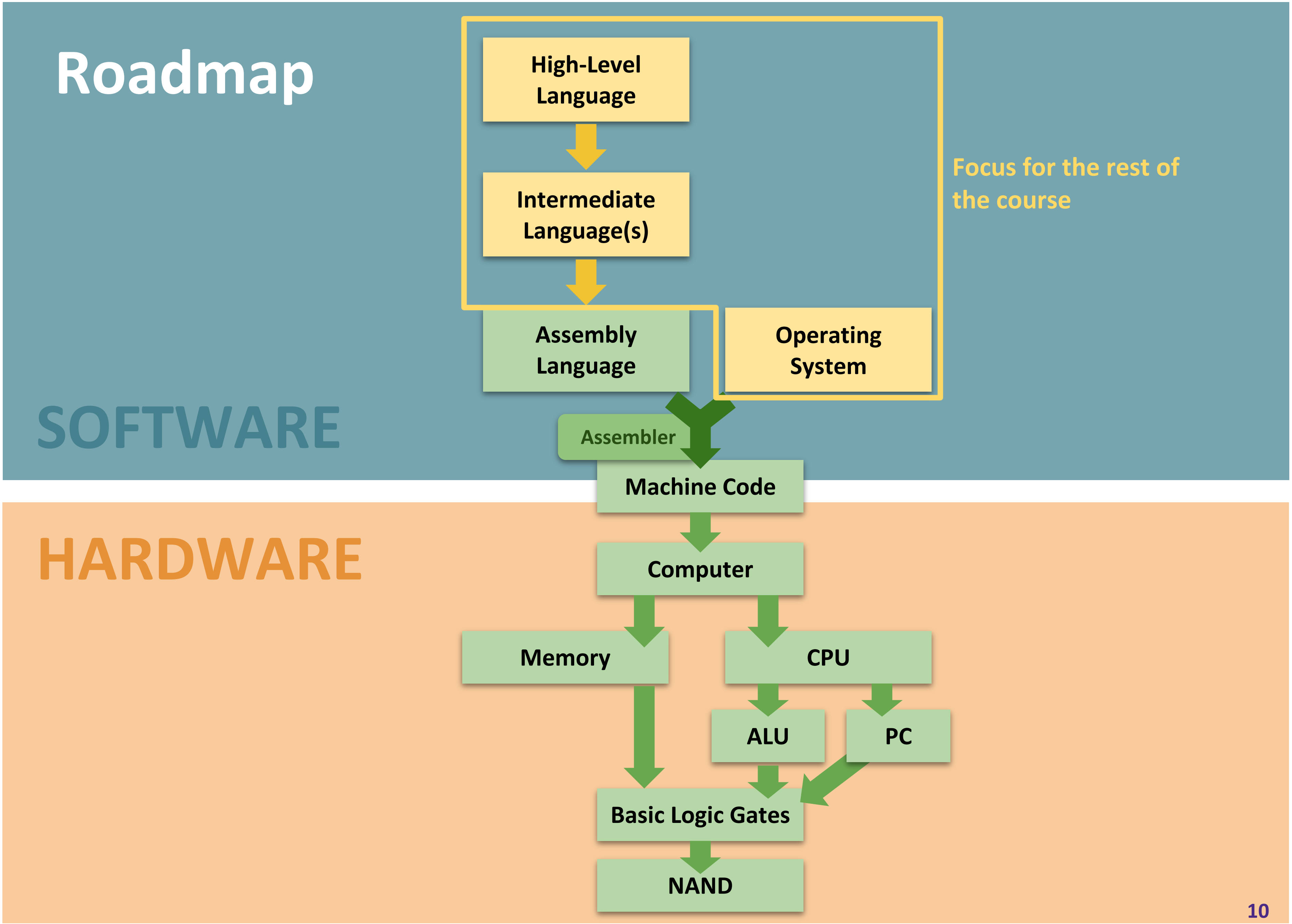
Roadmap


High-Level Language
Intermediate Language(s)
Assembly Language
Operating System
SOFTWARE
Machine Code
HARDWARE
Computer
Memory
CPU
ALU
PC
Basic Logic Gates
NAND

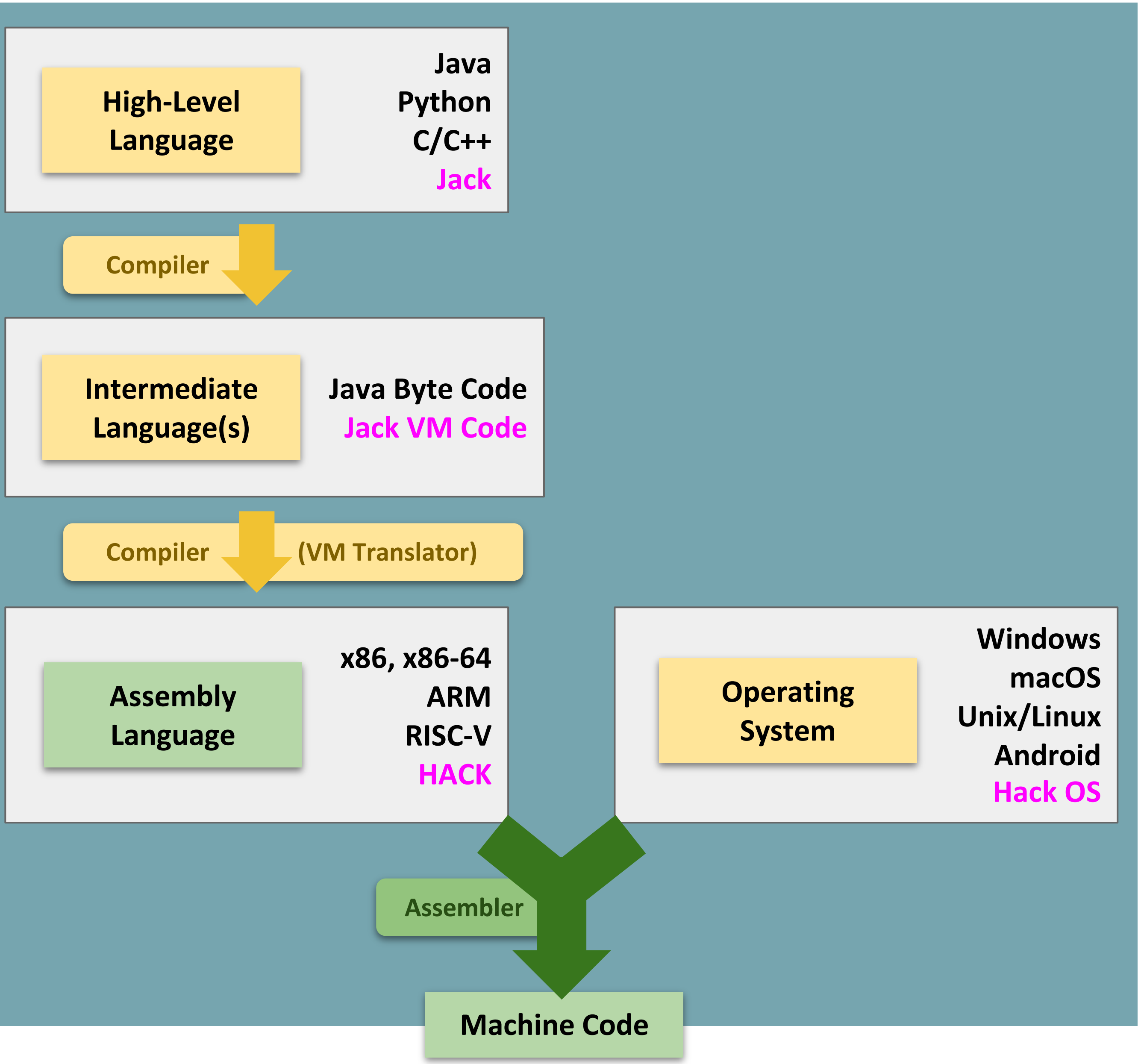
Roadmap


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

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



High-Level Language

Java
Python
C/C++
Jack

Compiler

Intermediate Language(s)

Java Byte Code
Jack VM Code

Compiler (VM Translator)

Assembly Language

x86, x86-64
ARM
RISC-V
HACK

Operating System

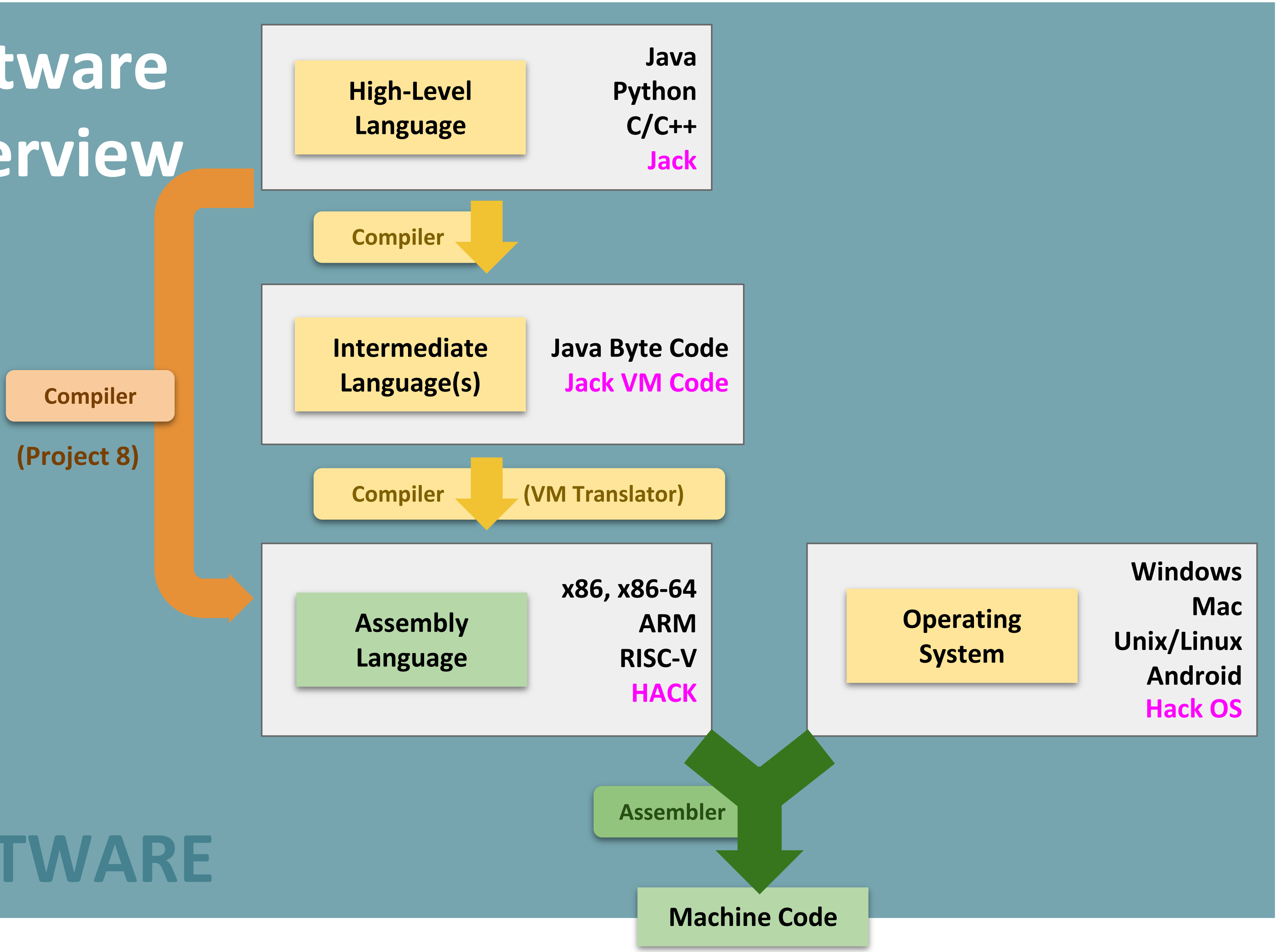
Windows
macOS
Unix/Linux
Android
Hack OS

Assembler

Machine Code

SOFTWARE

Software Overview



High-Level Language

Java
Python
C/C++
Jack

Compiler

Intermediate Language(s)

Java Byte Code
Jack VM Code

Compiler (VM Translator)

Compiler

(Project 8)

Assembly Language

x86, x86-64
ARM
RISC-V
HACK

Operating System

Windows
Mac
Unix/Linux
Android
Hack OS

Assembler

Machine Code

SOFTWARE

The Compiler: Goal

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

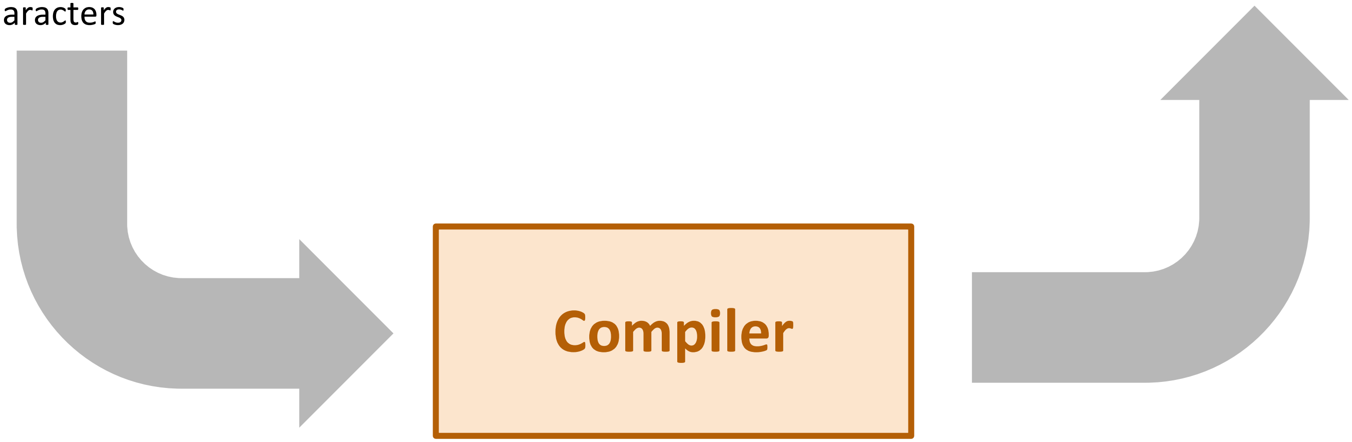
High-Level Language

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



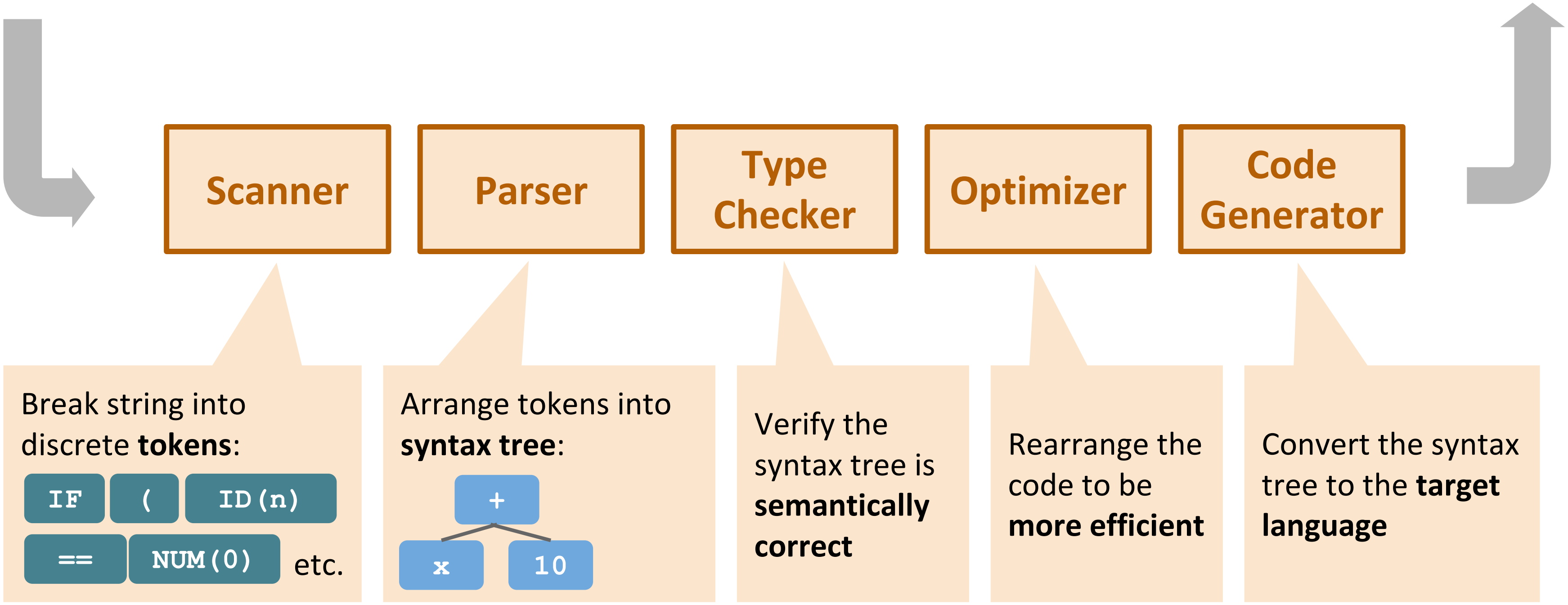
The Compiler: Implementation

```
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



Break string into discrete **tokens**: IF (ID(n) == NUM(0) etc.

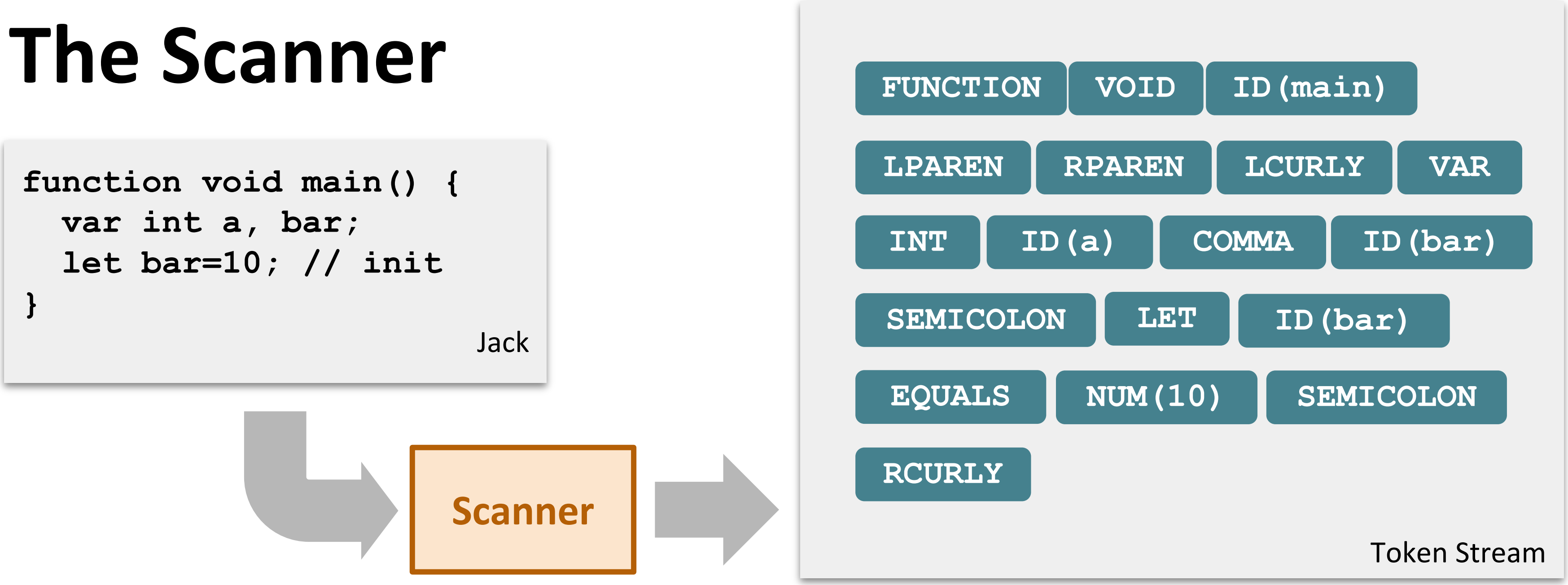
Arrange tokens into **syntax tree**:

Verify the syntax tree is **semantically correct**

Rearrange the code to be **more efficient**

Convert the syntax tree to the **target language**

The Scanner



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

Jack

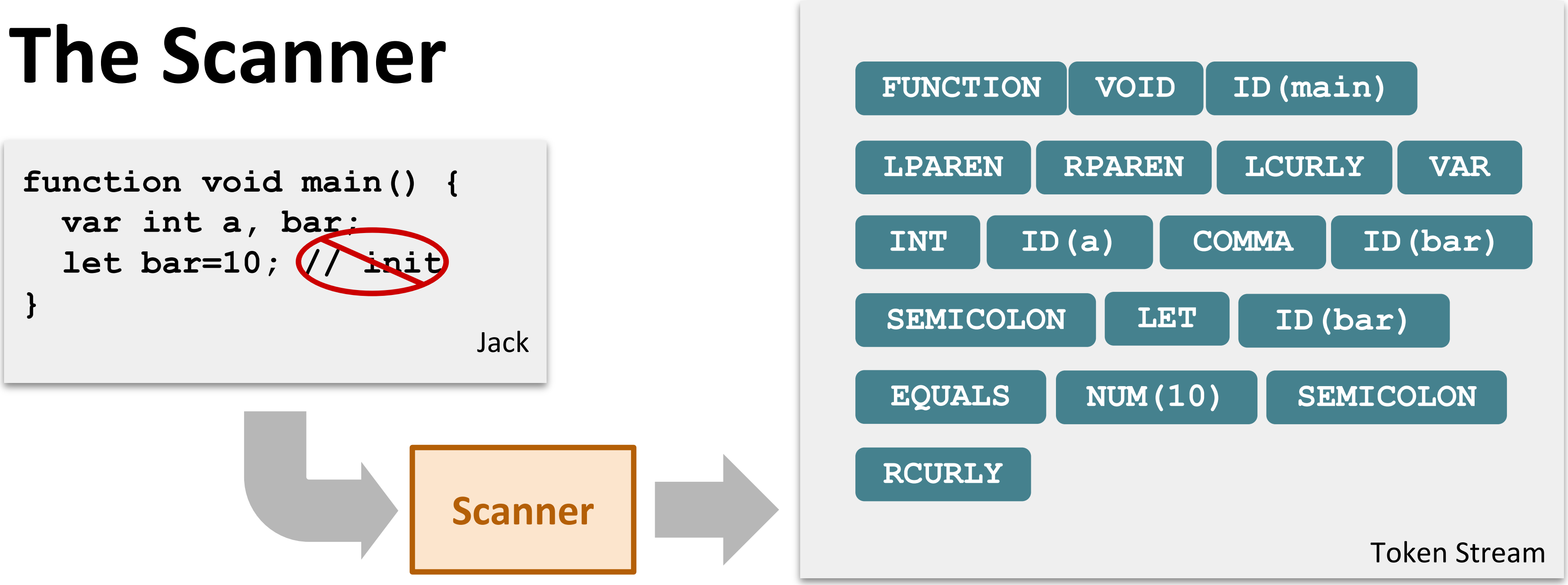
Scanner

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), and punctuation (e.g., SEMICOLON or COMMA)

The Scanner



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

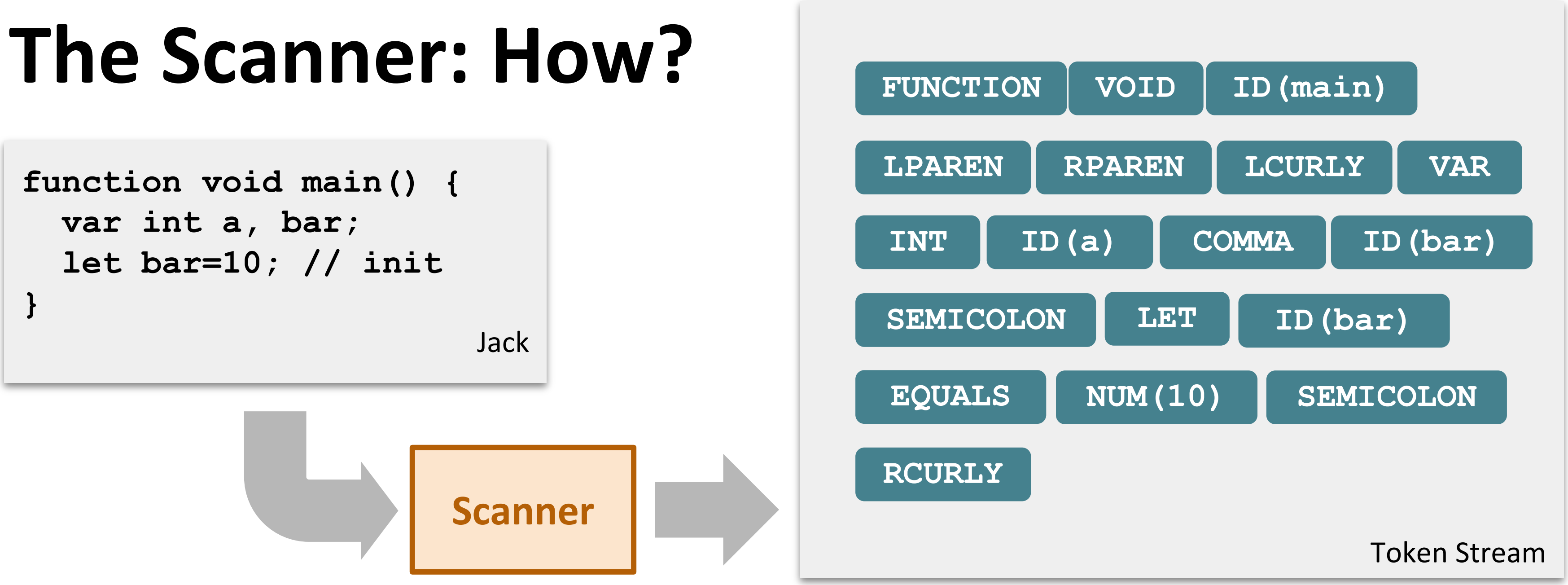
Jack

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

- 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: How?



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

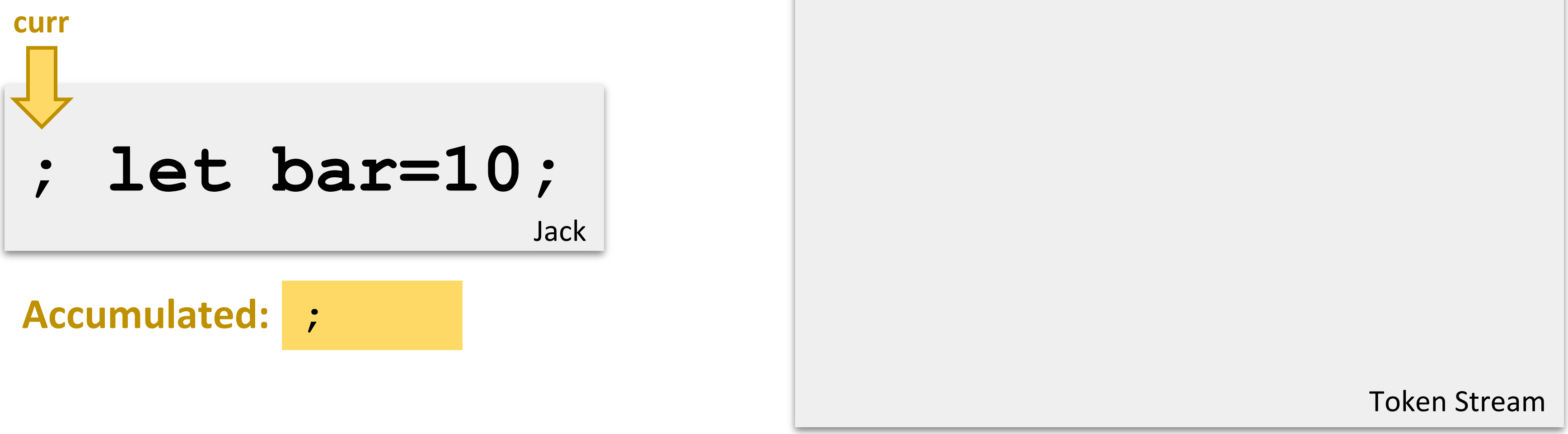
Jack

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

- 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

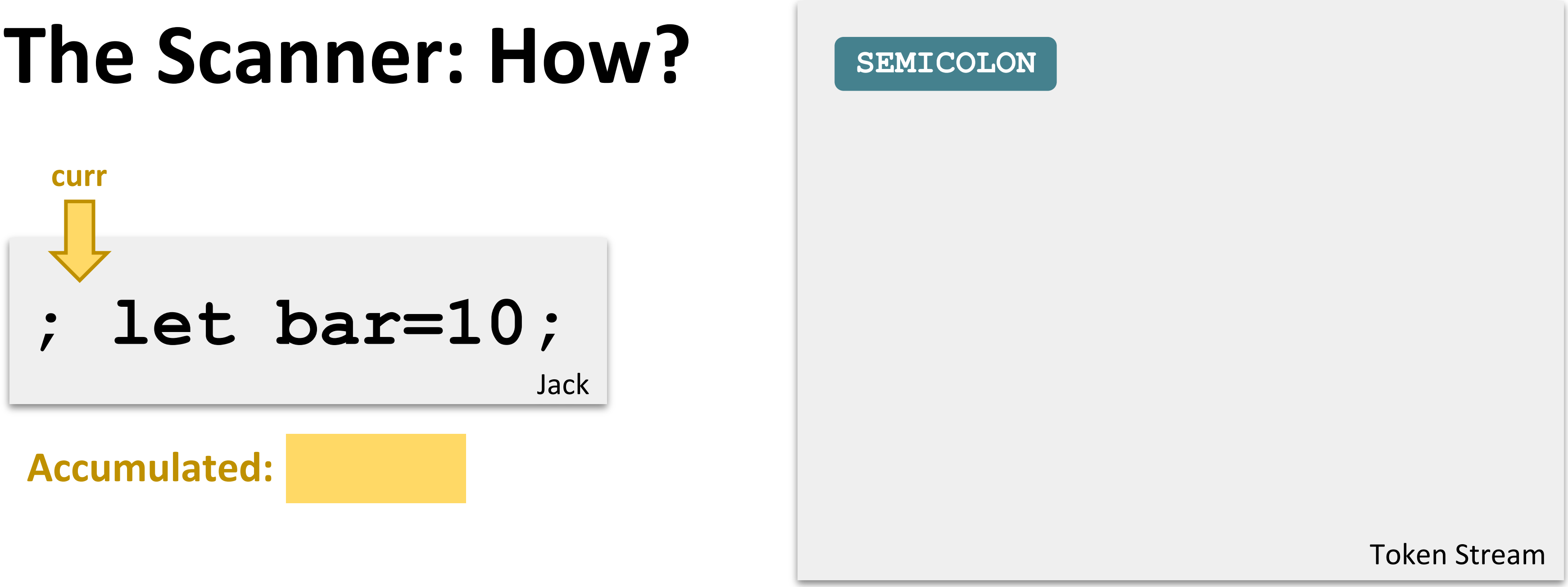
The Scanner: How?



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

- How can we take a line of code in Jack and convert this into a token stream?
    - 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

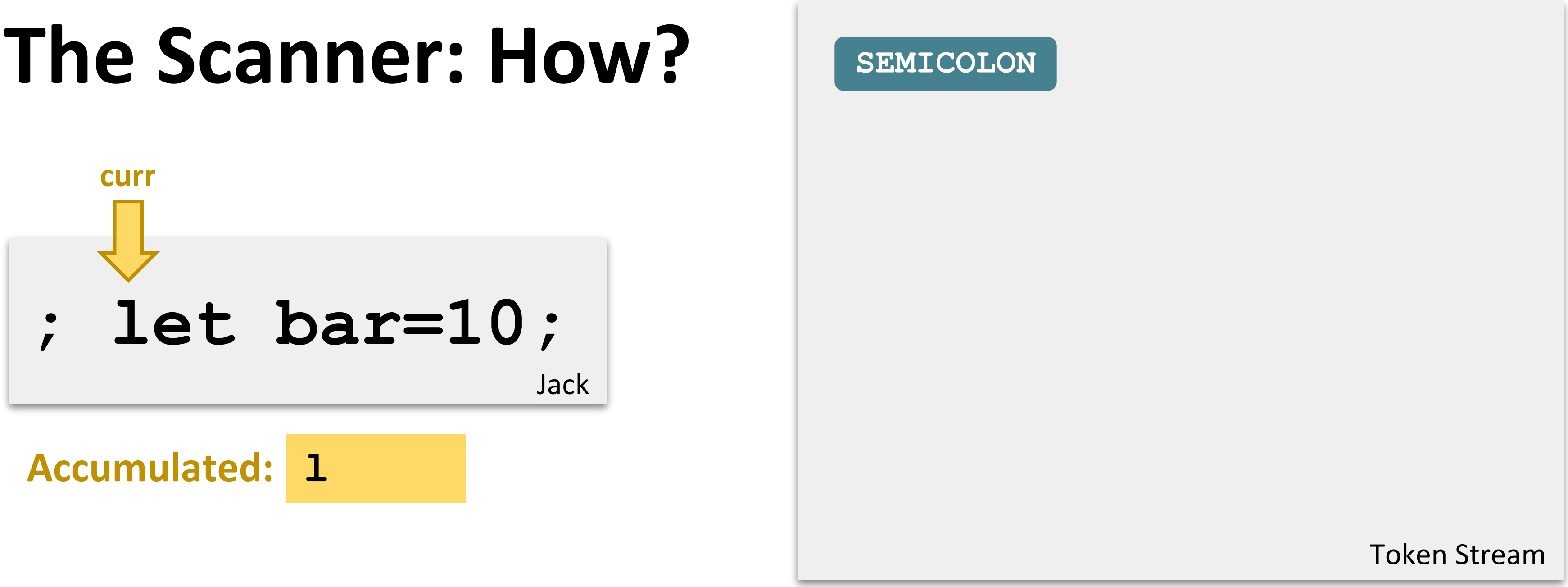
The Scanner: How?



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

- How can we take a line of code in Jack and convert this into a token stream?
  - 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

The Scanner: How?



curr

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

Jack

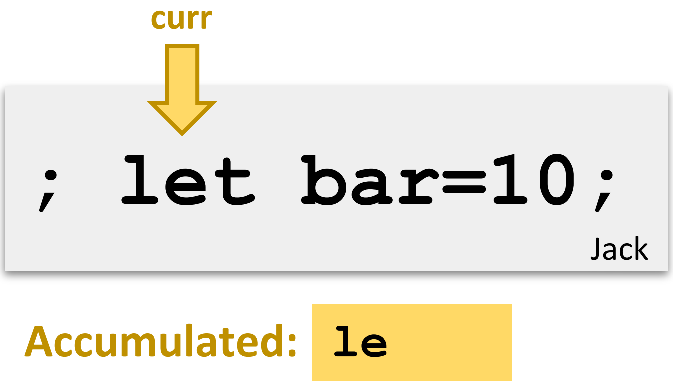
Accumulated: 1

SEMICOLON

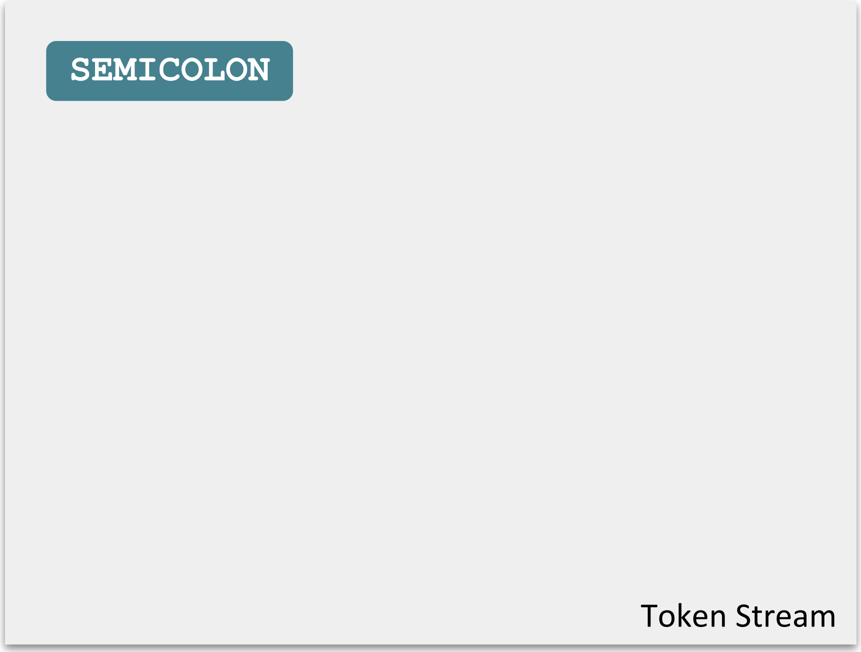
Token Stream

- How can we take a line of code in Jack and convert this into a token stream?
  - 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

The Scanner: How?

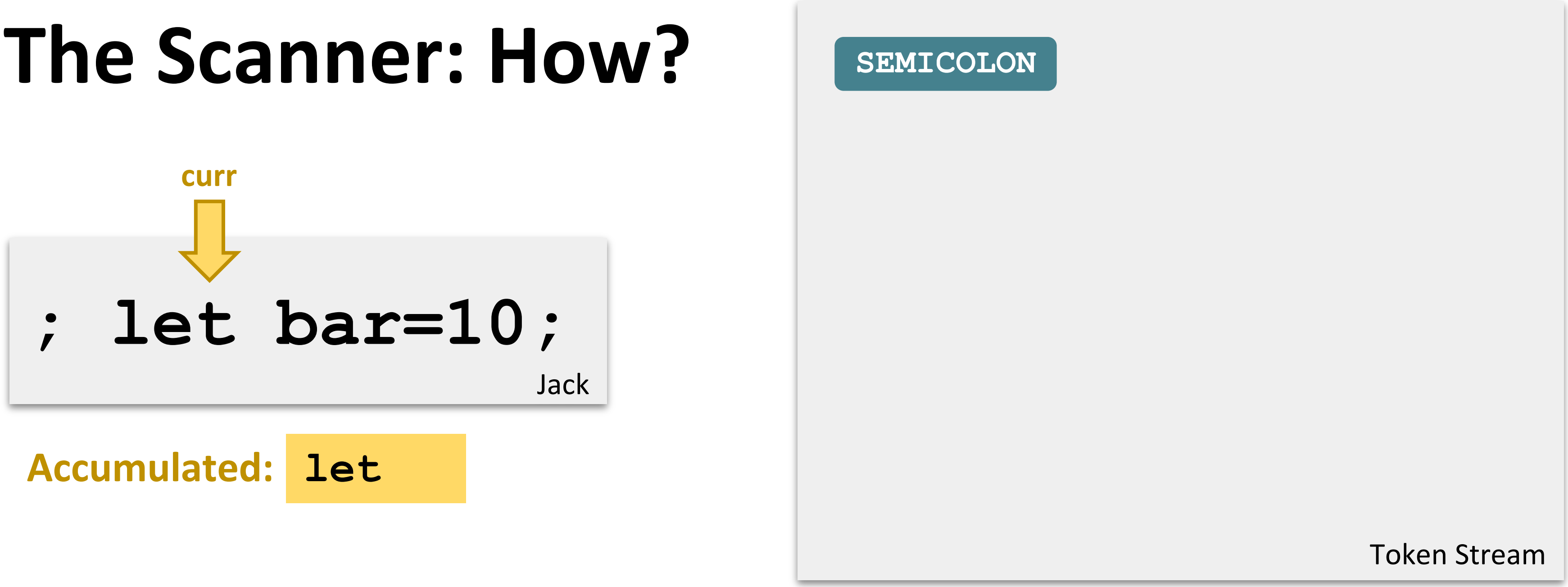


```
; let bar=10;
```



- How can we take a line of code in Jack and convert this into a token stream?
  - 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

The Scanner: How?



curr

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

Jack

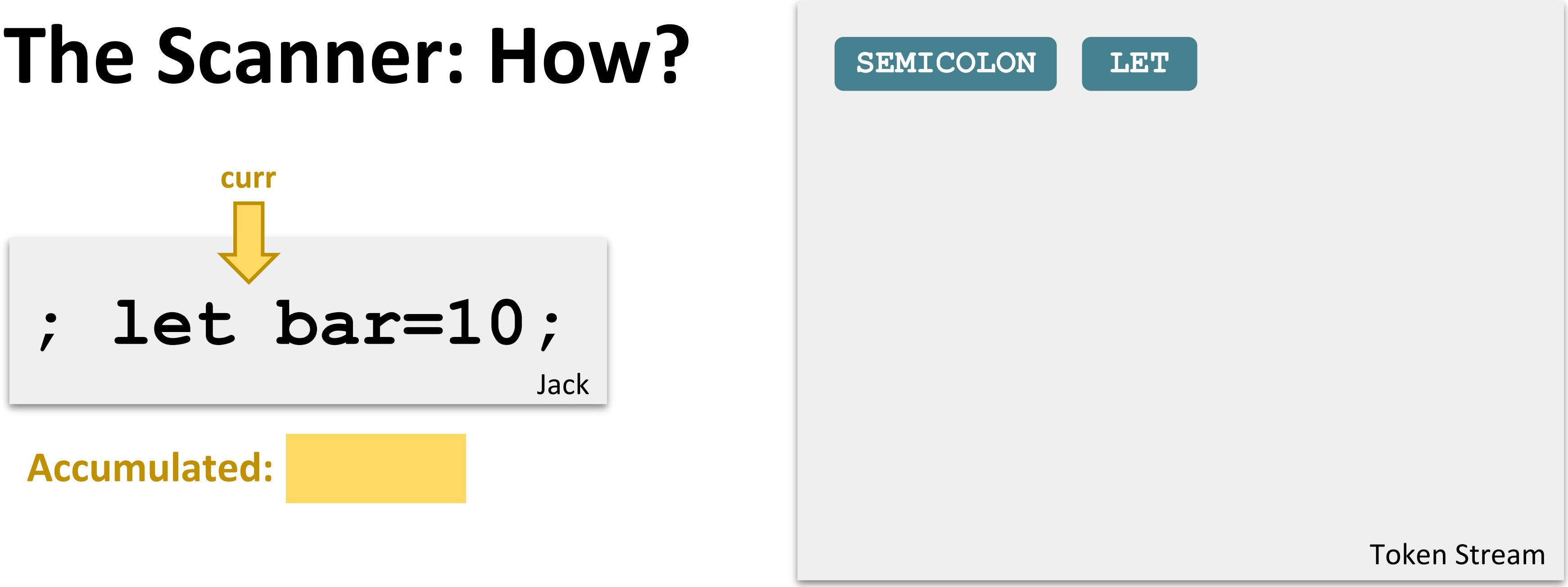
Accumulated: `let`

SEMICOLON

Token Stream

- How can we take a line of code in Jack and convert this into a token stream?
  - 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

The Scanner: How?



curr

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

Jack

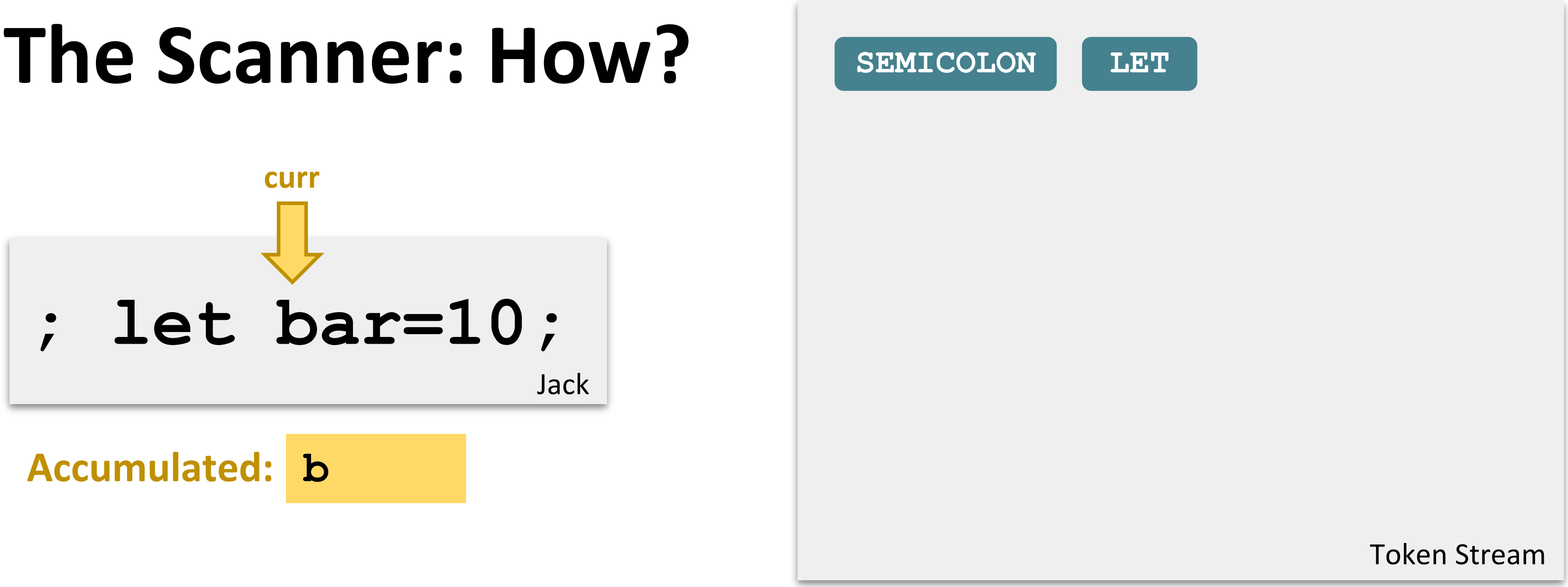
Accumulated:

SEMICOLON LET

Token Stream

- How can we take a line of code in Jack and convert this into a token stream?
  - 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

The Scanner: How?



curr

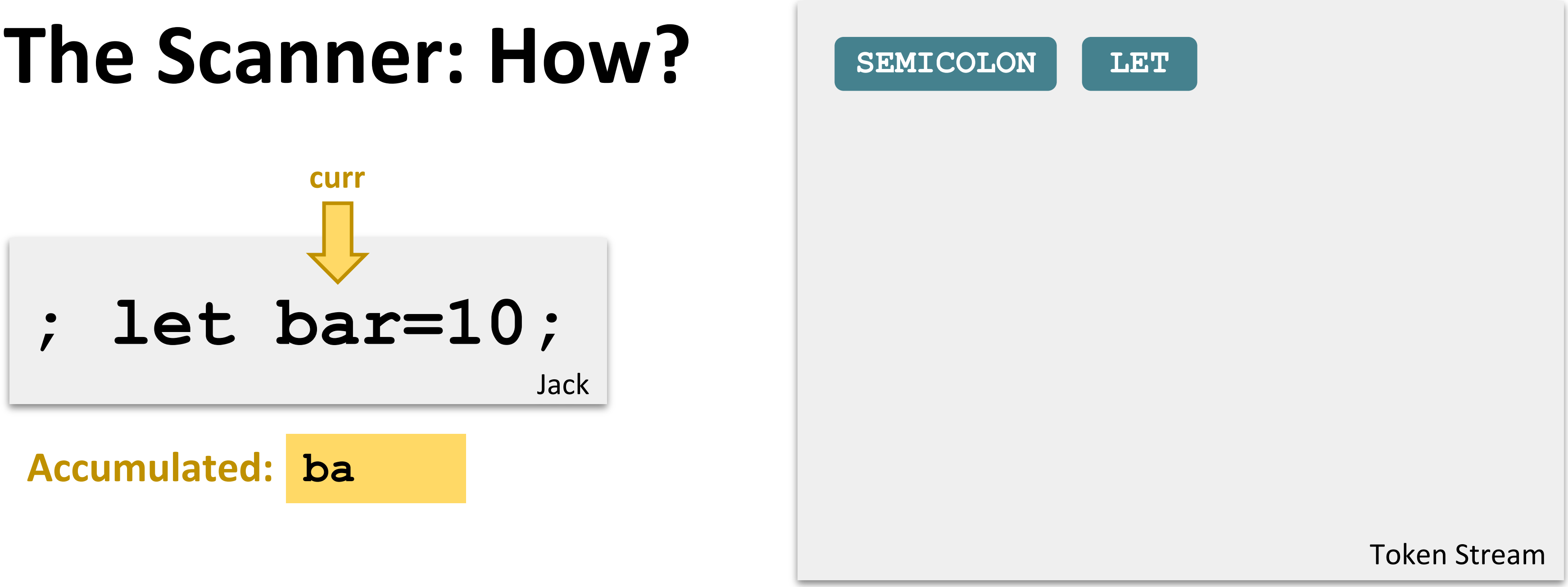
```
; let bar=10;
```

Jack

Accumulated: `b`

- How can we take a line of code in Jack and convert this into a token stream?
    - 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

The Scanner: How?



curr

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

Jack

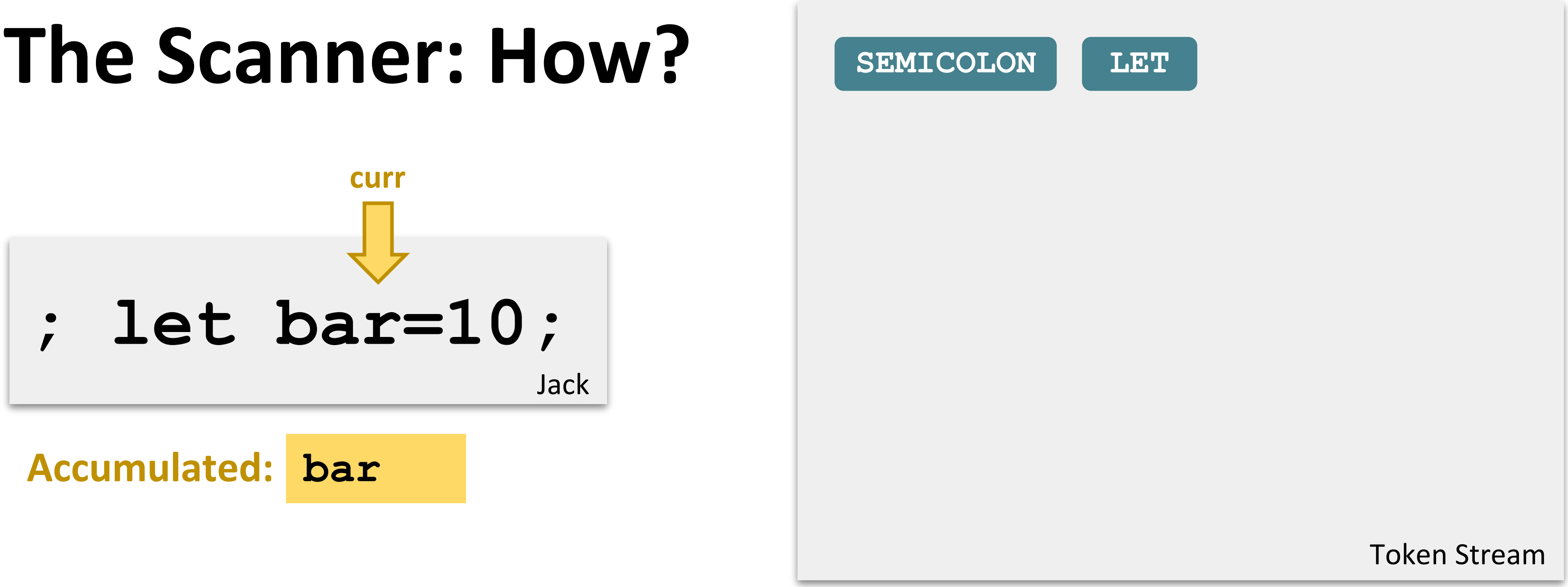
Accumulated: ba

SEMICOLON LET

Token Stream

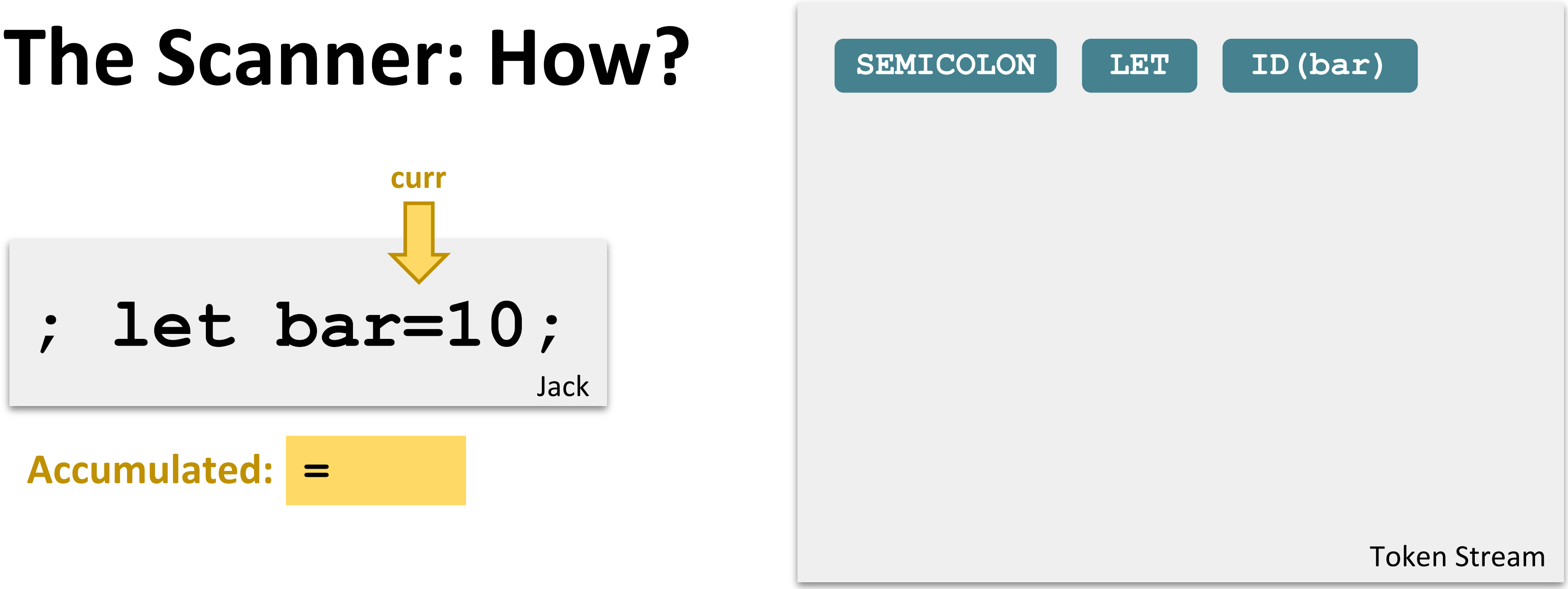
- How can we take a line of code in Jack and convert this into a token stream?
  - Keep cursor on current char
  - Break off a token when we complete one
  - If the next char could be part of this token, accumulate it

The Scanner: How?



- How can we take a line of code in Jack and convert this into a token stream?
  - 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

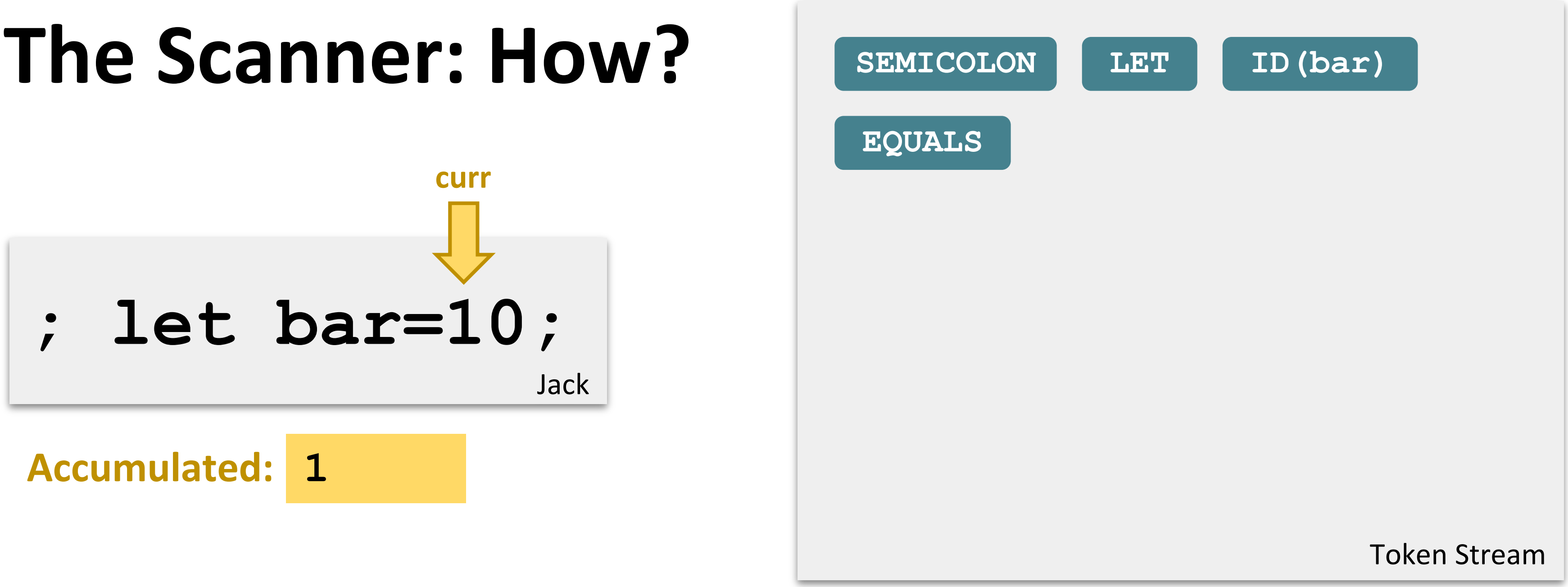
The Scanner: How?



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

- How can we take a line of code in Jack and convert this into a token stream?
  - 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

The Scanner: How?



curr

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

Jack

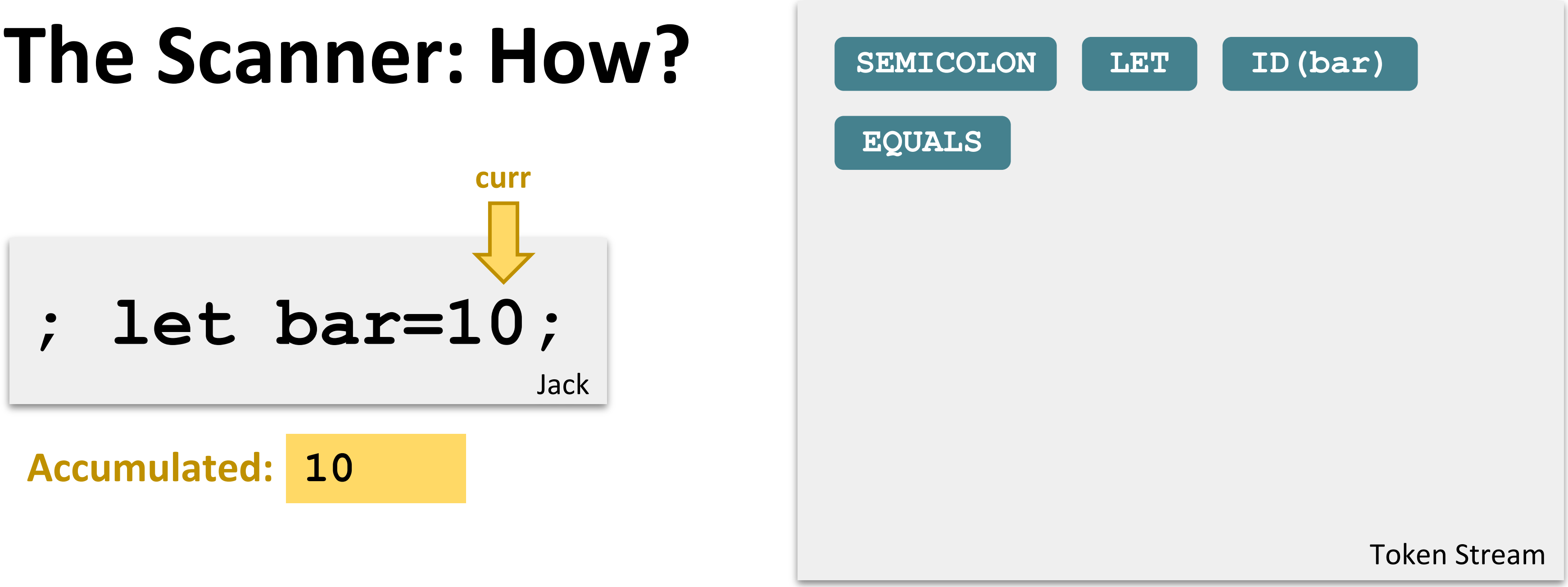
Accumulated: `1`

`SEMICOLON` `LET` `ID(bar)` `EQUALS`

Token Stream

- How can we take a line of code in Jack and convert this into a token stream?
  - 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

The Scanner: How?



curr

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

Jack

Accumulated: 10

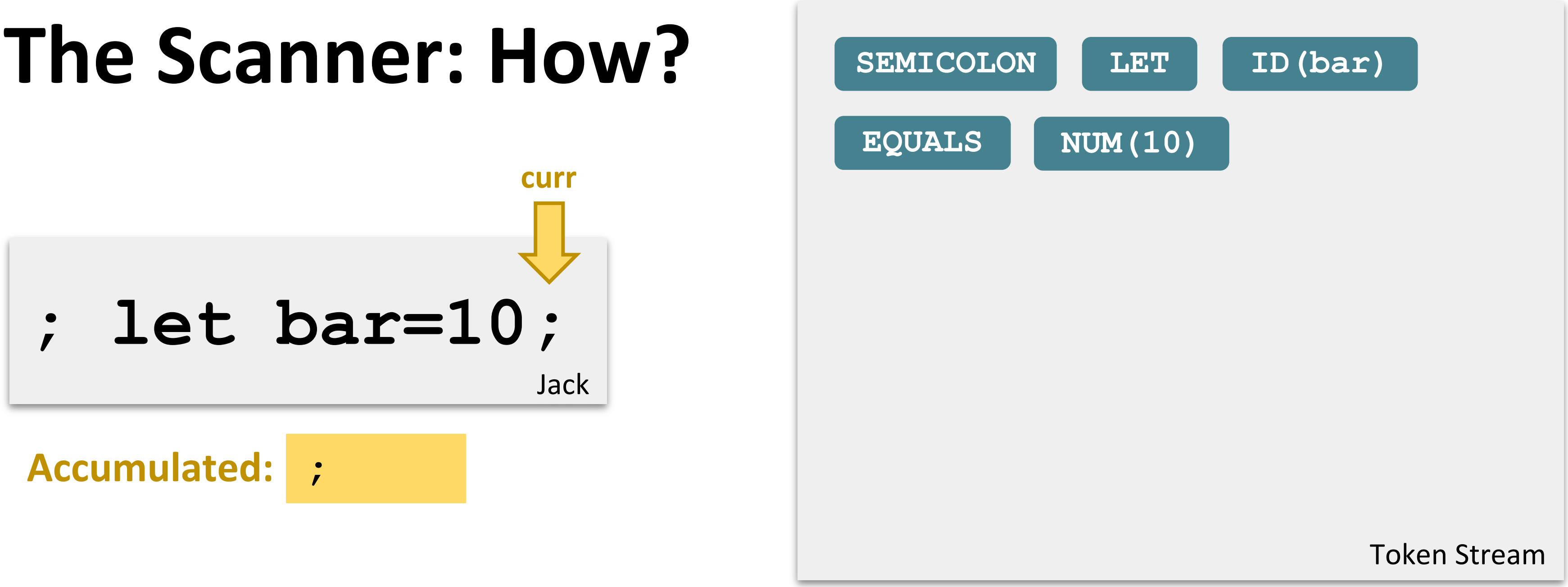
SEMICOLON LET ID(bar)

EQUALS

Token Stream

- How can we take a line of code in Jack and convert this into a token stream?
  - 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

The Scanner: How?



curr

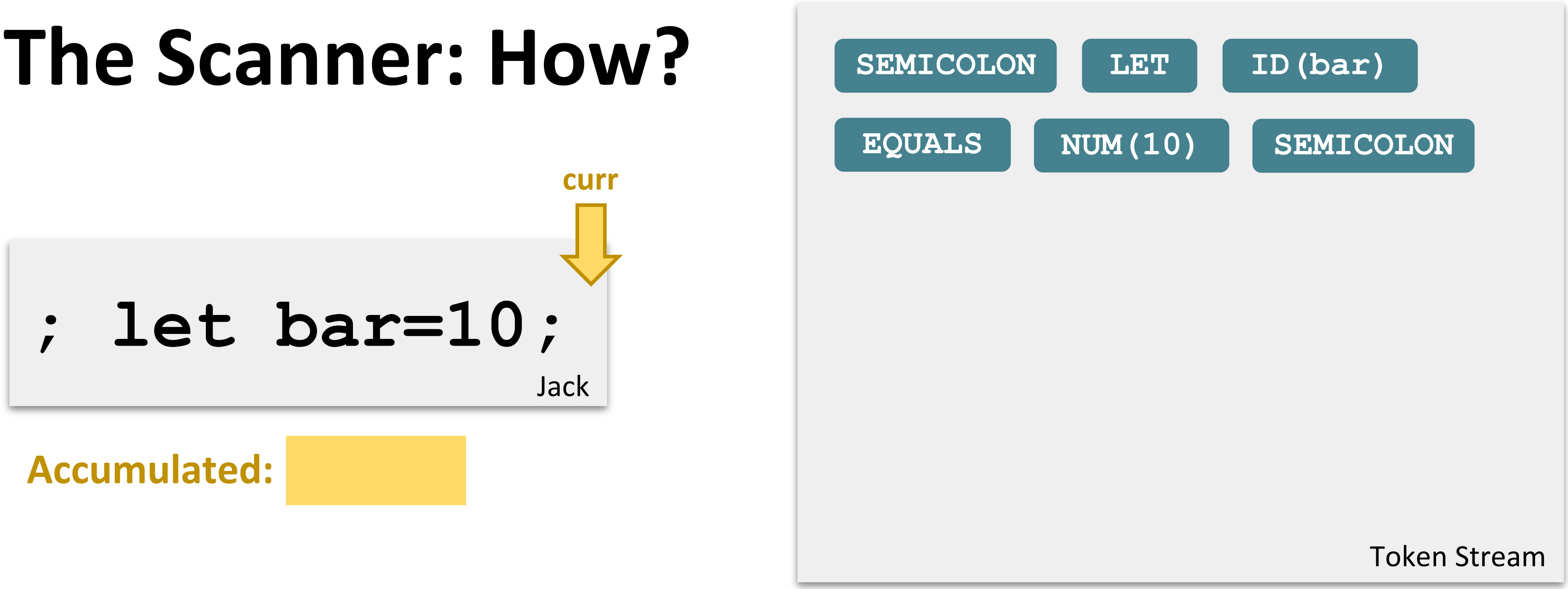
```
; let bar=10;
```

Jack

Accumulated: ;

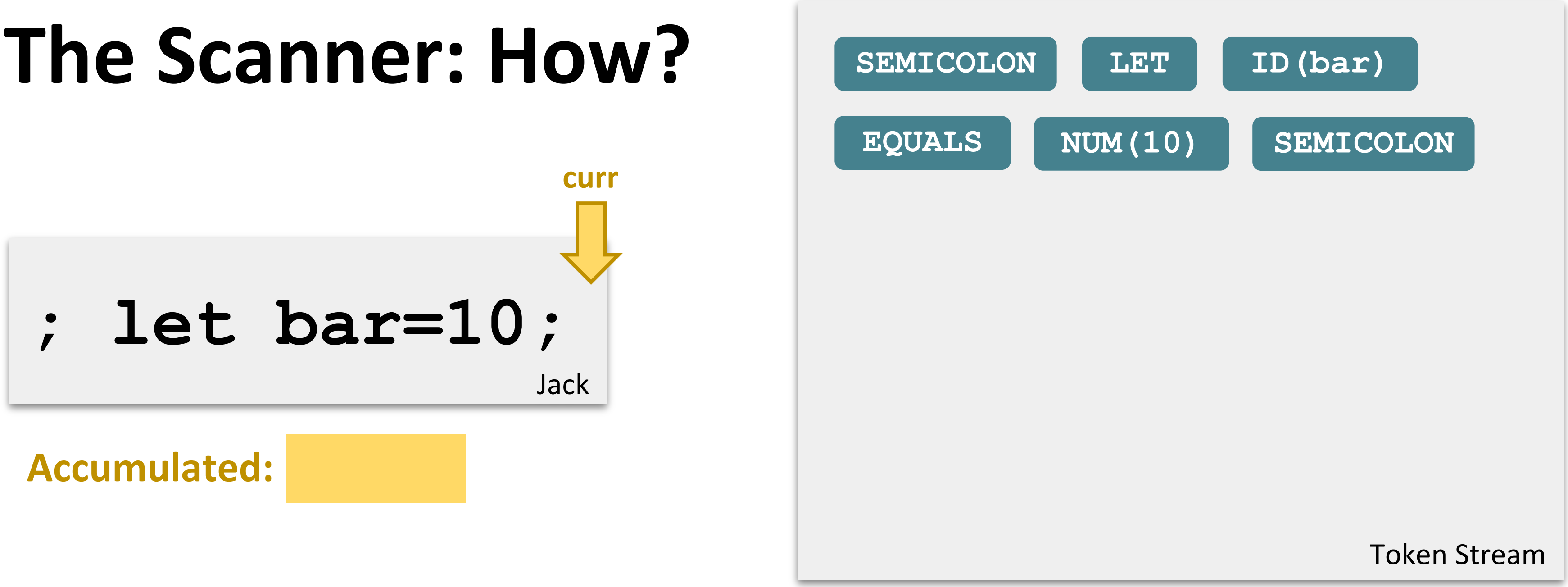
- How can we take a line of code in Jack and convert this into a token stream?
  - Keep cursor on current char
  - Break off a token when we complete one
  - If the next char could be part of this token, accumulate it

The Scanner: How?



- How can we take a line of code in Jack and convert this into a token stream?
  - 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

The Scanner: How?



curr

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

Jack

Accumulated:

SEMICOLON LET ID(bar)

EQUALS NUM(10) SEMICOLON

Token Stream

- How to distinguish built-in keywords (e.g., “let”) from identifiers (e.g., “bar”)?
  - 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
  - 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

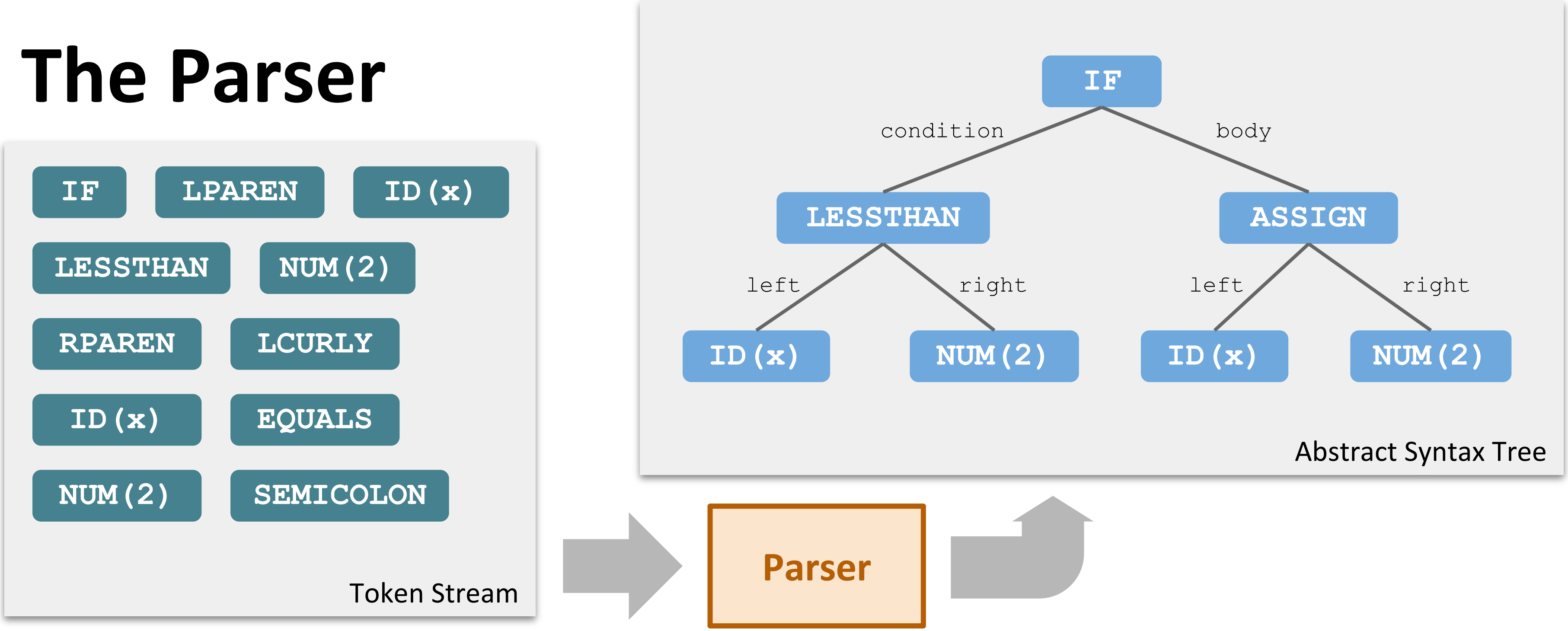
Lecture Outline

- Meeting with a Professor
  - How to Connect with Professors
  - How Connection with Professors Benefit Us

- **Exploring the Compiler Phases**
  - Scanner: Process of Tokenizing an Input File
  - **Parser: Making Meaning From Tokens Through ASTs**
  - Type Checking, Optimization, and Code Generation

- Project 7 Overview
  - Midterm Corrections, Professor Meeting Report

The Parser



IF LPAREN ID(x) LESSTHAN NUM(2) RPAREN LCURLY ID(x) EQUALS NUM(2) SEMICOLON

Token Stream

Parser

IF

condition: LESSTHAN (left: ID(x), right: NUM(2))

body: ASSIGN (left: ID(x), right: NUM(2))

Abstract Syntax Tree

- 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

- Assignment Statement

```
x = y;
```

- If Statement

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

Expressions

Evaluate to a result

- Operators

```
x == 0;
```

- Variable

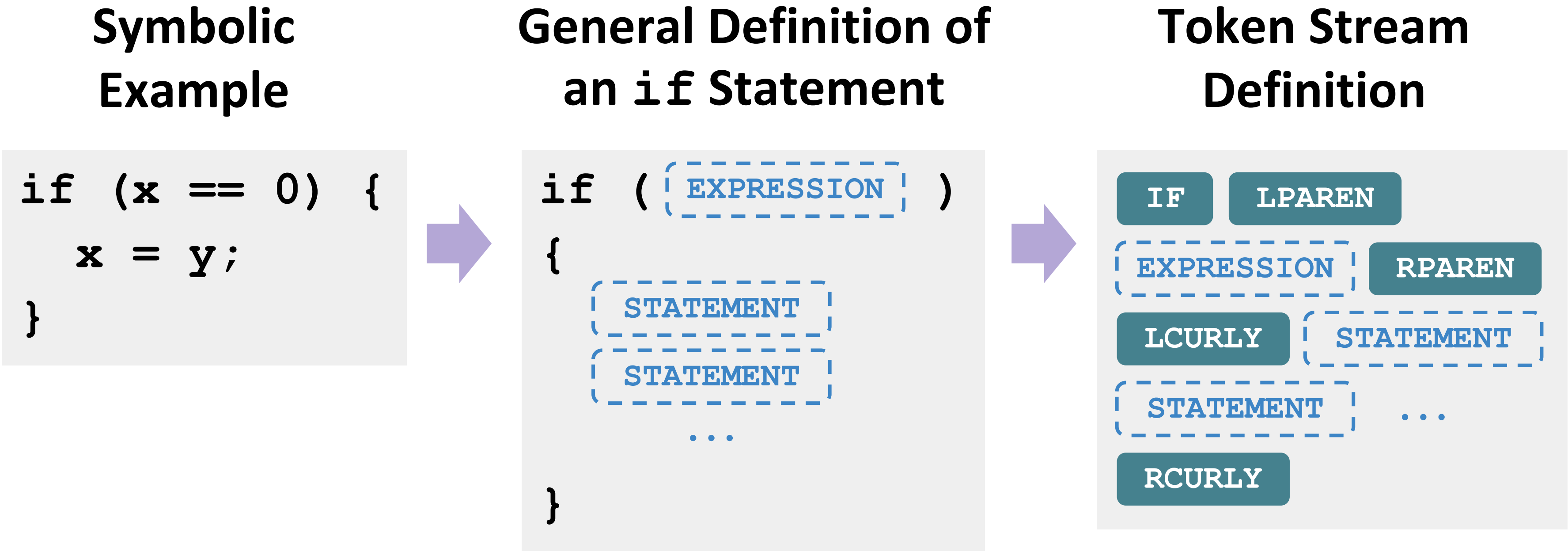
```
x
```

- 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;
}
```

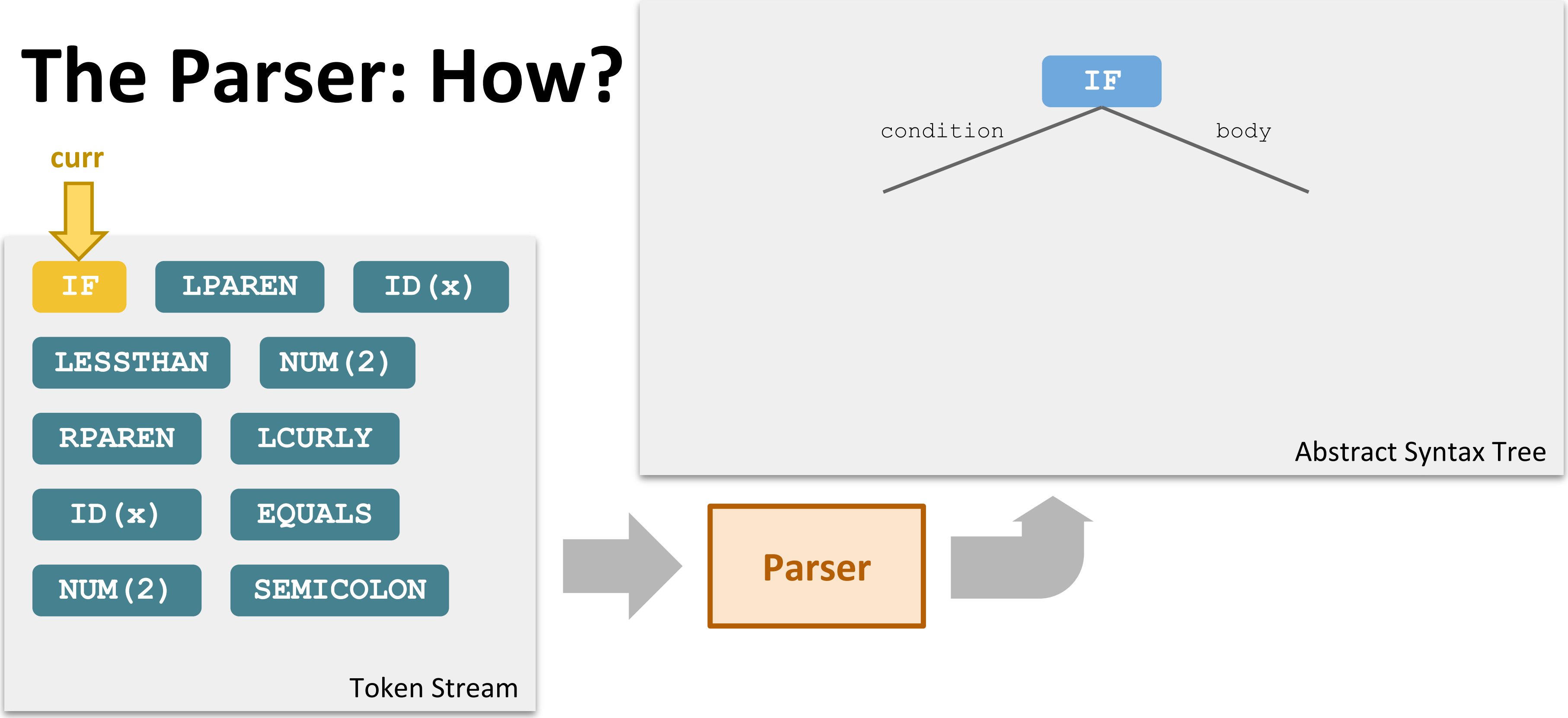
General Definition of an `if` Statement

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

Token Stream Definition

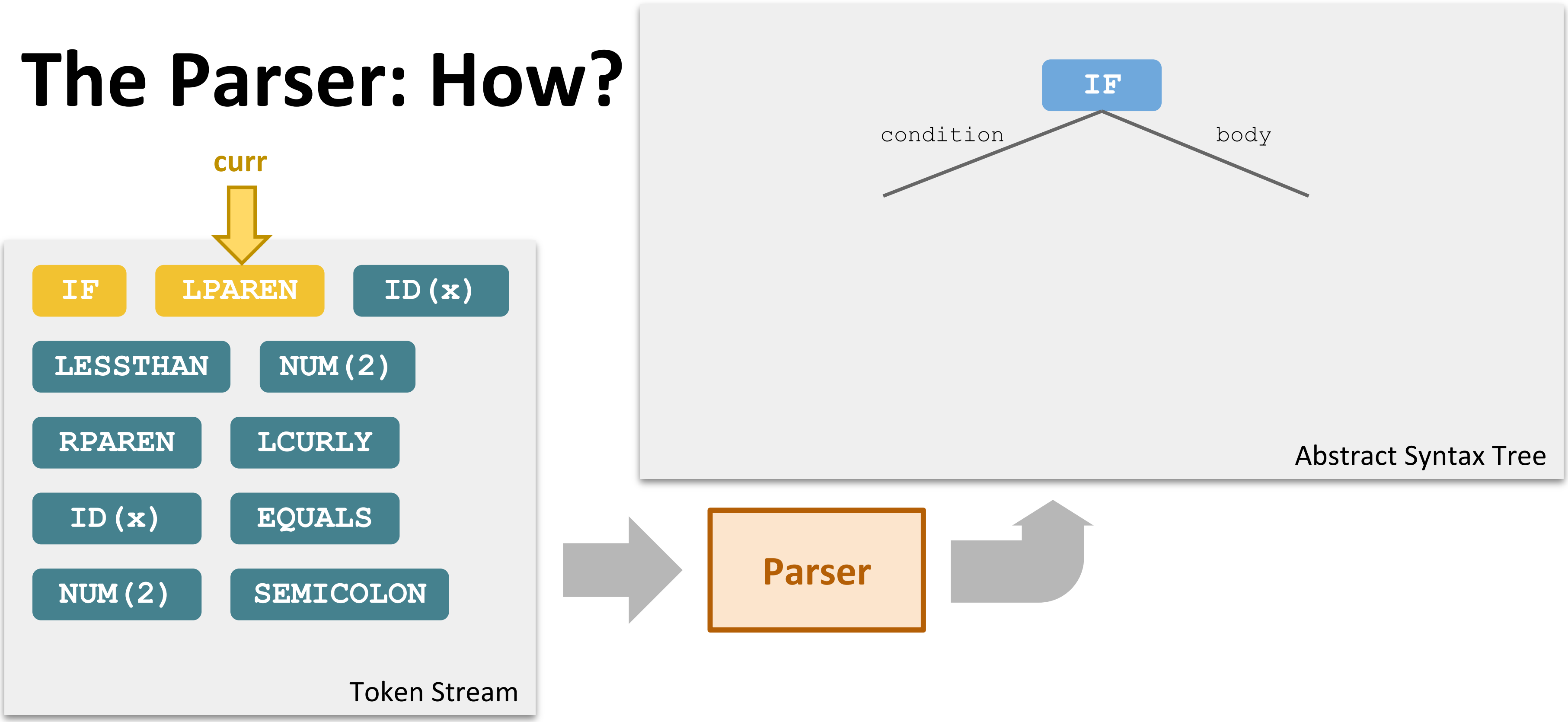
IF LPAREN EXPRESSION RPAREN LCURLY STATEMENT STATEMENT ... RCURLY

The Parser: How?



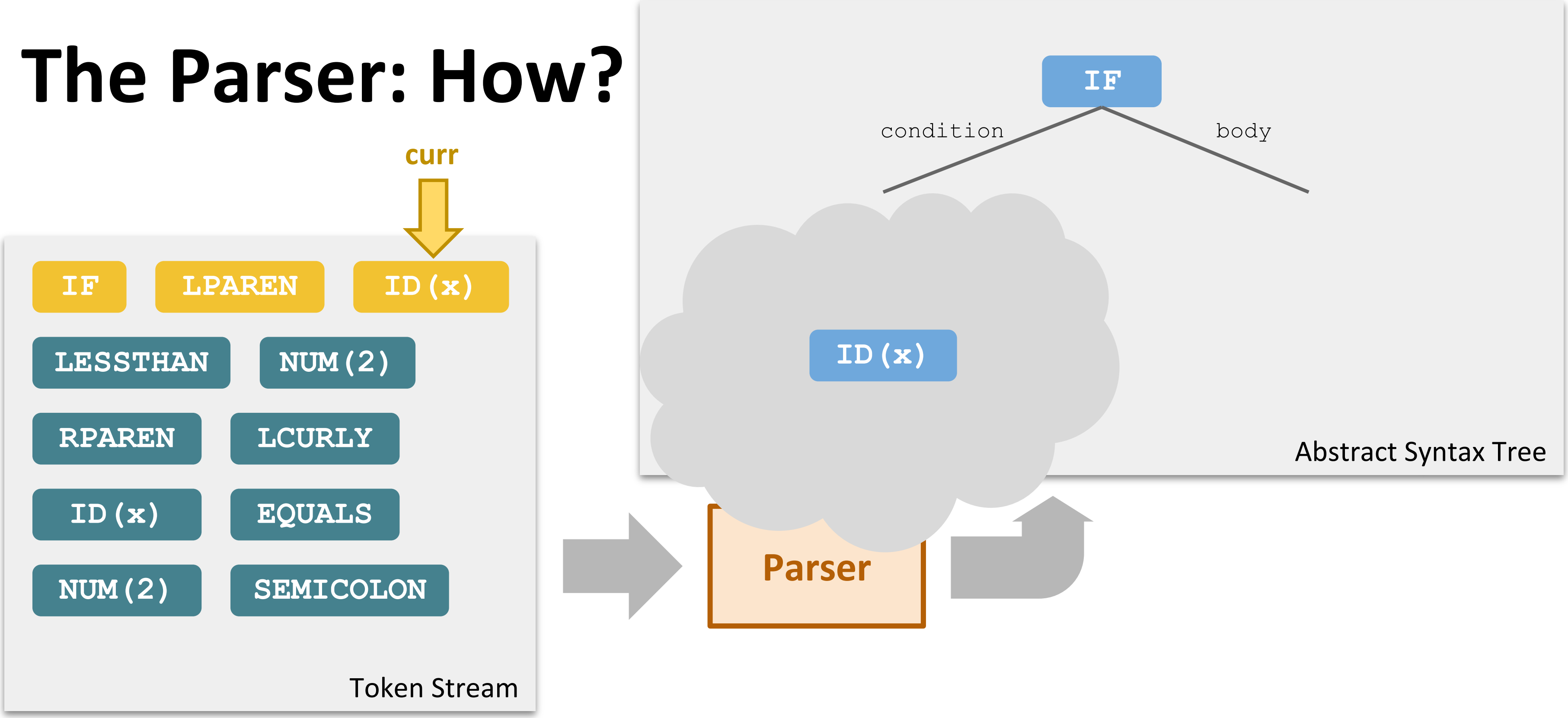
- Like a scanner: pass token stream, building up as we go
- Intuition: If we see IF and LPAREN , we are entering an if statement and next we must see a complete expression
  - Keep reading until we have a complete expression (recursively parse that) and attach on the condition side of the IF

The Parser: How?



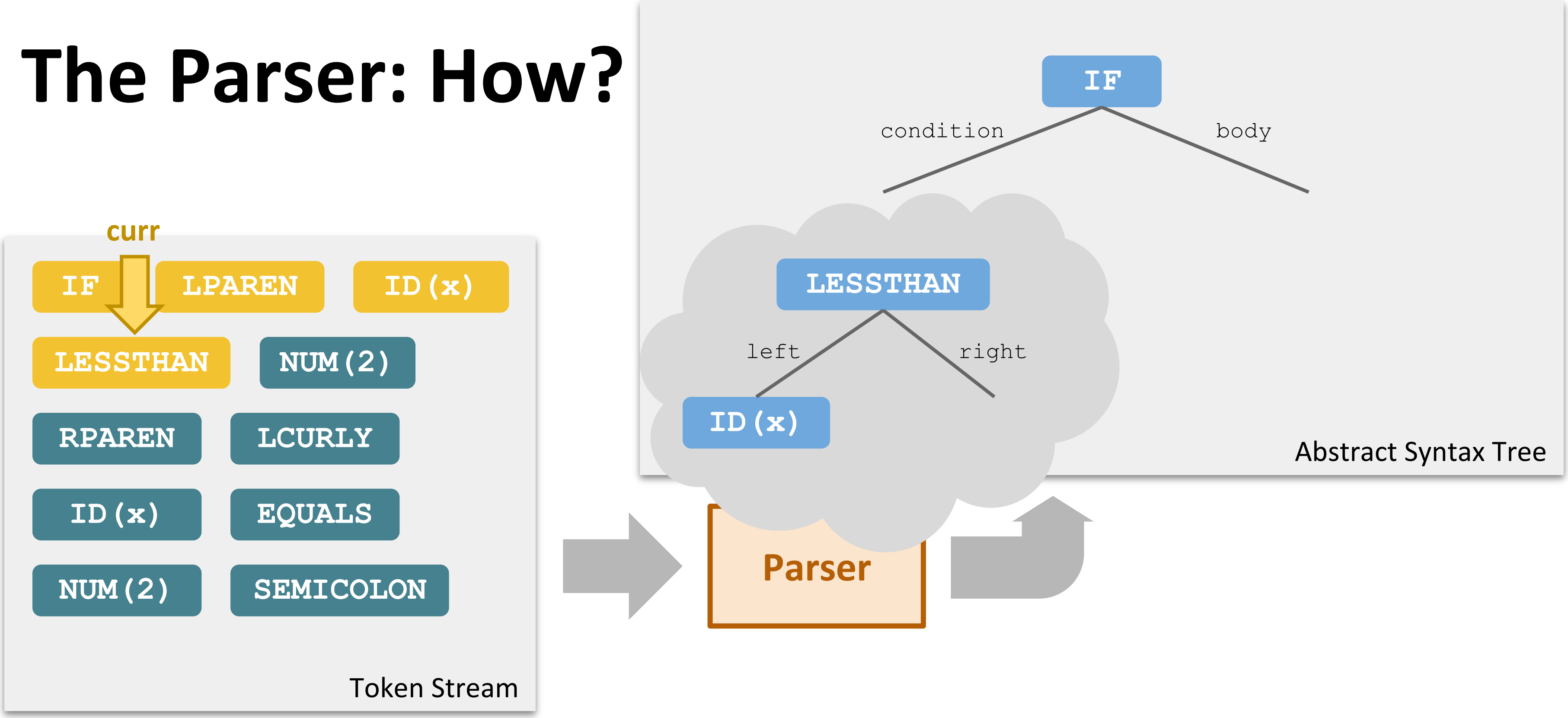
- Like a scanner: pass token stream, building up as we go
- Intuition: If we see IF and LPAREN , we are entering an if statement and next we must see a complete expression
  - Keep reading until we have a complete expression (recursively parse that) and attach on the condition side of the IF

The Parser: How?



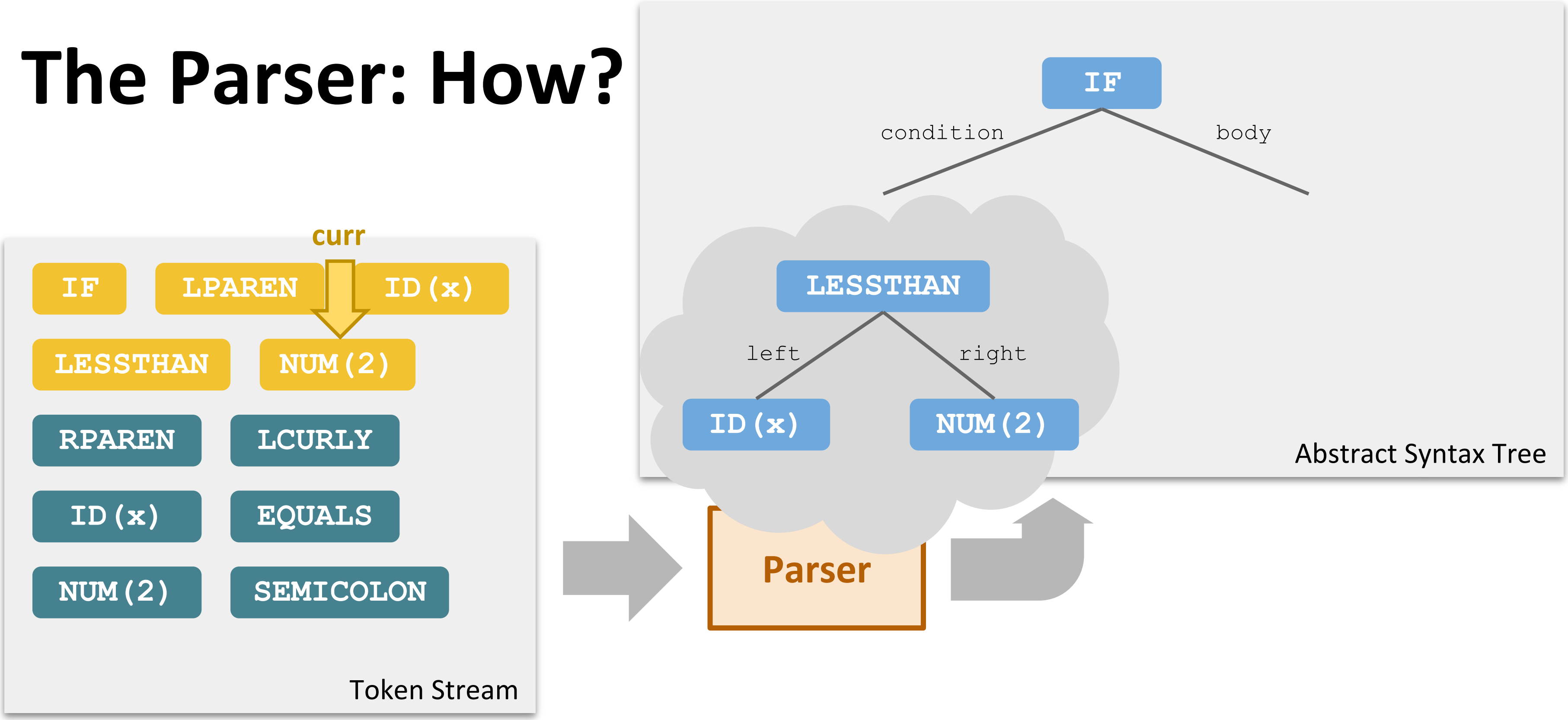
- Like a scanner: pass token stream, building up as we go
- Intuition: If we see IF and LPAREN , we are entering an if statement and next we must see a complete expression
  - Keep reading until we have a complete expression (recursively parse that) and attach on the condition side of the IF

The Parser: How?



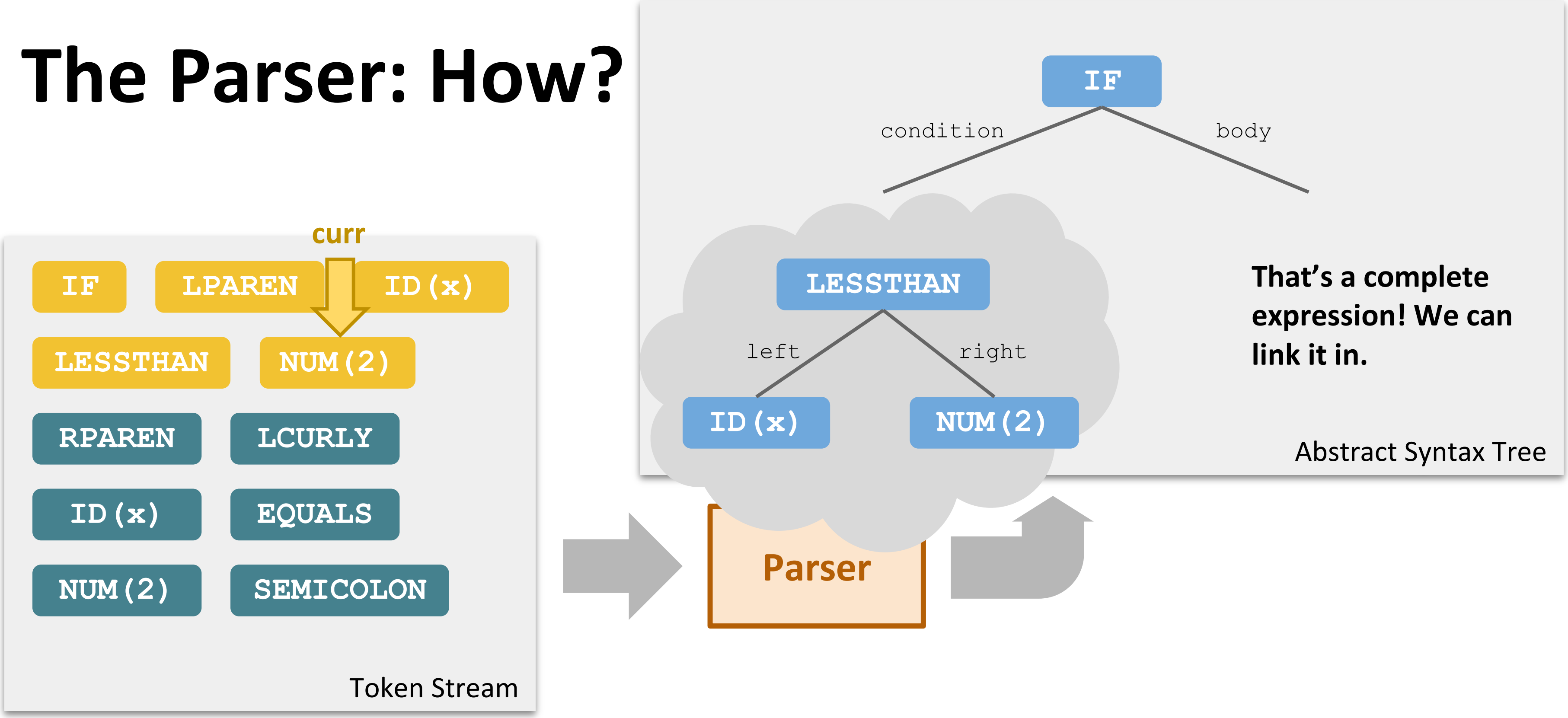
- Like a scanner: pass token stream, building up as we go
- Intuition: If we see IF and LPAREN , we are entering an if statement and next we must see a complete expression
  - Keep reading until we have a complete expression (recursively parse that) and attach on the condition side of the IF

The Parser: How?



- Like a scanner: pass token stream, building up as we go
- Intuition: If we see IF and LPAREN , we are 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

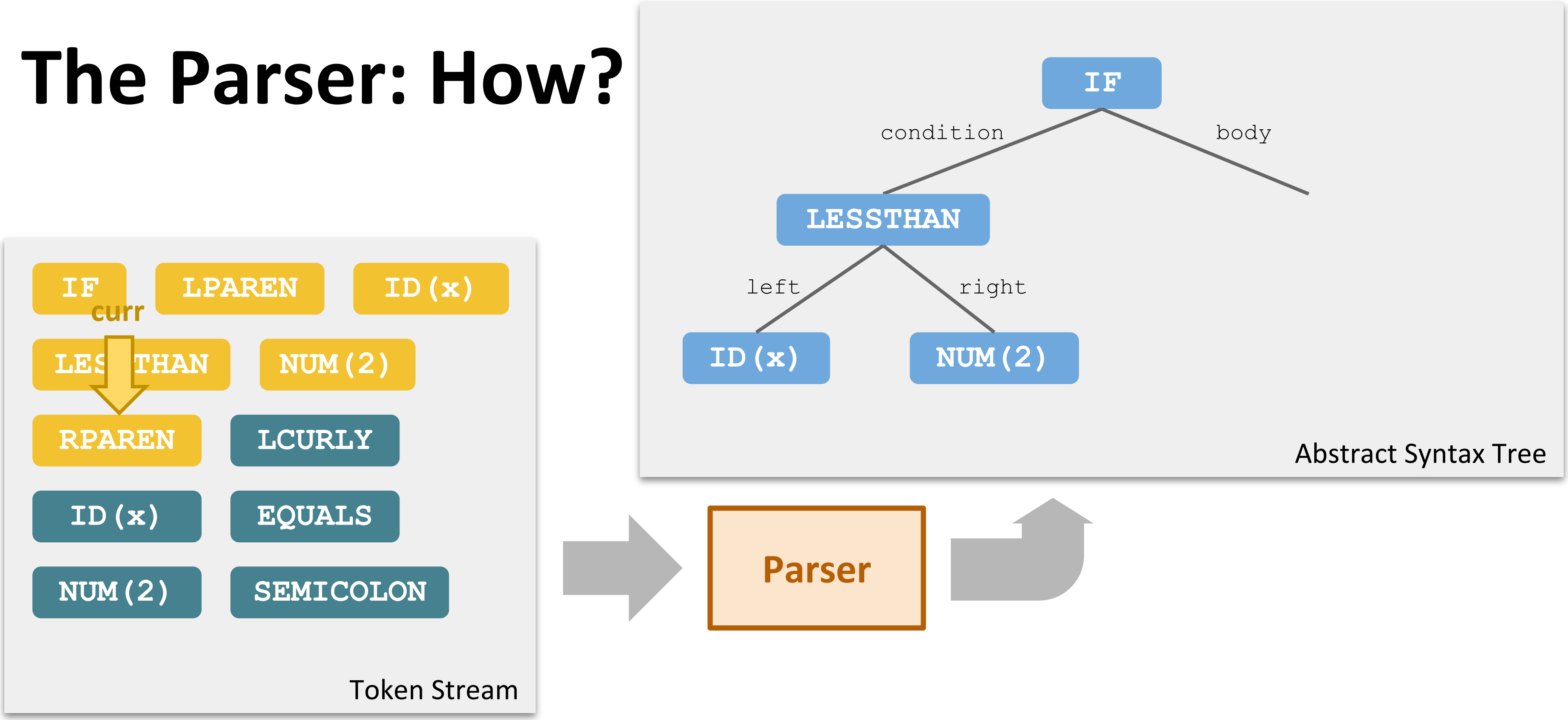
The Parser: How?



That's a complete expression! We can link it in.

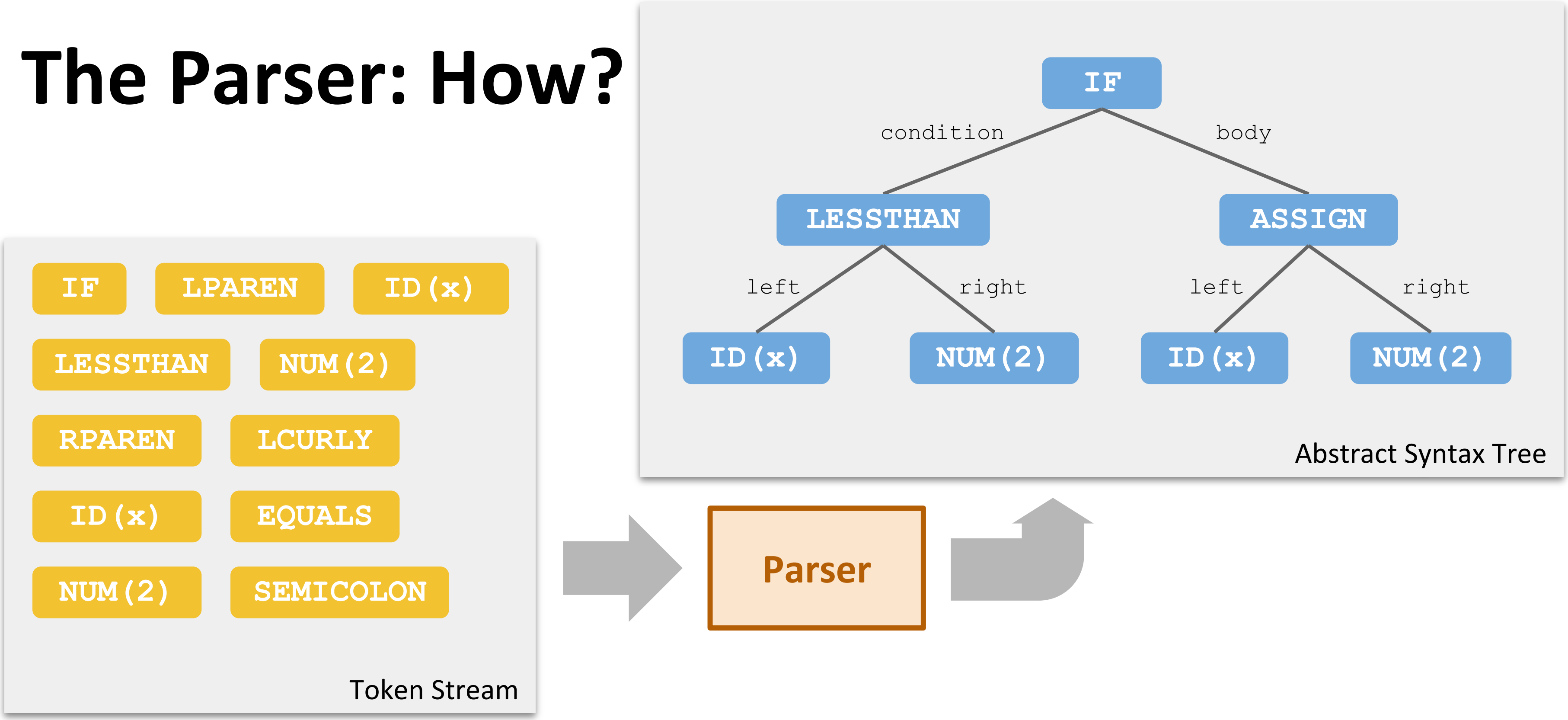
- Like a scanner: pass token stream, building up as we go
- Intuition: If we see IF and LPAREN , we are entering an if statement and next we must see a complete expression
  - Keep reading until we have a complete expression (recursively parse that) and attach on the condition side of the IF

The Parser: How?



- Like a scanner: pass token stream, building up as we go
- Intuition: If we see `IF` and `LPAREN`, we are 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`

The Parser: How?

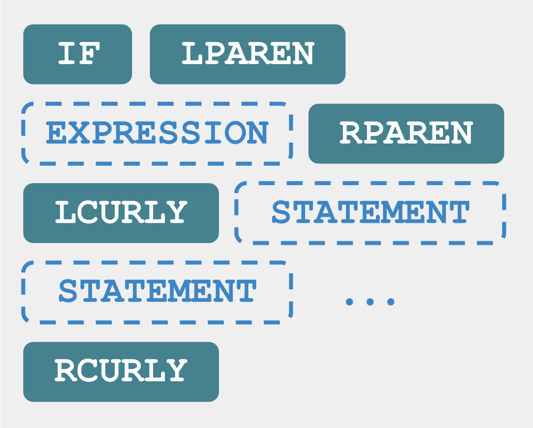


- Like a scanner: pass token stream, building up as we go
- Intuition: If we see `IF` and `LPAREN` , we are entering an if statement and next we must see a complete expression
  - Keep reading until we have a complete expression (recursively parse that) and attach on the condition side of the `IF`

The Parser: How?

- Implementing the Parser is essentially encoding the token stream definition, which can be recursive

Token Stream Definition



```
parseStatement() {
  ...
  if (currToken() == IF) {
      next() //consume "if"
      next() //consume "("

      // consumes tokens in expr
      e = parseExpression()

      next() // consume ")"
      next() // consume "{"

      // consumes tokens in
      // statement
      s = parseStatement()
      ...
      return new If(e, s)
  }
  ...
}
```

Lecture Outline

- Meeting with a Professor
  - How to Connect with Professors
  - How Connection with Professors Benefit Us

- **Exploring the Compiler Phases**
  - Scanner: Process of Tokenizing an Input File
  - Parser: Making Meaning From Tokens Through ASTs
  - **Type Checking, Optimization, and Code Generation**

- Project 7 Overview
  - Midterm Corrections, Professor Meeting Report

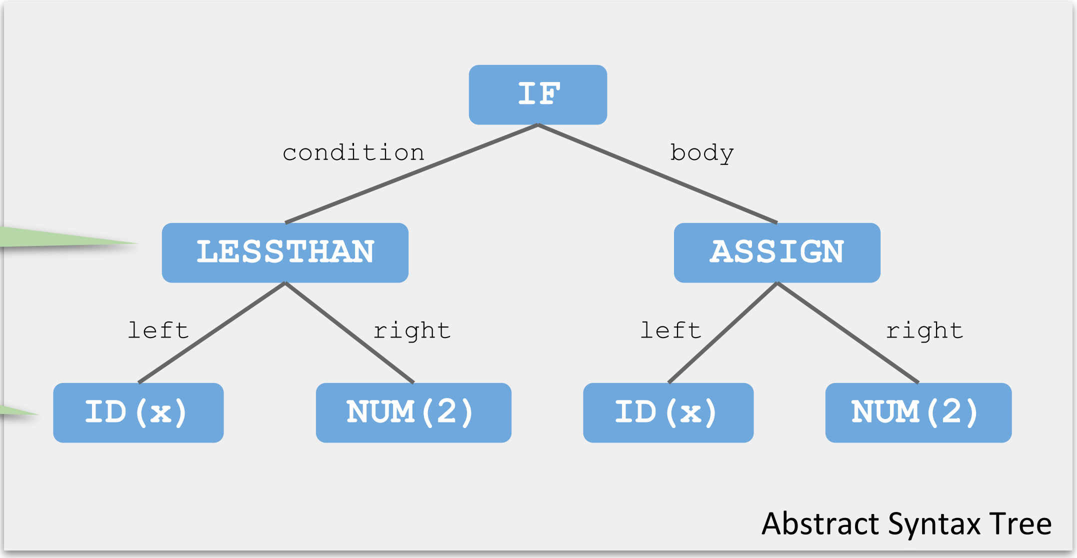
Type Checking (Semantic Analysis)

- 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 and the type of arguments in each function

Does this expression evaluate to a Boolean?

Is the variable "x" defined at this point?



Abstract Syntax Tree

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

- At its core, all that the code generation phase does is read through the Abstract Syntax Tree and print a set of statements depending on the AST node

- More on code generation next week

Lecture Outline

- Meeting with a Professor
  - How to Connect with Professors
  - How Connection with Professors Benefit Us
- Exploring the Compiler Phases
  - Scanner: Process of Tokenizing an Input File
  - Parser: Making Meaning From Tokens Through ASTs
  - Type Checking, Optimization, and Code Generation
- **Project 7 Overview**
  - **Midterm Corrections, Professor Meeting Report**

Project 7 Overview

- Part I: Midterm Corrections
  - Due next Friday (2/23) at 11:59pm (**no late days** can be used on midterm corrections)
  - Open-notes, open-tools
  - Only need to redo the problems that you missed
  - 50% of the points you earn back from midterm corrections will be added to your original midterm score
  - You can calculate your new midterm score using this formula:

$$Original\ Midterm\ Score + \frac{New\ Midterm\ Score - Original\ Midterm\ Score}{2}$$

- Part II: Professor Meeting Report
  - Due in two weeks on 3/1 at 11:59pm
  - Please schedule the meeting as early as possible

Project 7, Part I: Midterm Corrections

- Review feedback from the course staff, celebrate the questions you got right, reflect on which areas you can continue to grow in

- If you think a problem was graded incorrectly, feel free to submit a regrade request on Gradescope
  - Don’t be afraid to challenge our grading
  - This is a great learning opportunity for us all

- You can earn up to 50% of the points back that you missed on the midterm

Lecture 15 Reminders

- **Project 6 (Mock Exam Problem & Building a Computer) due tonight (2/16) at 11:59pm**

- Project 7, Part I (Midterm Corrections) due next Friday (2/23) at 11:59pm
  - Reminder that no late days may be used on midterm corrections

- Project 7, Part II (Professor Meeting Report) released, due in two weeks on 3/1 at 11:59pm

- Eric has office hours after class in CSE2 153
  - Feel free to post your questions on the Ed board as well