CSE 390B, Autumn 2022

Building Academic Success Through Bottom-Up Computing

Building Connections & Compiler Phases

Building People Connections in College, Exploring the Compiler Phases, Project 7 Overview

Lecture Outline

- **Building People Connections in College**
  - **Benefits of Building Connections, Networking Strategies**

- 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

Benefits of Building Connections

- Reaching out to your professors, TAs, and peers can be a great way to discover opportunities
- Taking the time to connect with these people can open several doors and leverage your potential
- 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

Strategies for Networking

- Get involved in communities on campus (e.g., RSOs, TAing, research, part-time campus job)

- Invest in building relationships with people and developing a presence in their lives

- Take time to reflect on how others can support you by bringing to them your interests and questions

- Not all networking efforts will be well-received, but don't be afraid to just go for it

Discussion on Building Connections

In groups, spend 4-6 minutes discussing these questions:

- In what ways do you already connection with others on a regular basis? How else can you build your connections?

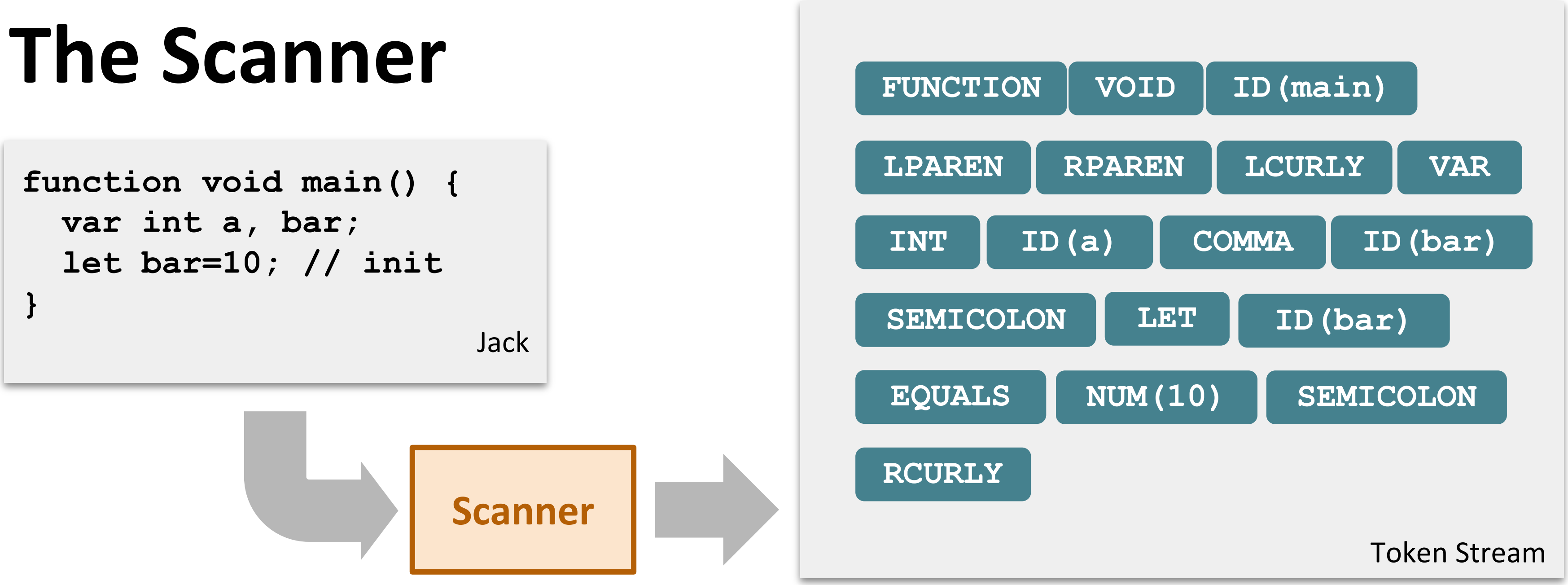
- How can you benefit from building your community of people you can network with?

- What would you share with someone you recently made a connection with?

Lecture Outline

- Building People Connections in College
  - Benefits of Building Connections, Networking Strategies
- **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 Scanner



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

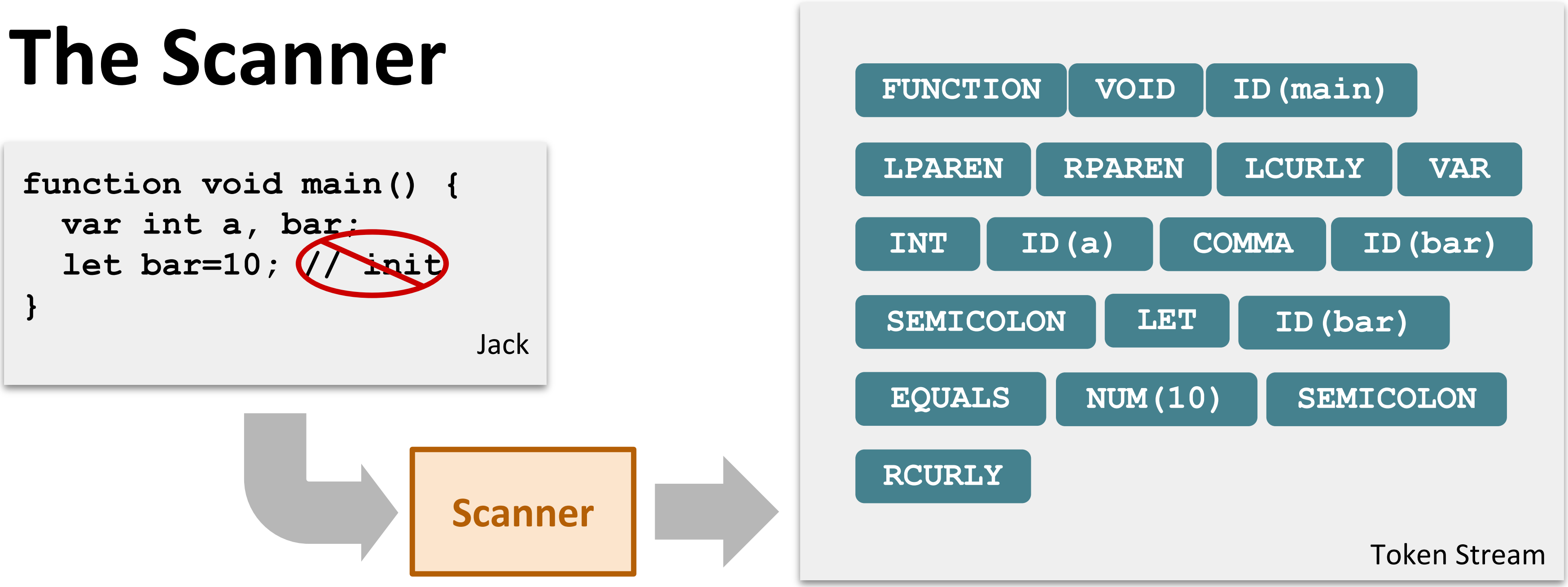
Jack

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

The Scanner



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

Jack

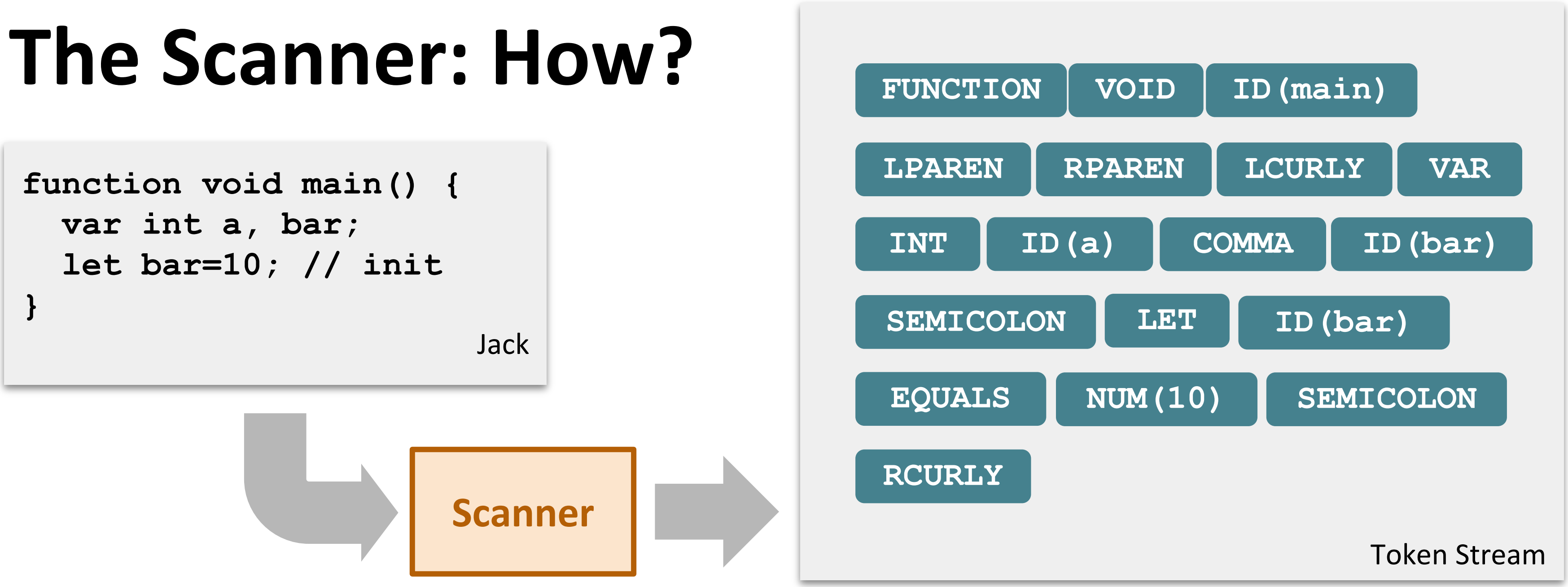
Scanner

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

- 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

Scanner

FUNCTION VOID ID(main)

LPAREN RPAREN LCURLY VAR

INT ID(a) COMMA ID(bar)

SEMICOLON LET ID(bar)

EQUALS NUM(10) SEMICOLON

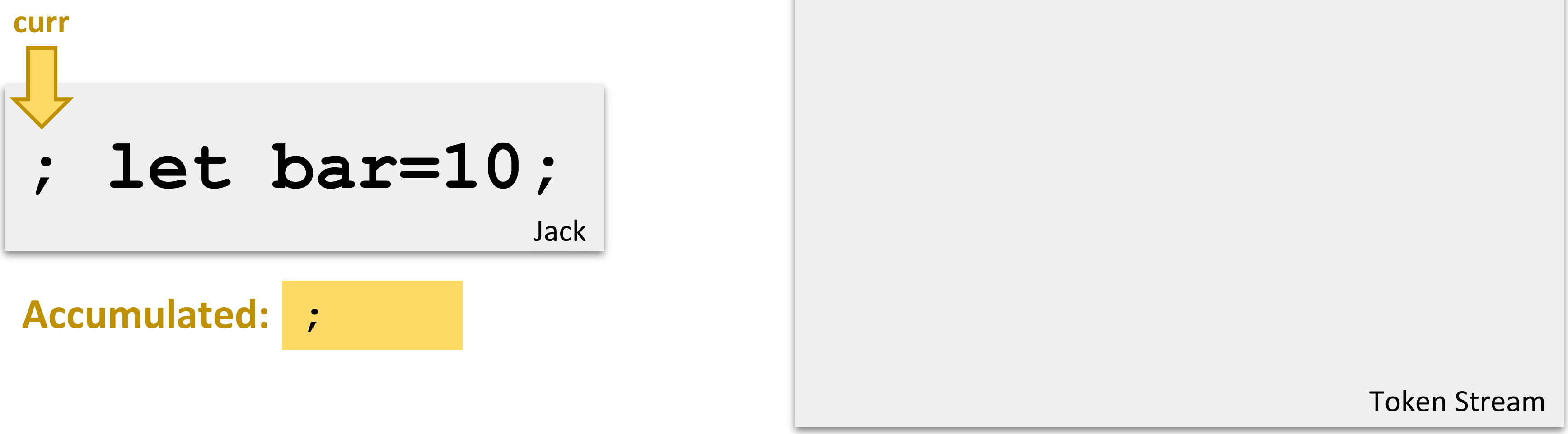
RCURLY

Token Stream

- 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;
```

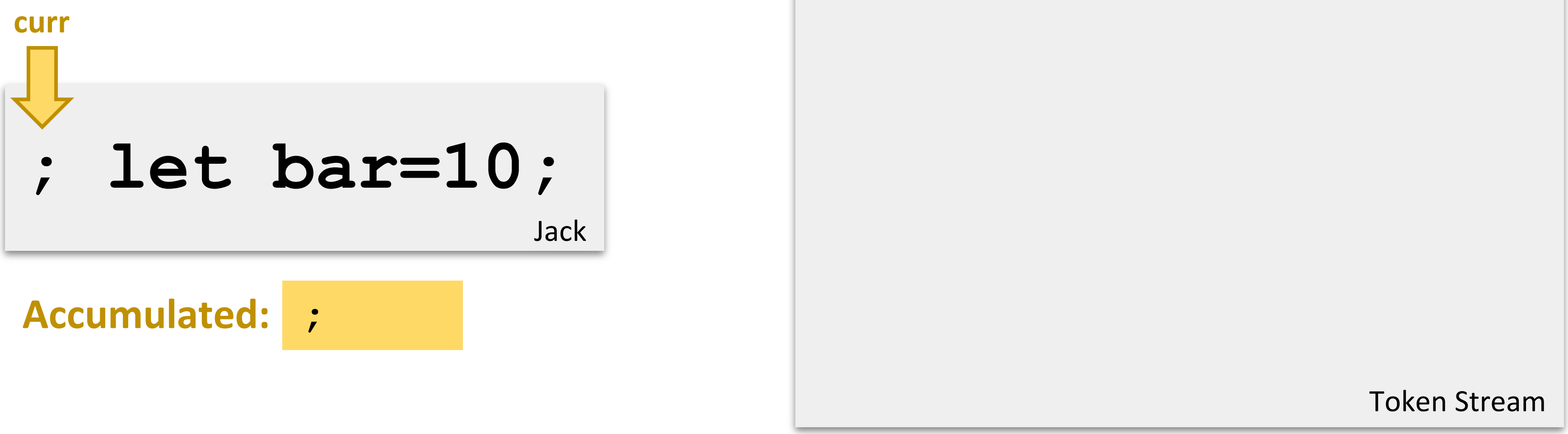
Jack

Accumulated: ;

Token Stream

- How to distinguish built-in keywords (e.g., “let”) from identifiers (e.g., “bar”)?
  - When token is done, check against list of keywords

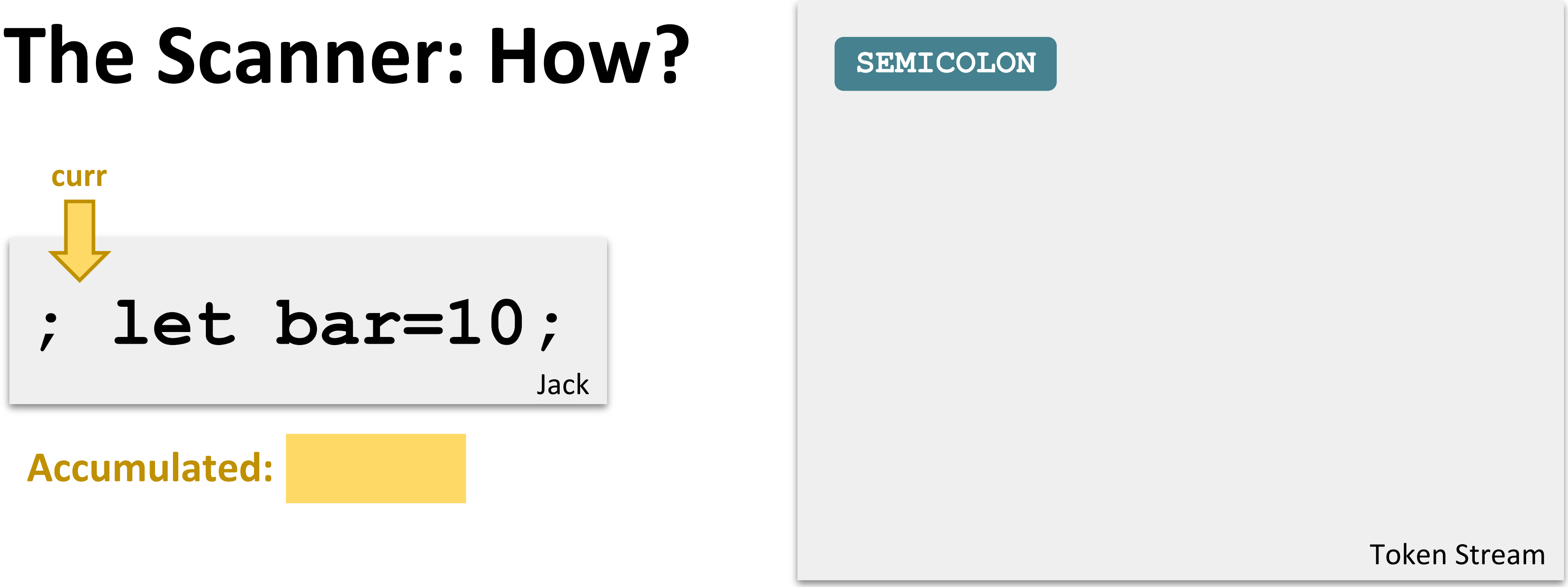
The Scanner: How?



```
; 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

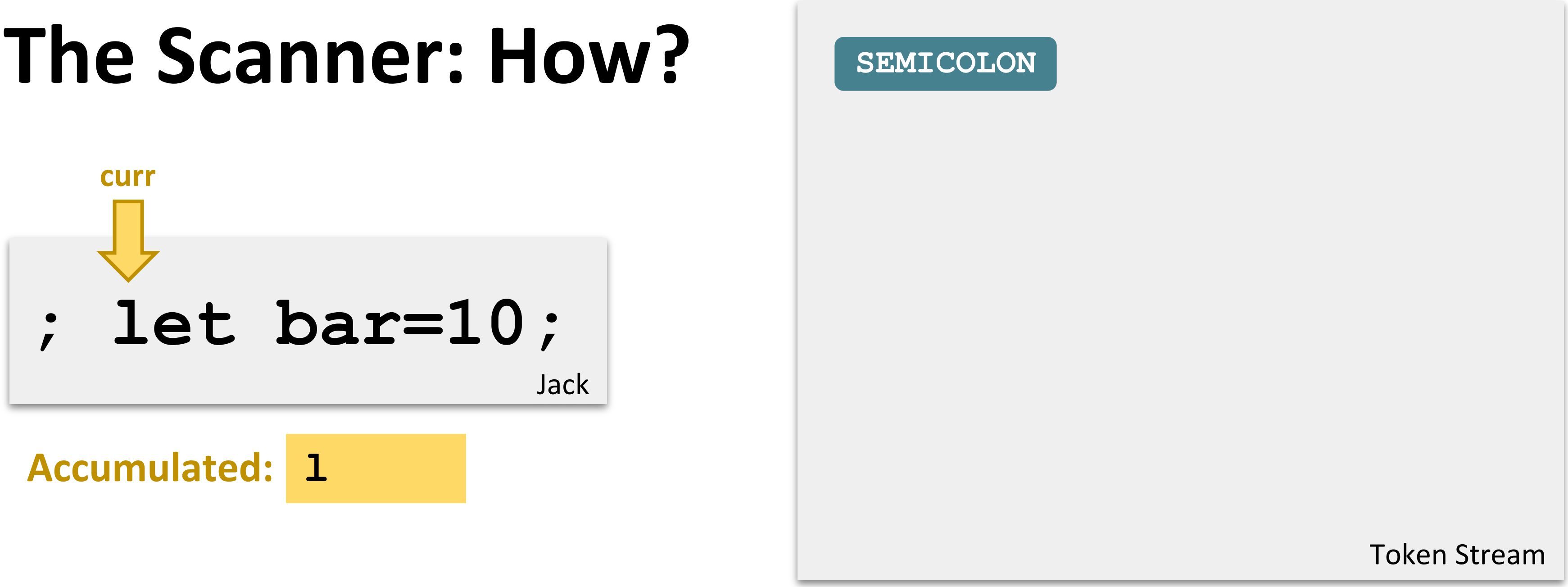
Accumulated:

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

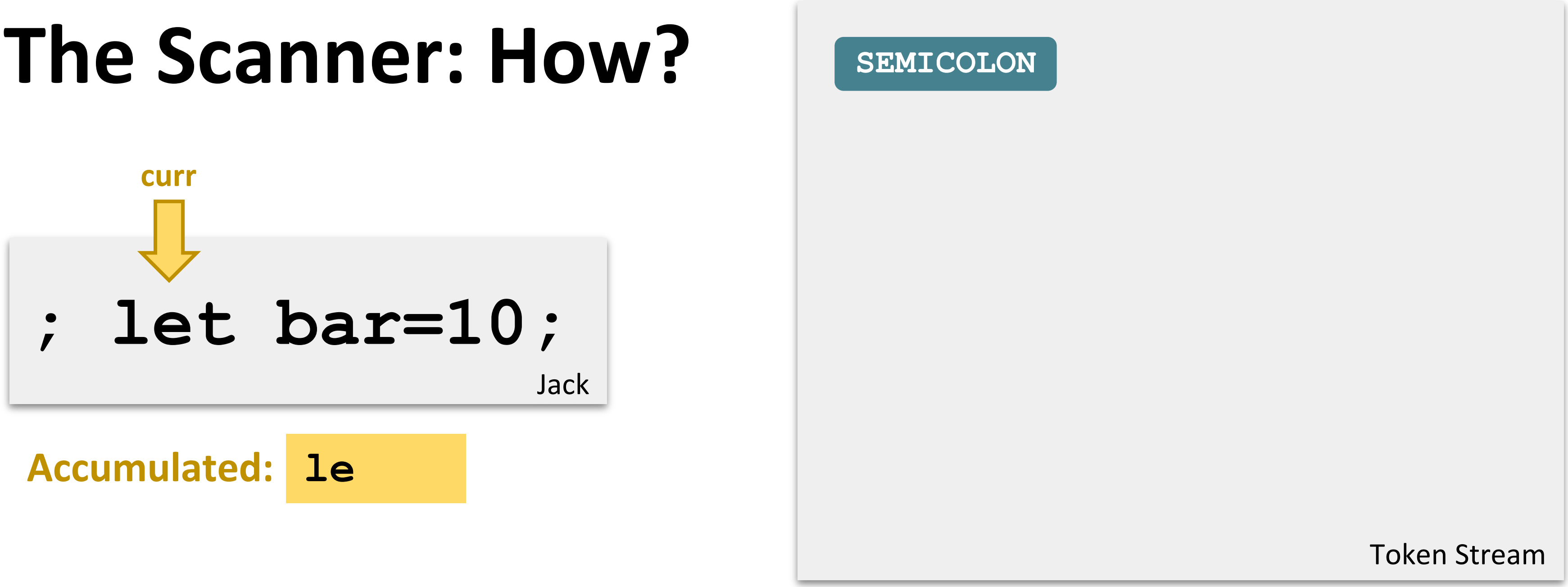
Accumulated: 1

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?



curr

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

Jack

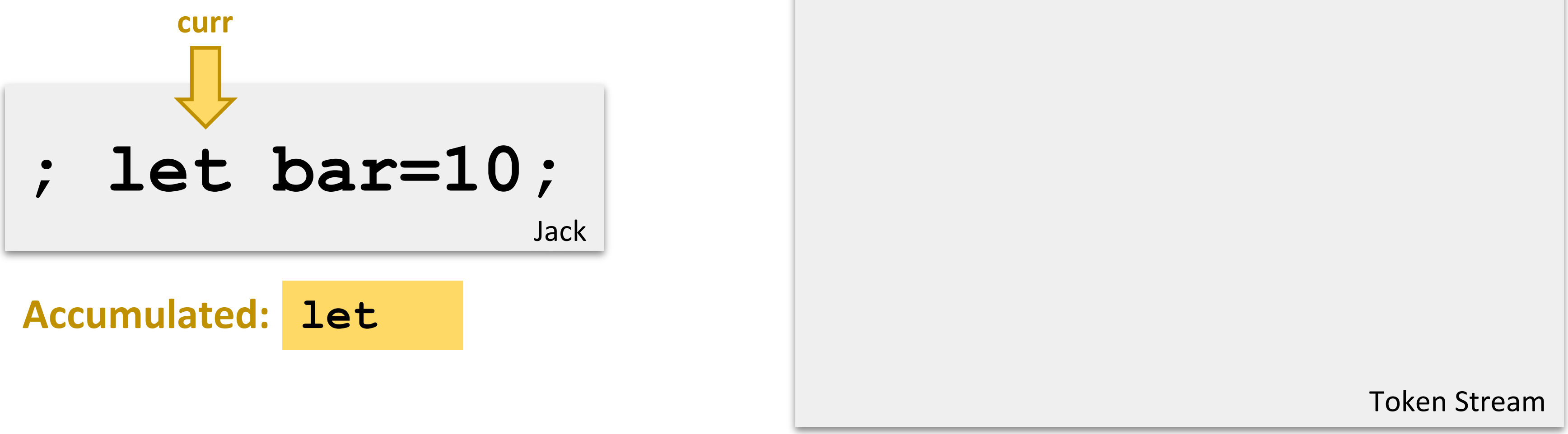
Accumulated: `le`

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

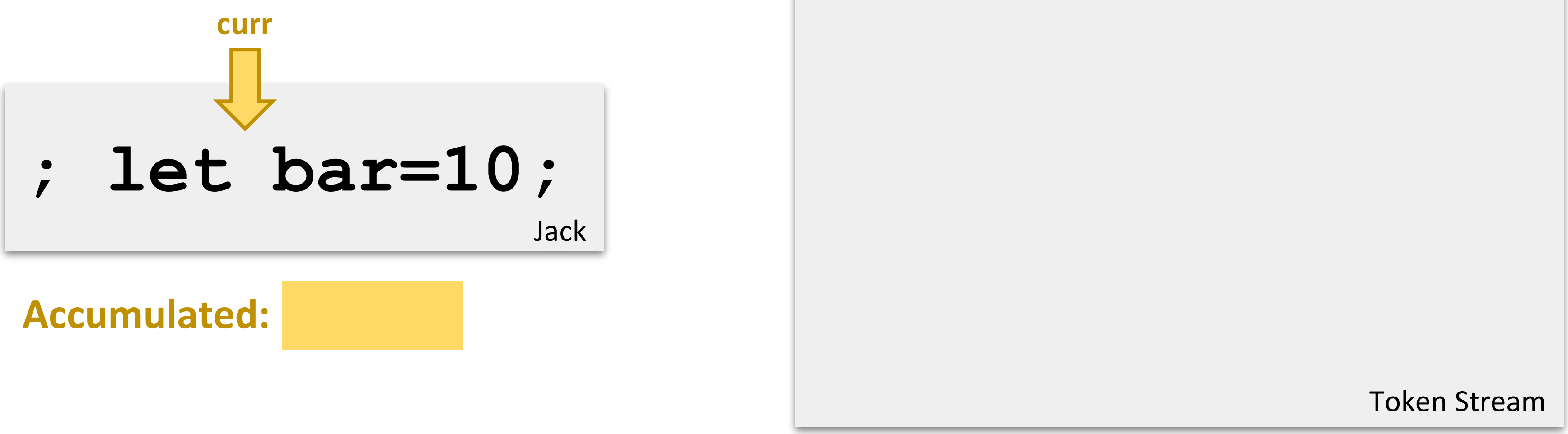
Accumulated: `let`

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

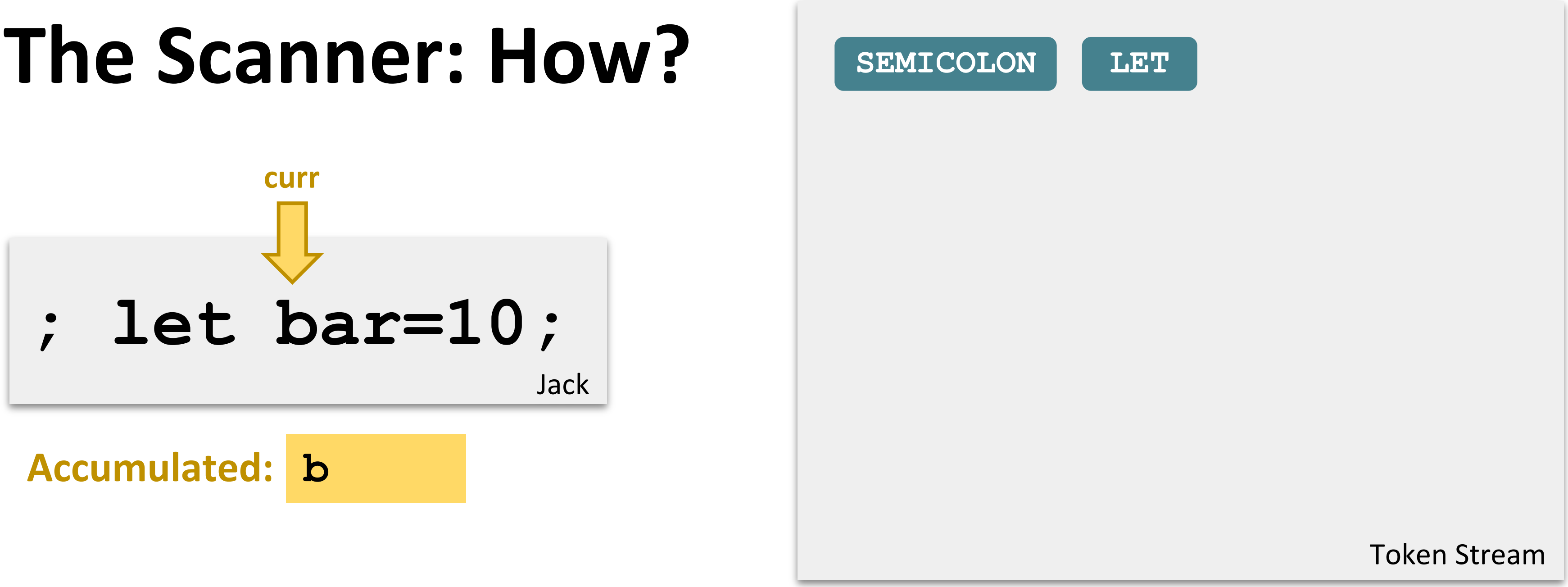
Accumulated:

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

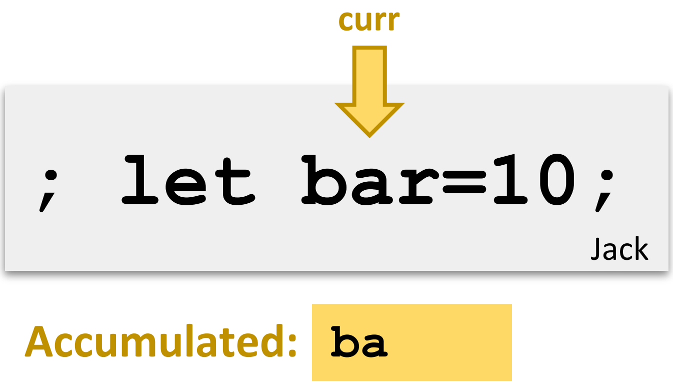
Accumulated: b

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?

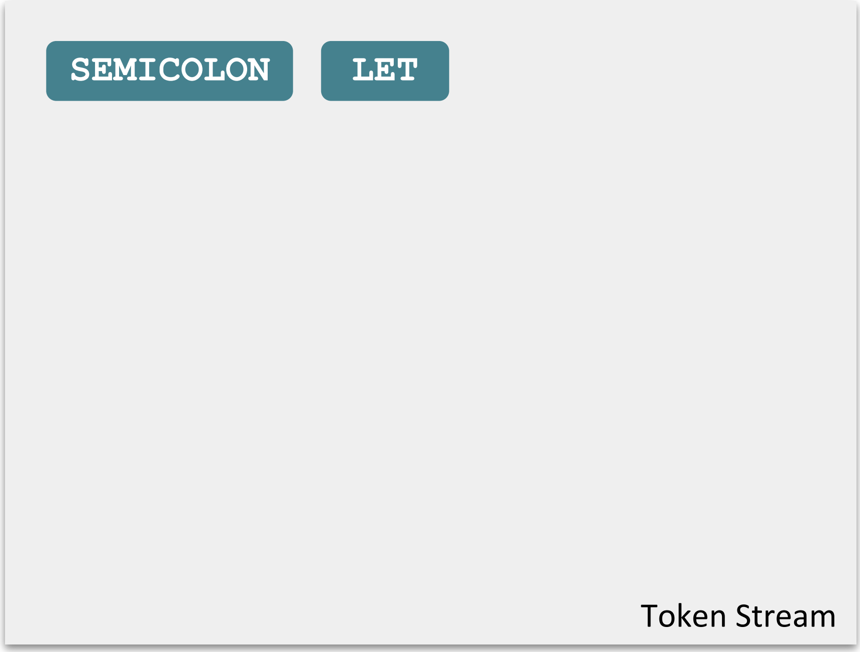


curr

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

Jack

Accumulated: `ba`

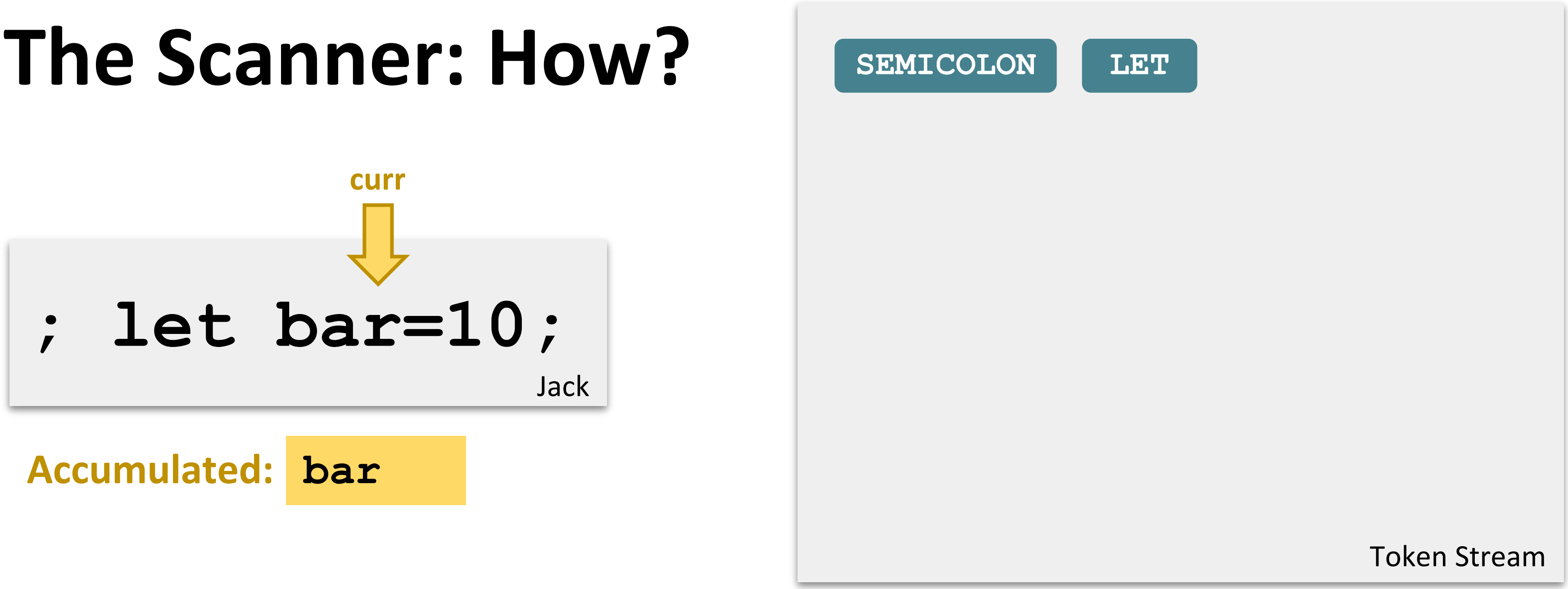


SEMICOLON LET

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?



curr

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

Jack

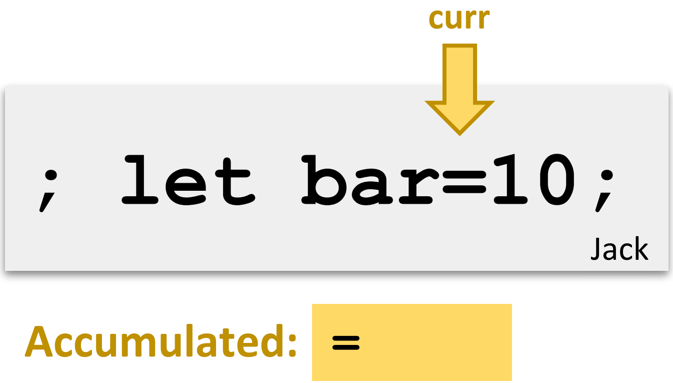
Accumulated: bar

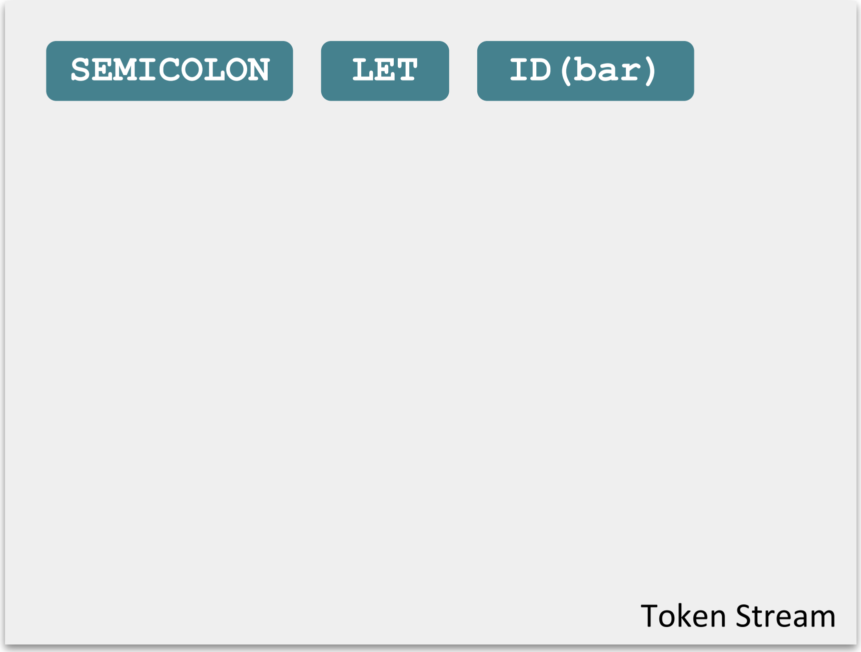
SEMICOLON LET

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

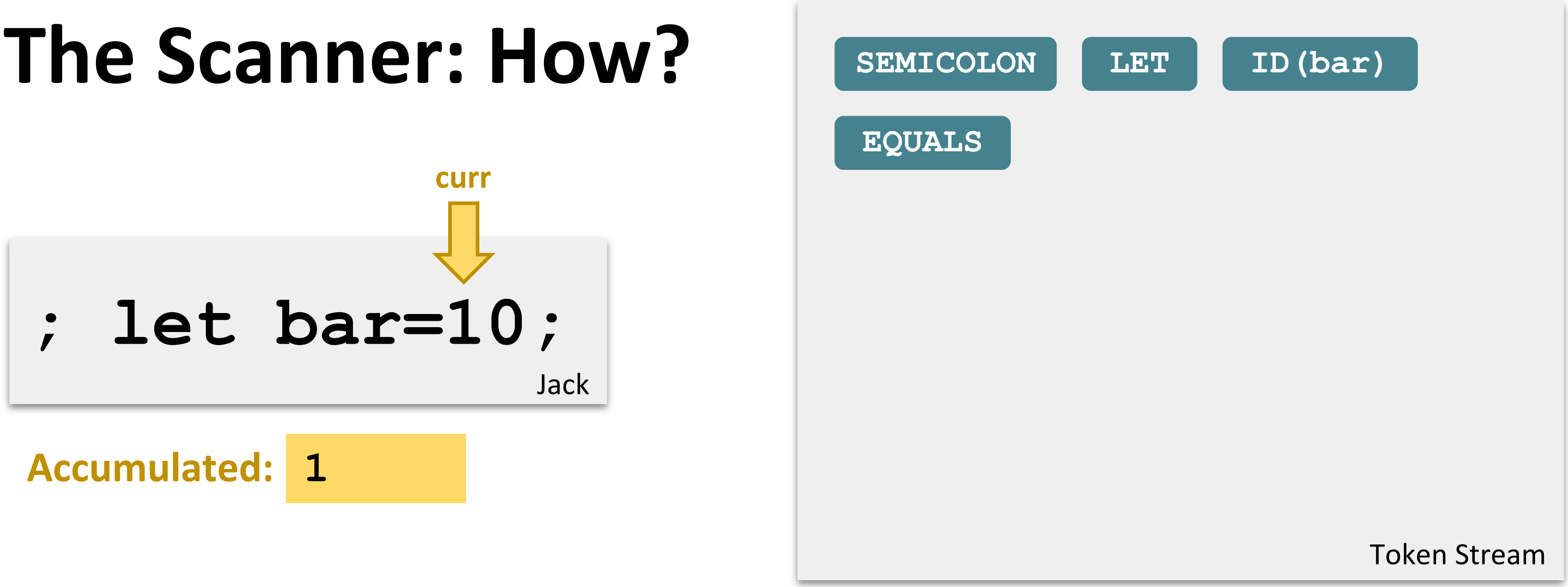
The Scanner: How?





- 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?



curr

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

Jack

Accumulated: 1

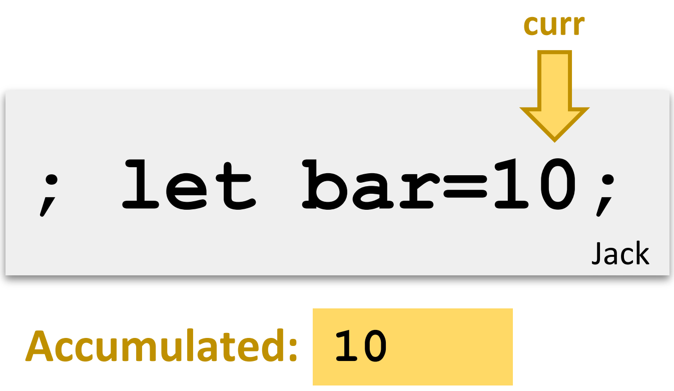
SEMICOLON LET ID(bar)

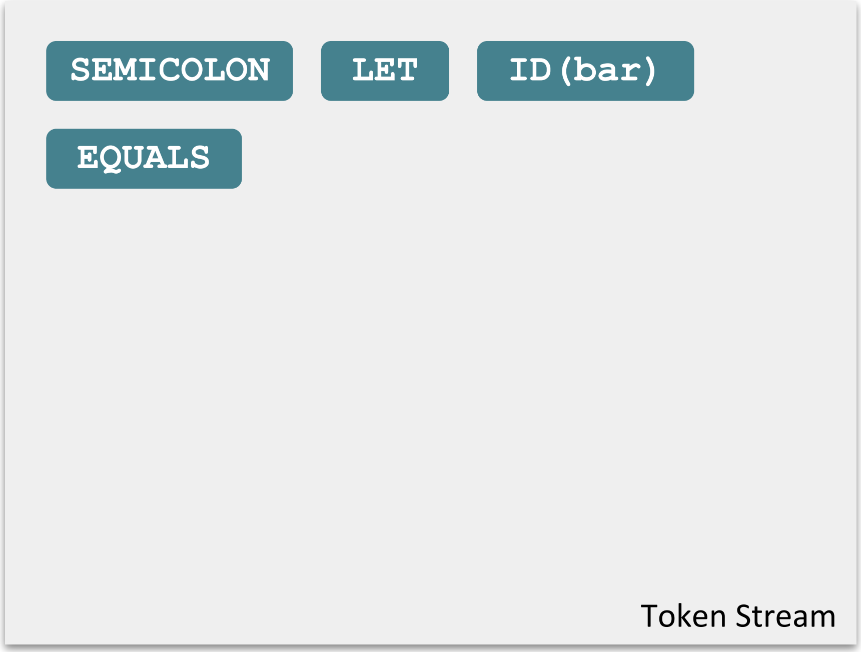
EQUALS

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

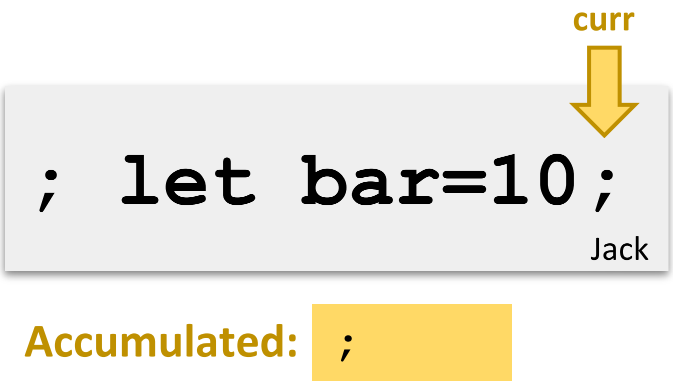
The Scanner: How?

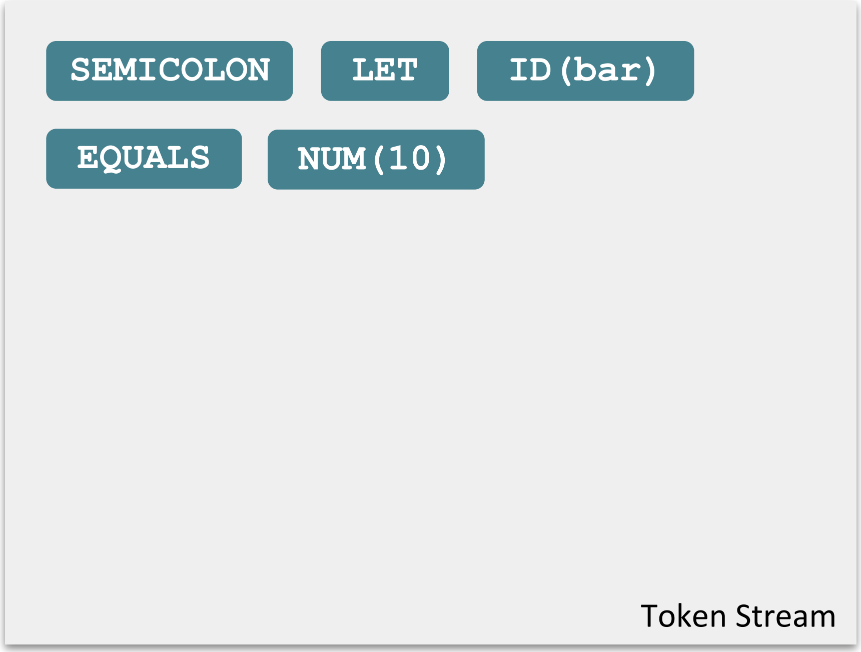




- How can we take a line of code in Jack and convert this into a token stream?
  - Keep cursor on current char
  - Break off a token when we complete one
  - If the next char could be part of this token, accumulate it

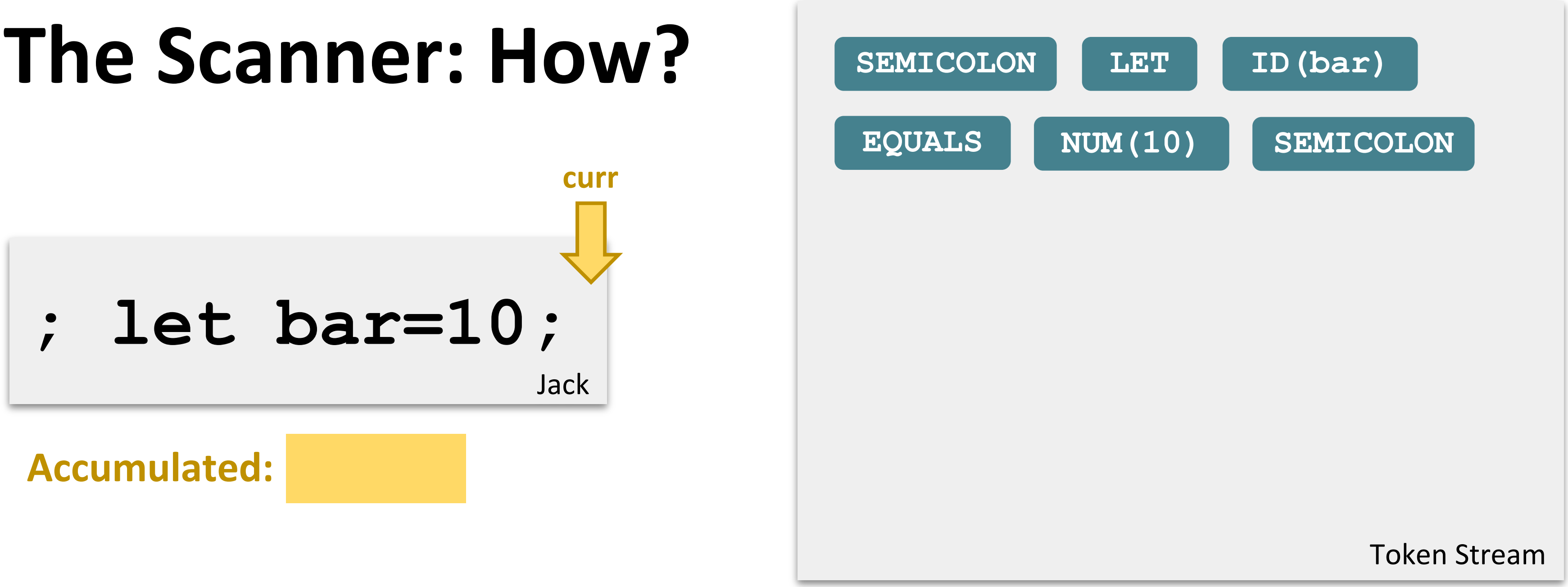
The Scanner: How?





- 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?



- 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: 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

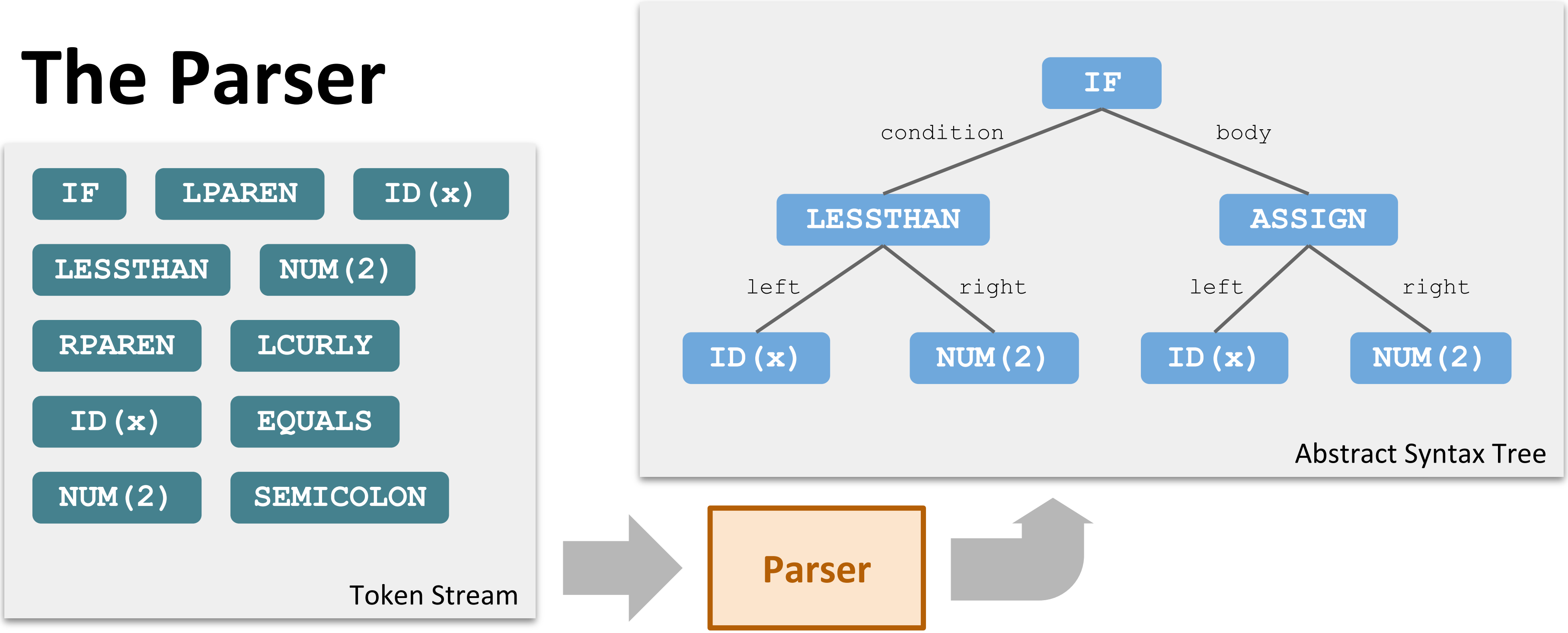
Lecture Outline

- Building People Connections in College
  - Benefits of Building Connections, Networking Strategies

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

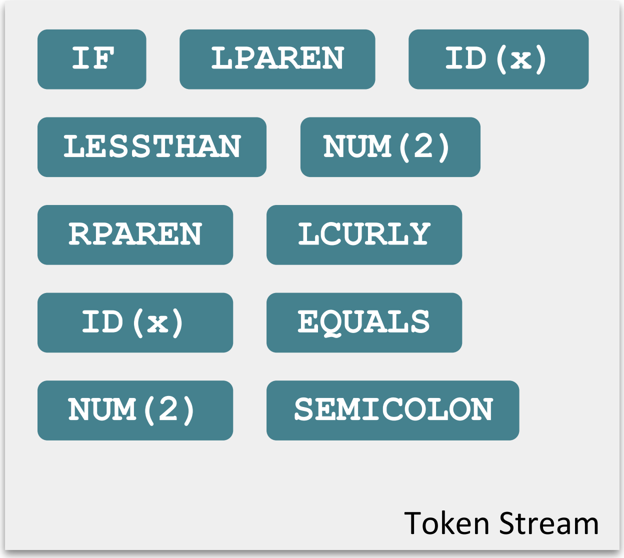
- Project 7 Overview
  - Midterm Corrections, Professor Meeting Report

The Parser

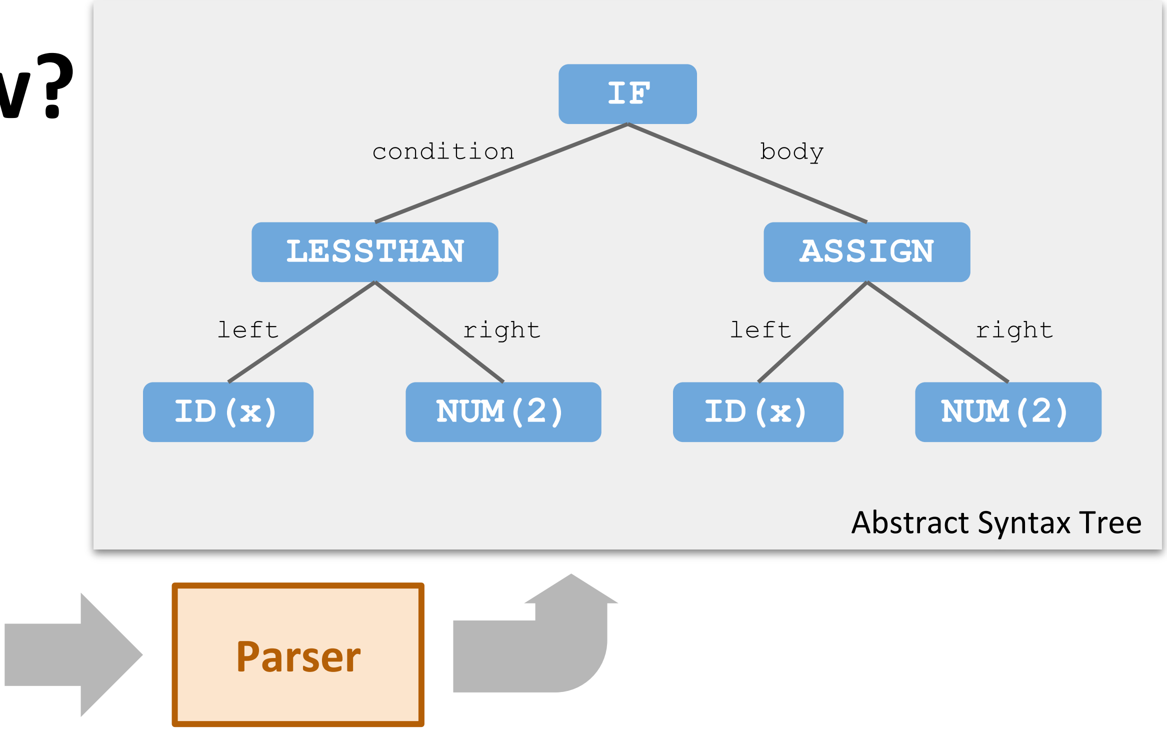


- Takes in the *flat* token stream and outputs a *structured* tree representation of program constructs
- Result: an **Abstract Syntax Tree**
  - Captures the structural features of the program
  - **Important distinction**: cares about big-picture syntax (E.g., entire **`if`** statement) rather than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

The Parser: How?



Token Stream



Abstract Syntax Tree

- Like scanner: single pass-through 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

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
```

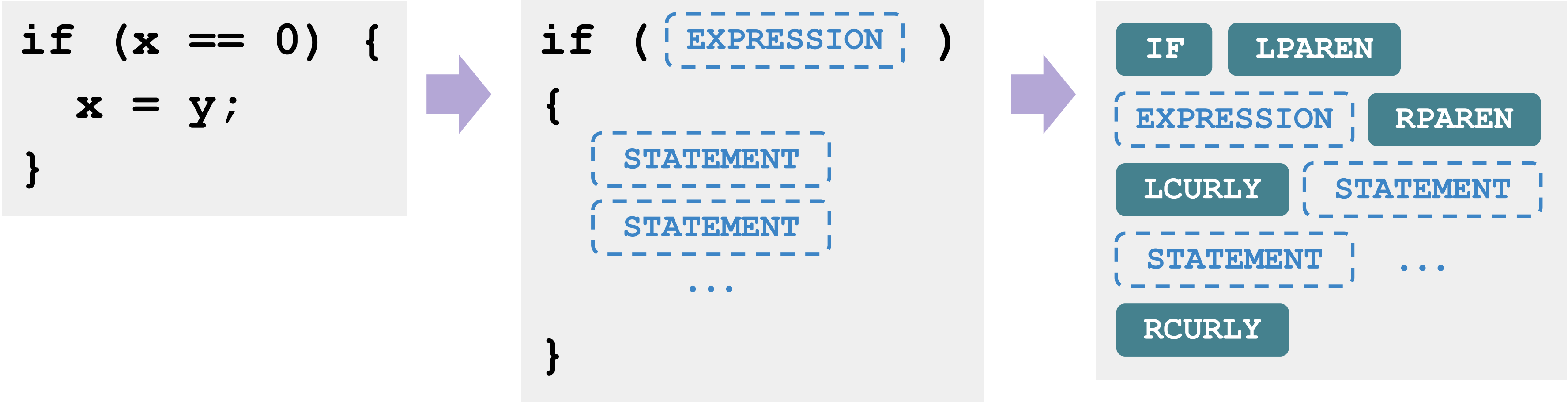
- Constant

```
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 | **General Definition of an `if` Statement** | **Token Stream Definition**



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

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

Lecture Outline

- Building People Connections in College
  - Benefits of Building Connections, Networking Strategies
- **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

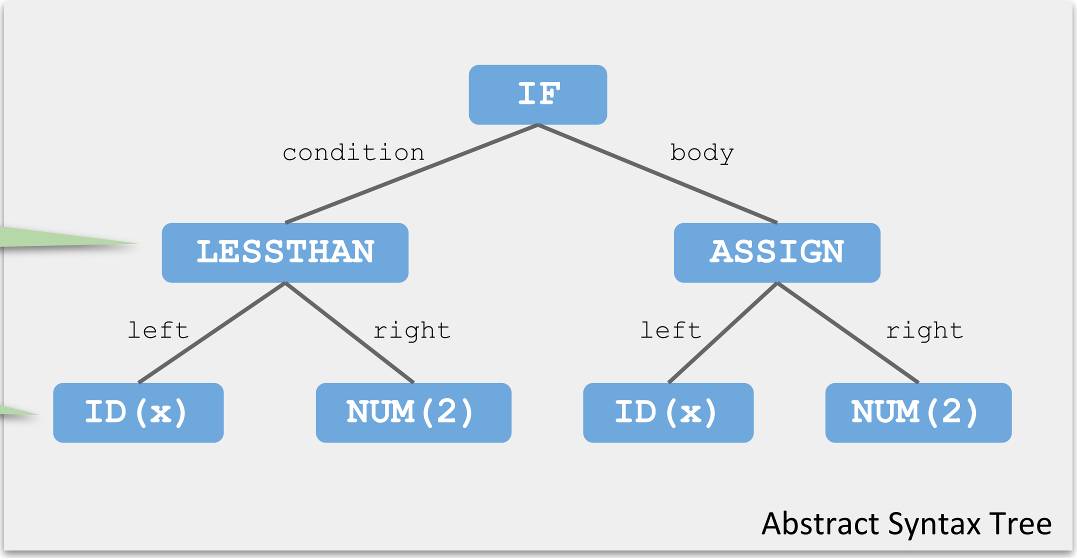
Type Checking (Semantic Analysis)

- 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

Lecture Outline

- Midterm Debrief
  - Grading Observations and Next Steps
- Introduction to Compilers
  - 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 on 11/23 (Wednesday) at 11:59pm (no late days can be used on this part)
  - Open-notes, open-tools
  - Only need to redo the problems that you missed
  - After midterm corrections, your midterm grade will be updated to be the average of your original midterm score and your redo score
  - Reach out to the course staff for support

- Part II: Professor Meeting Report
  - Due in two weeks on 12/1 at 11:59pm
  - Schedule the meeting as early as possible
  - Please do not tell your professor this is for an assignment

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

Professor Meeting Report Discussion

In groups, spend 4-6 minutes discussing these questions:

- Which professors are you thinking about reaching out to? Why do you choose them?
- What questions would you ask to your professor? Why did you choose those questions?
- How can you apply the skill of meeting with professors in different contexts to help you succeed as a UW student? In your career?

Lecture 15 Reminders

- **Project 6: Mock Exam Problem & Building a Computer due tonight (11/17) at 11:59pm**

- Project 7: Midterm Corrections & Professor Meeting Report released, due next **Wednesday** (11/23) at 11:59pm
  - *Eric will host an extra office hours next Tuesday (11/22) at 1:30pm*

- Course staff support
  - Eric has office hours in CSE2 153 today after lecture
  - Feel free to post your questions on the Ed board as well