CSE 390 B Spring 2021

Intro to Compilers & Project 6 Overview

Compiler Introduction and Project 6 Overview

Agenda

- **Reading Q&A**
- Introduction to the Compiler
  - Overview
  - The Scanner
  - The Parser
- Project 6 Overview

Reading Q&A

Agenda

- Reading Q&A
- **Introduction to the Compiler**
  - **Overview**
  - The Scanner
  - The Parser
- Project 6 Overview

The Compiler: Goal

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

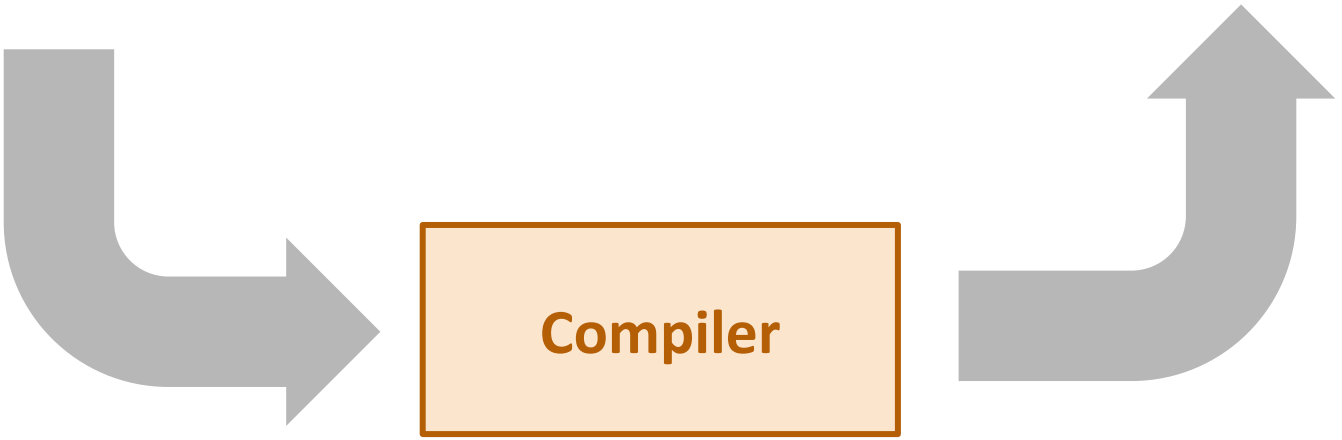
High-Level Language

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

Assembly Language

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

Practical Definition: a file containing a bunch of characters



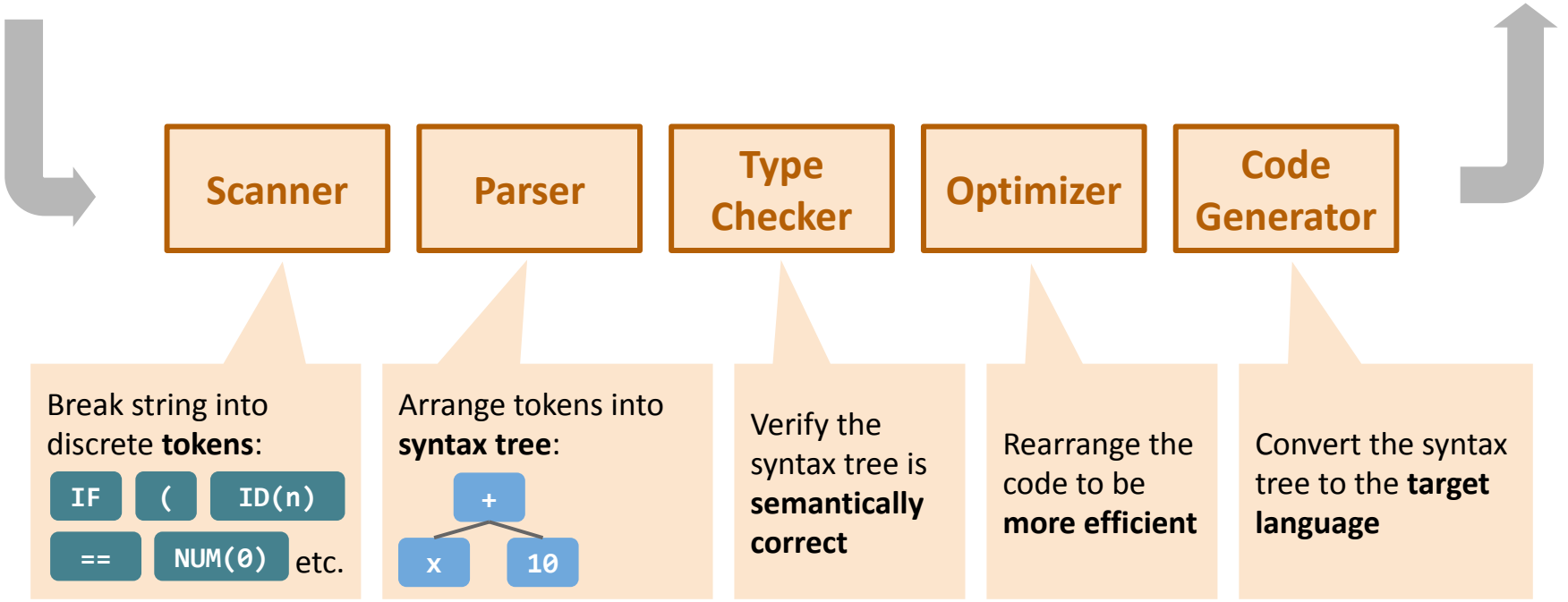
The Compiler: Implementation

```
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



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*
- We will compile from Jack to Hack assembly!

```
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

Compiler Lectures

- Today: First steps of the compiler
  - Important background information
  - Helps frame Thursday's lecture and project 7
  - You won't implement the parts we talk about today in this course (but you might in other courses!)

- Tuesday: code generation and project 7 specifics
  - Lots of details related to what you will be doing in project 7
  - Builds upon the context given in Tuesday's lecture
  - Only one part of the compiler! Each step we learn about is important

- Next Thursday: Debugging!
  - Metacognitive focus for Project 7

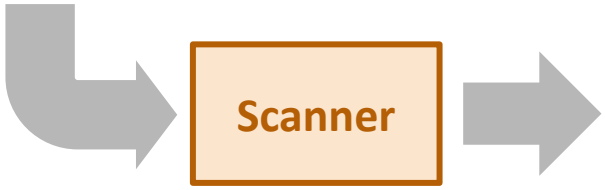
Agenda

- Reading Q&A
- **Introduction to the Compiler**
  - Overview
  - **The Scanner**
  - The Parser
- Project 6 Overview

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

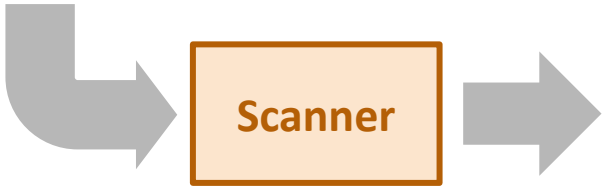
Token Stream

- 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

```
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

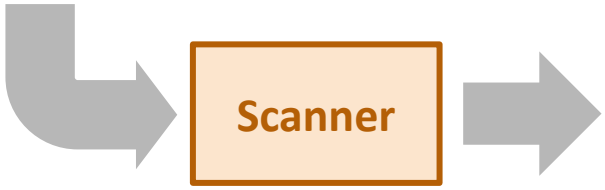
Token Stream

- 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



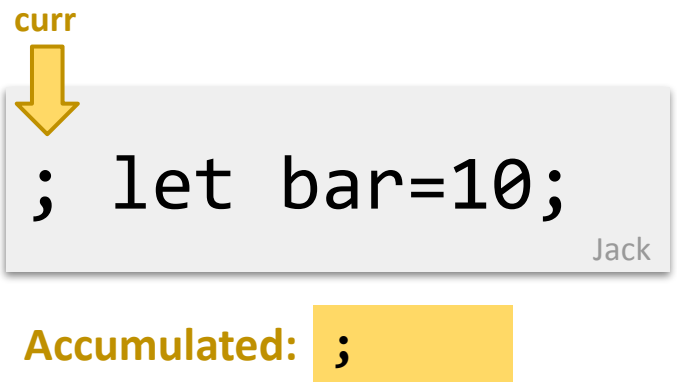
FUNCTION VOID ID(main)
LPAREN RPAREN LCURLY VAR
INT ID(a) COMMA ID(bar)
SEMICOLON LET ID(bar)
EQUALS NUM(10) SEMICOLON
RCURLY

Token Stream

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

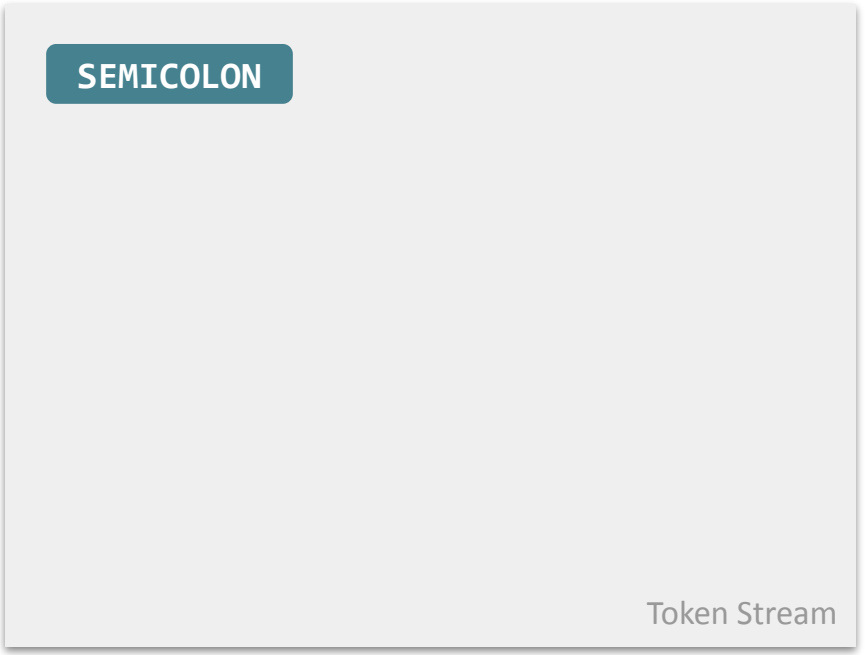
- Tempting, but we would end up with "`a,`" "`bar;`" "`bar=10;`"
  - Whitespace is tricky: generally want to ignore, but can't count on it being there!

The Scanner: How?



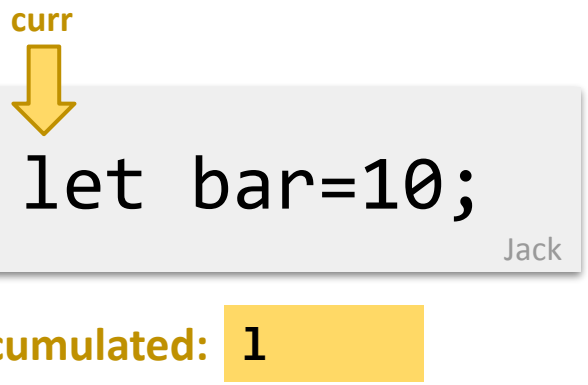
- Observation: many tokens have disjoint starting characters!
- Keep cursor on current char
  - Break off a token when we complete one
  - If the next char *could* be part of this token, accumulate it
- How to distinguish built-in keywords (e.g. “let”) from identifiers (e.g. “bar”)?
  - Solution: when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

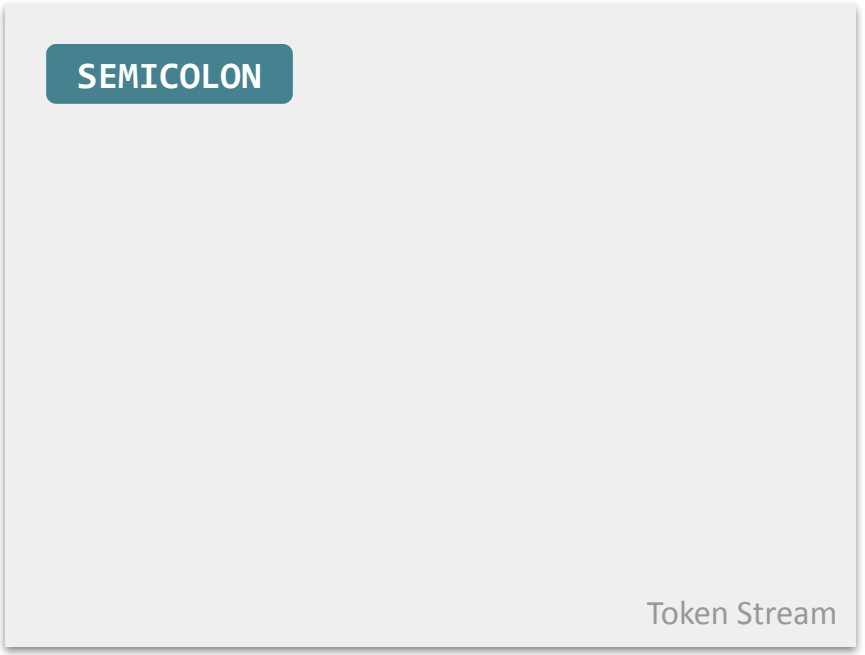
The Scanner: How?



- Observation: many tokens have disjoint 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”)?
  - Solution: when token is done, check against list of keywords

The Scanner: How?





- Observation: many tokens have disjoint 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”)?
  - Solution: when token is done, check against list of keywords

The Scanner: How?


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

Jack

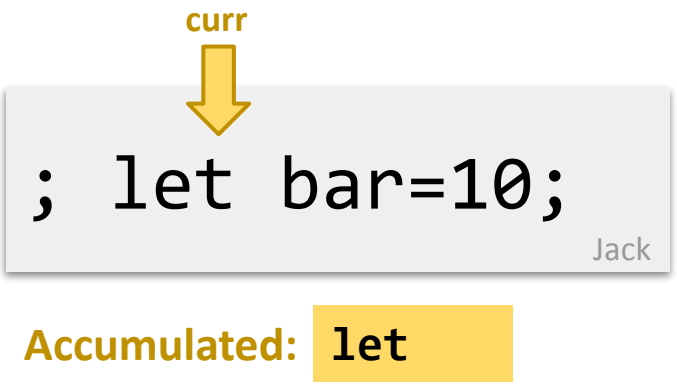
Accumulated: `le`

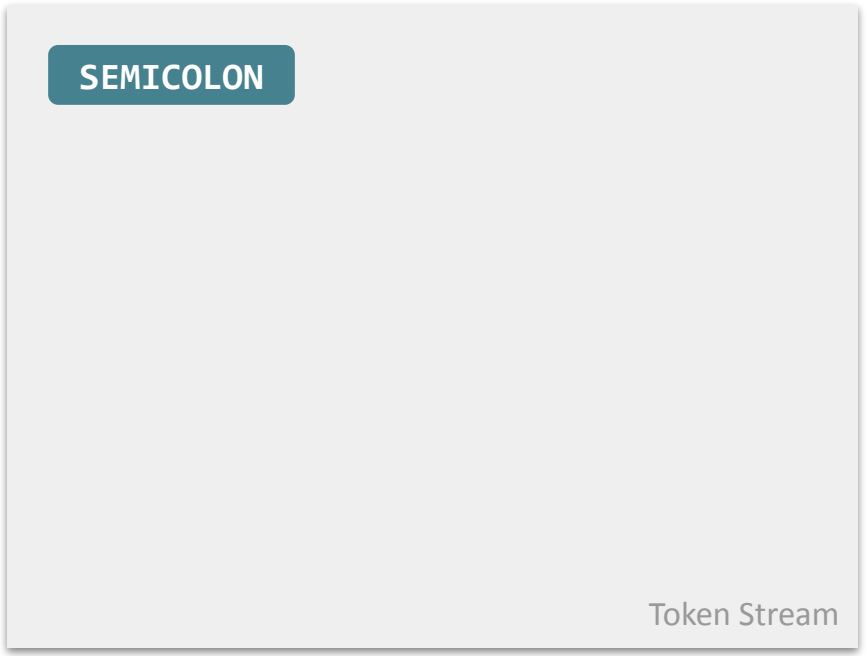
`SEMICOLON`

Token Stream

- Observation: many tokens have disjoint 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”)?
  - Solution: when token is done, check against list of keywords

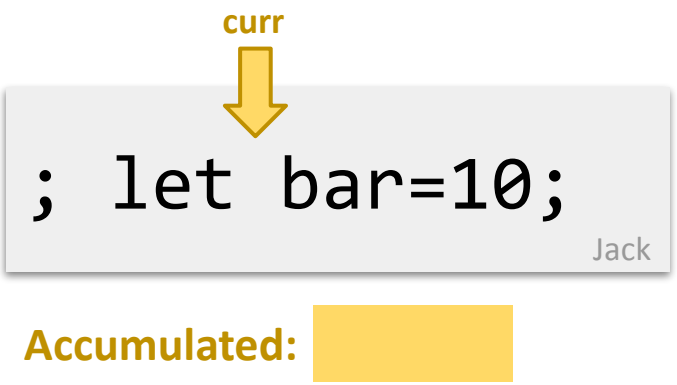
The Scanner: How?

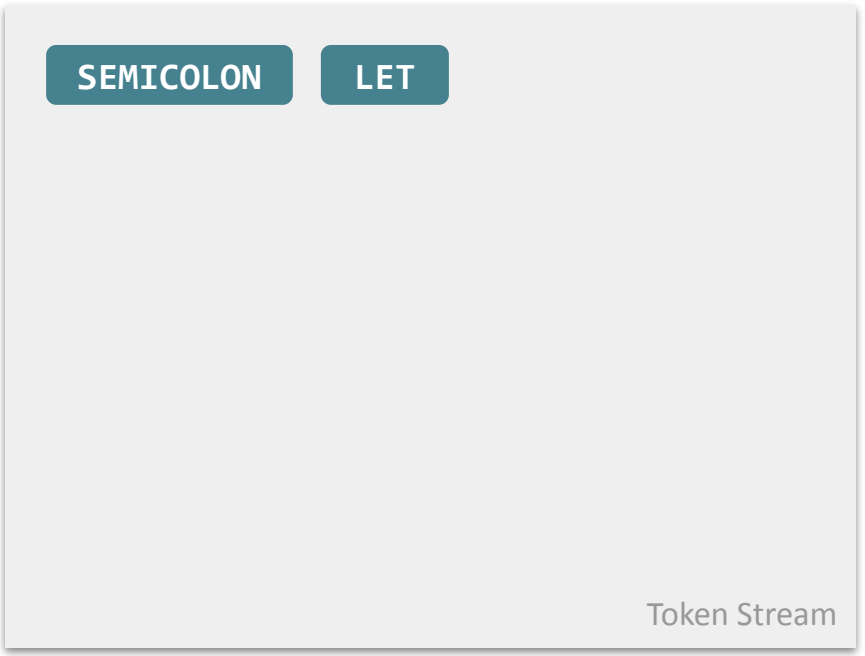




- Observation: many tokens have disjoint starting characters!
- Keep cursor on current char
  - If the char *could* be part of this token, accumulate it
  - If not, complete the current token
- How to distinguish built-in keywords (e.g. “let”) from identifiers (e.g. “bar”)?
  - Solution: when token is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

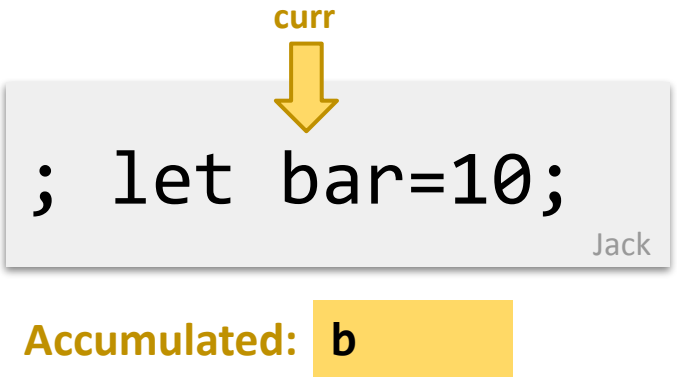
The Scanner: How?

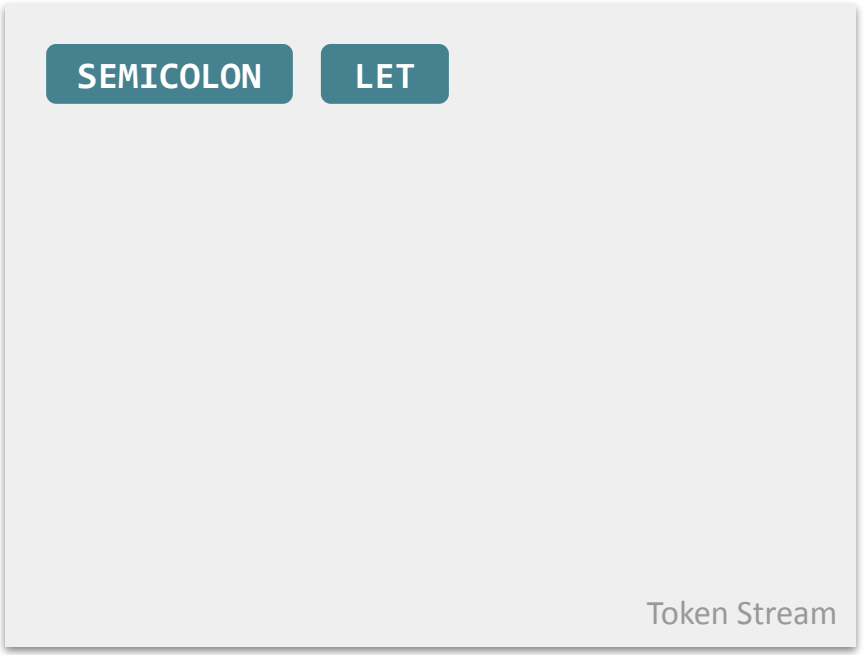




- Observation: many tokens have disjoint starting characters!
- Keep cursor on current char
  - If the char *could* be part of this token, accumulate it
  - If not, complete the current token
- How to distinguish built-in keywords (e.g. “let”) from identifiers (e.g. “bar”)?
  - Solution: when token is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

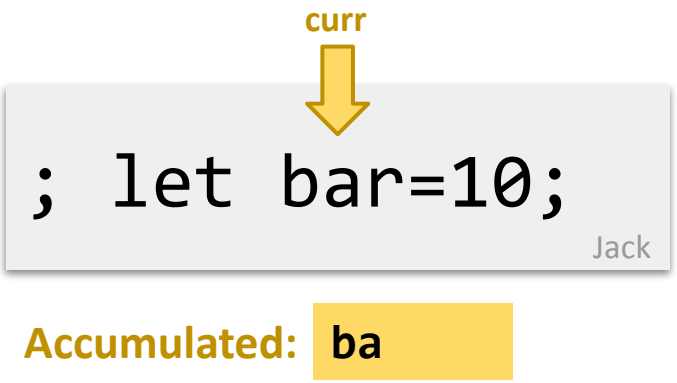
The Scanner: How?

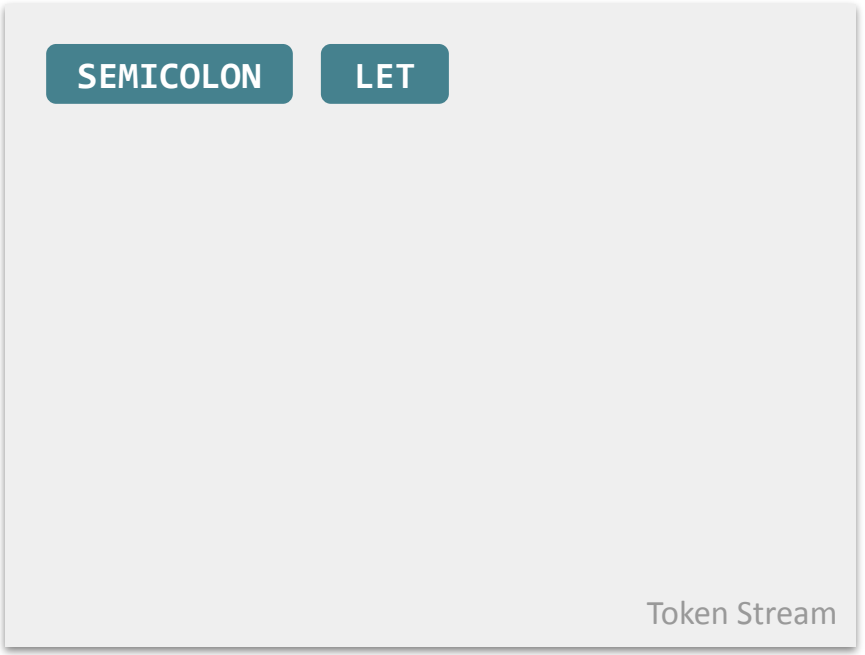




- Observation: many tokens have disjoint starting characters!
- Keep cursor on current char
  - If the char *could* be part of this token, accumulate it
  - If not, complete the current token
- How to distinguish built-in keywords (e.g. “let”) from identifiers (e.g. “bar”)?
  - Solution: when token is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

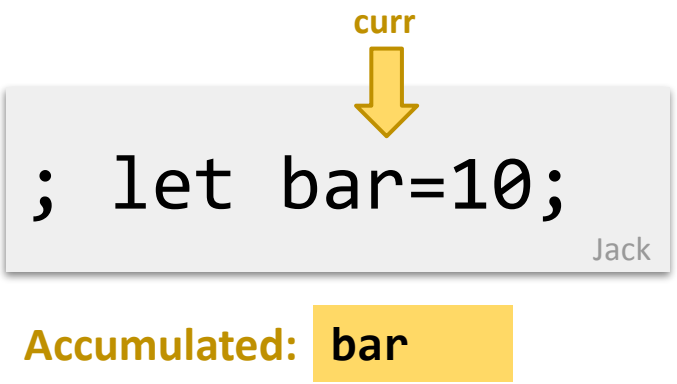
The Scanner: How?

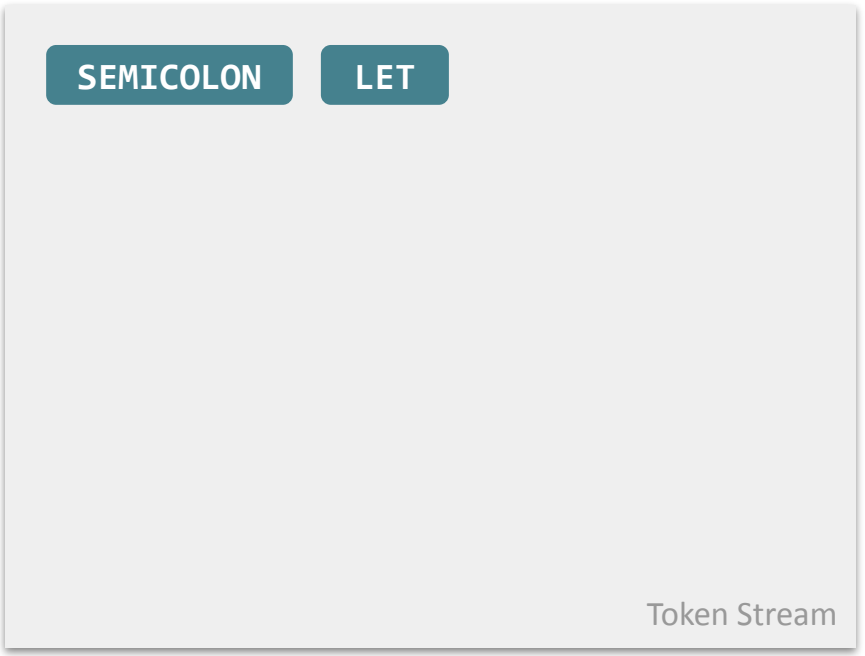




- Observation: many tokens have disjoint starting characters!
- Keep cursor on current char
  - If the char *could* be part of this token, accumulate it
  - If not, complete the current token
- How to distinguish built-in keywords (e.g. “let”) from identifiers (e.g. “bar”)?
  - Solution: when token is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

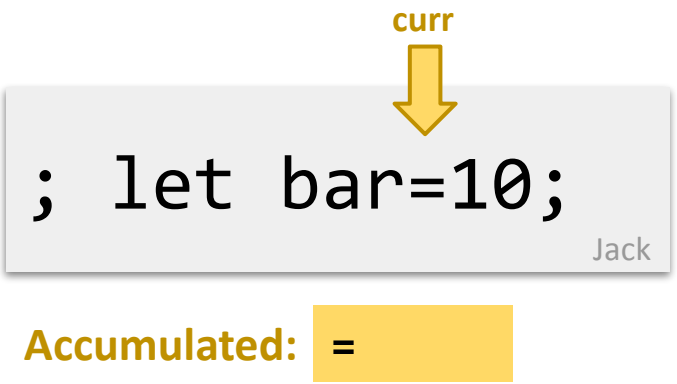
The Scanner: How?

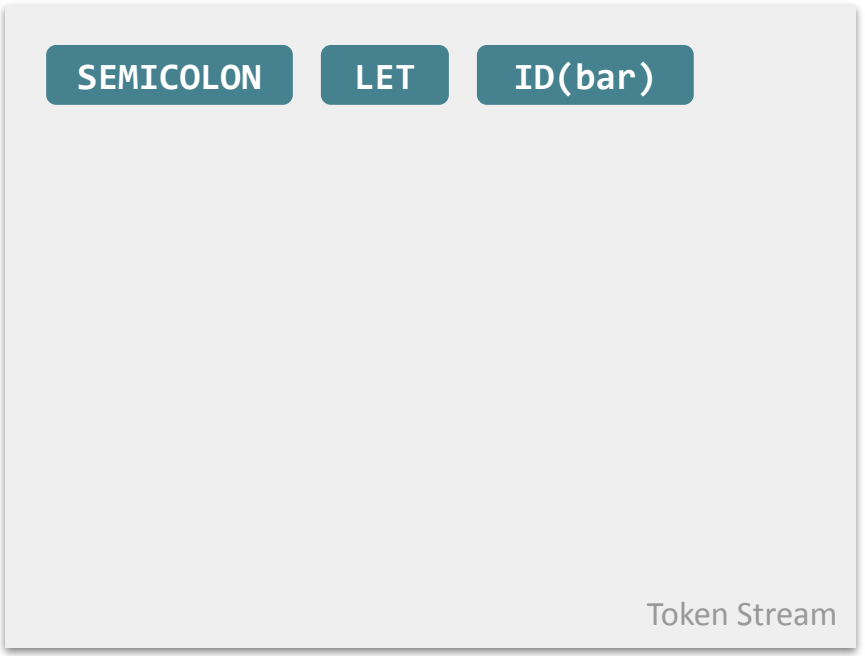




- Observation: many tokens have disjoint starting characters!
- Keep cursor on current char
  - If the char *could* be part of this token, accumulate it
  - If not, complete the current token
- How to distinguish built-in keywords (e.g. “let”) from identifiers (e.g. “bar”)?
  - Solution: when token is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

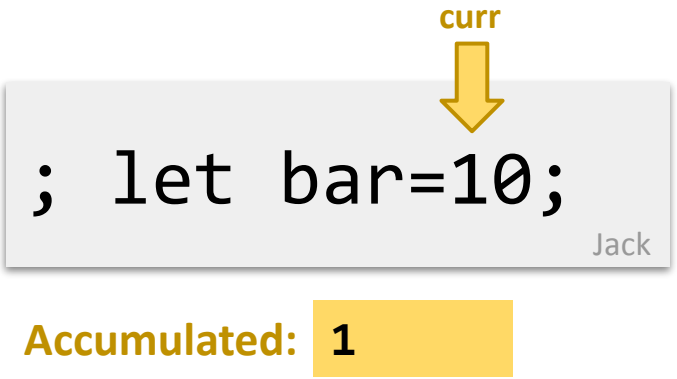
The Scanner: How?

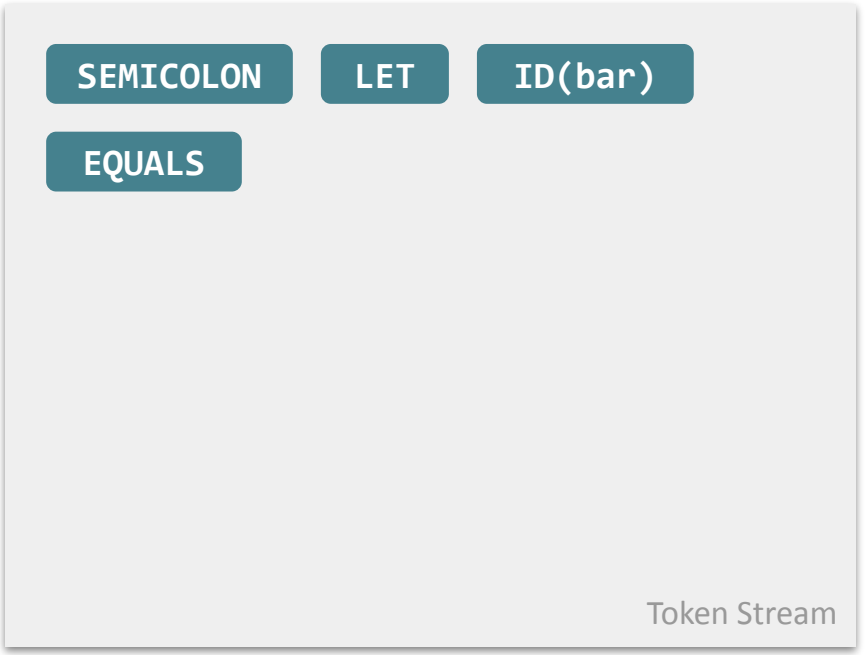




- Observation: many tokens have disjoint starting characters!
- Keep cursor on current char
  - If the char *could* be part of this token, accumulate it
  - If not, complete the current token
- How to distinguish built-in keywords (e.g. “let”) from identifiers (e.g. “bar”)?
  - Solution: when token is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

The Scanner: How?





- Observation: many tokens have disjoint 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”)?
  - Solution: 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

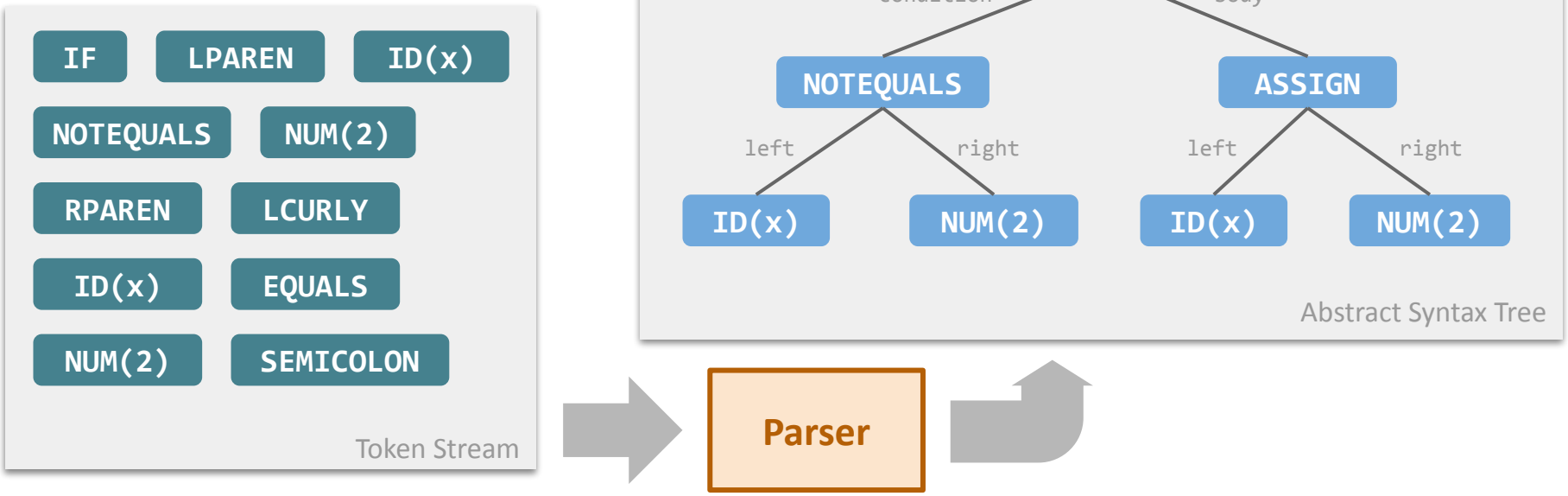
Quick Break (5 min)

- Stretch, get some water and/or food, relax
- Review your notes and figure out if you have any lingering questions from the first part of lecture
- Feel free to ask any questions or chat about anything else!

Agenda

- Reading Q&A
- **Introduction to the Compiler**
  - Overview
  - The Scanner
  - **The Parser**
- Project 6 Overview

The Parser



- Takes in the *flat* token stream and outputs a *structured* tree representation of program constructs
  - Called an **Abstract Syntax Tree**
  - This will be our **Intermediate Representation** used by the rest of the compiler

Describing a Programming Language

- Many (formal) ways to define programming languages
  - We won't cover language definition in depth
  - See 341, 401, 402
- Common Distinction: 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

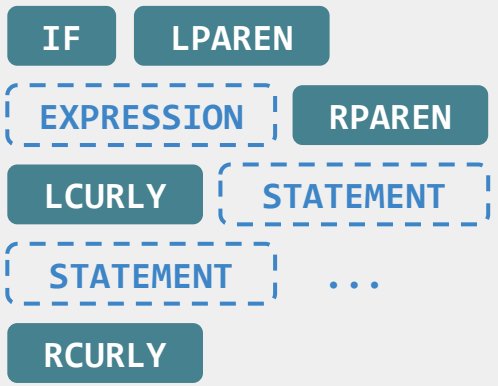
Symbolic Example

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

General Definition of an `if` Statement

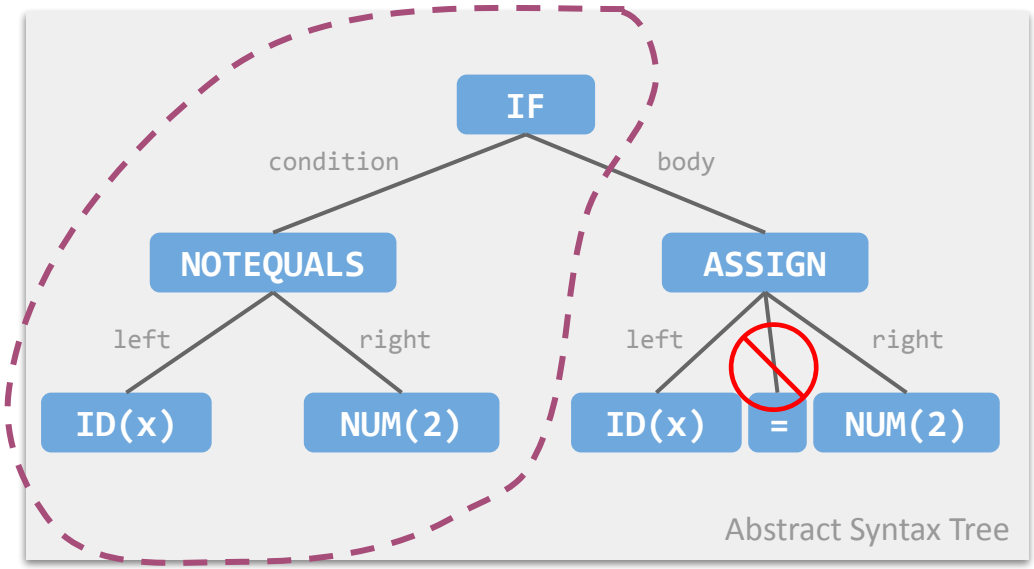
```
if ( EXPRESSION )
{
  STATEMENT
  STATEMENT
    ...
}
```

Token Stream Definition



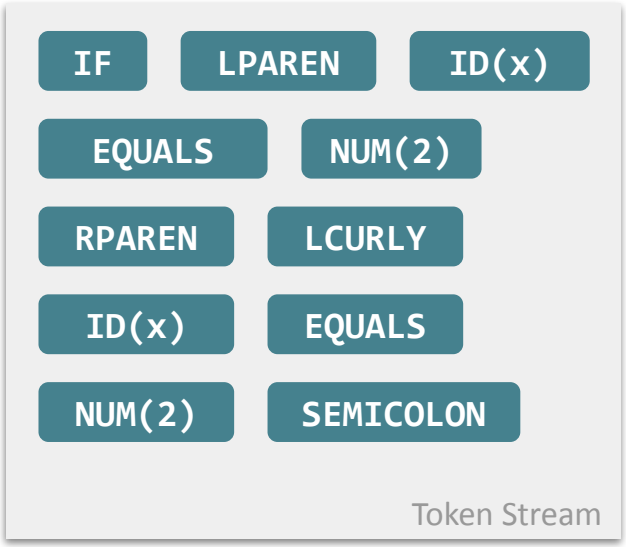
Abstract Syntax Trees

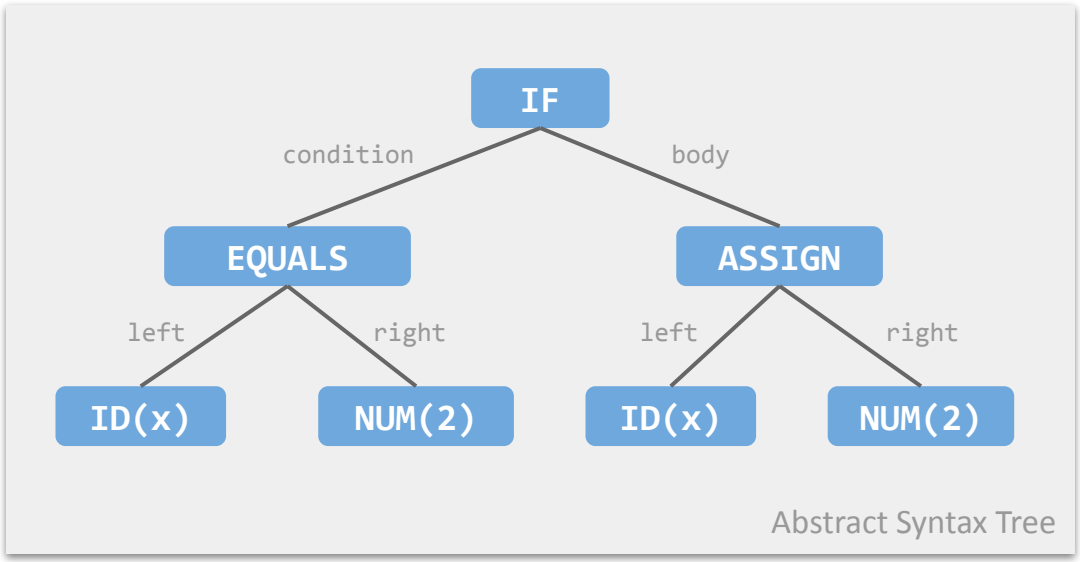
- Composed of nodes that capture the **structure** of input program
  - Can be recursive! (And almost always is)
- *Important distinction:* cares about big-picture syntax (e.g. entire `if` statement) rather than nitty-gritty syntax (e.g. semicolons, parentheses, even word "if" used to write that `if` statement)




```
class If {
  Expression condition; //stores a NotEquals
  List<Statement> body;
}

class NotEquals extends Expression {
  Expression left; //stores a VarAccess
  Expression right; //stores a NumLiteral
}

class VarAccess extends Expression {
  Identifier name; //stores x
}

class NumberLiteral extends Expression {
  int value; //stores 2
}
```

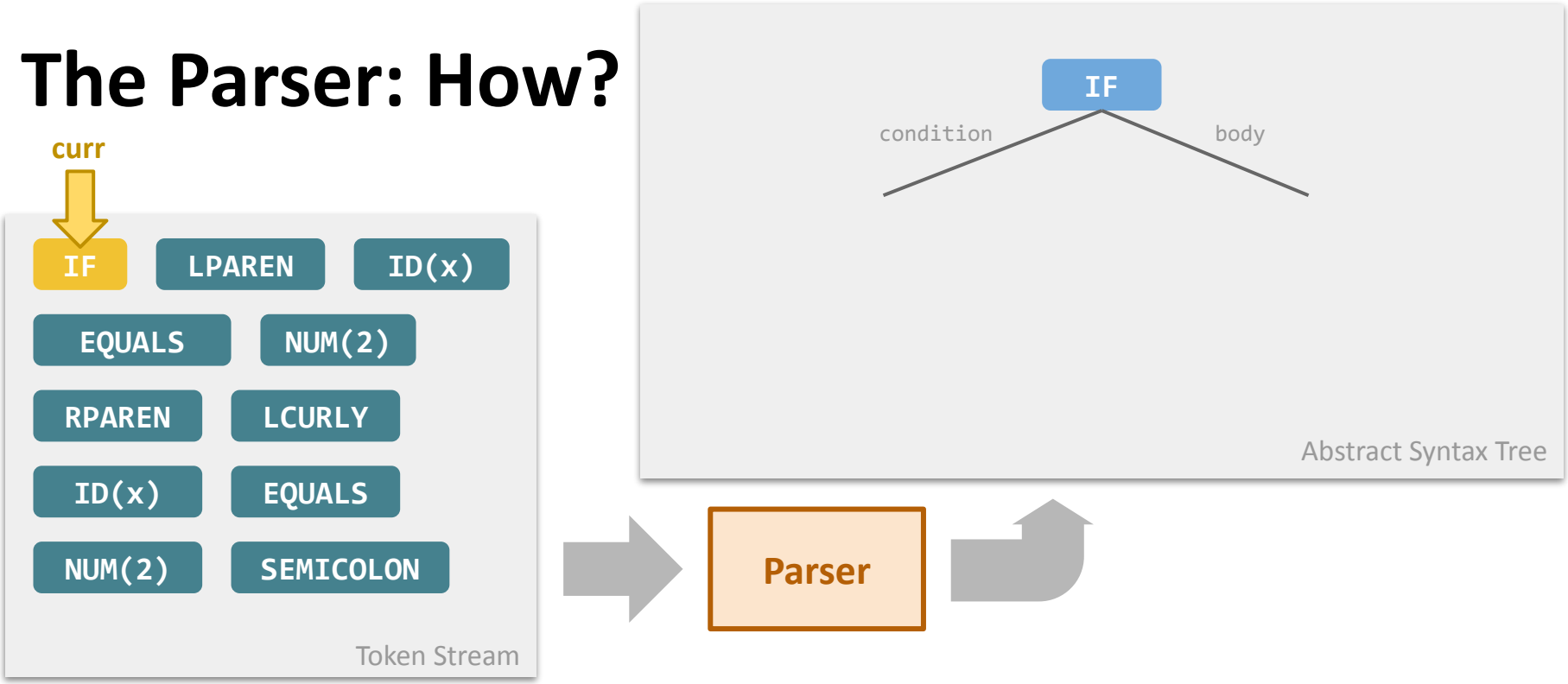
The Parser: How?





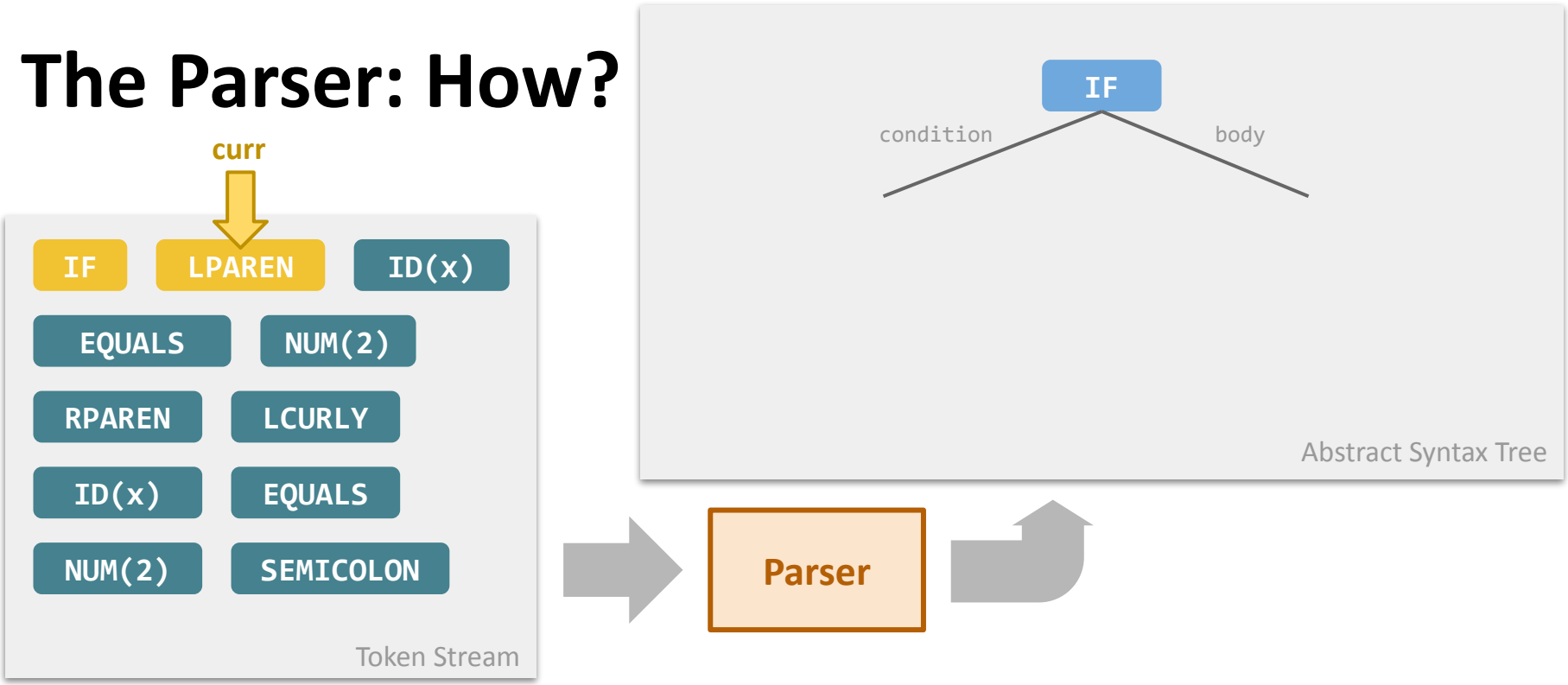
- Similar to scanner: single pass through token stream, building up as we go
- Intuition: If we see `IF` and `LPAREN`, we're entering an `if` statement and next we expect a complete expression
  - So keep reading until we have a complete expression, and attach on the `condition` side of the `IF`

The Parser: How?



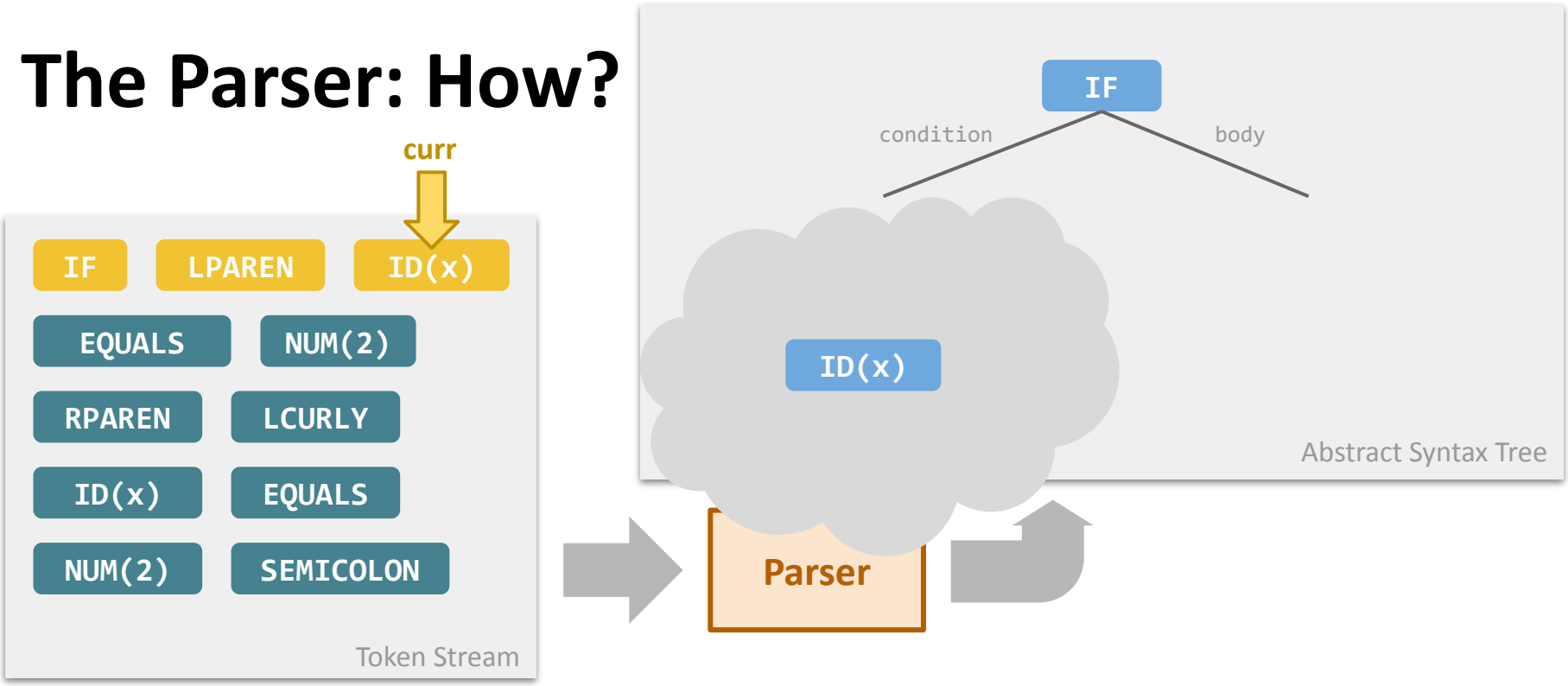
- Similar to scanner: single pass through token stream, building up as we go
- Intuition: If we see `IF` and `LPAREN`, we're entering an `if` statement and next we expect a complete expression
  - So keep reading until we have a complete expression, and attach on the `condition` side of the `IF`

The Parser: How?



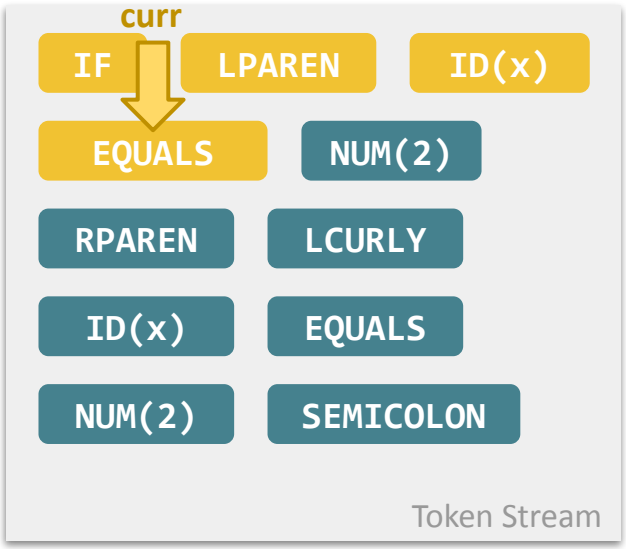
- Similar to scanner: single pass through token stream, building up as we go
- Intuition: If we see `IF` and `LPAREN`, we're entering an `if` statement and next we expect a complete expression
  - So keep reading until we have a complete expression, and attach on the `condition` side of the `IF`

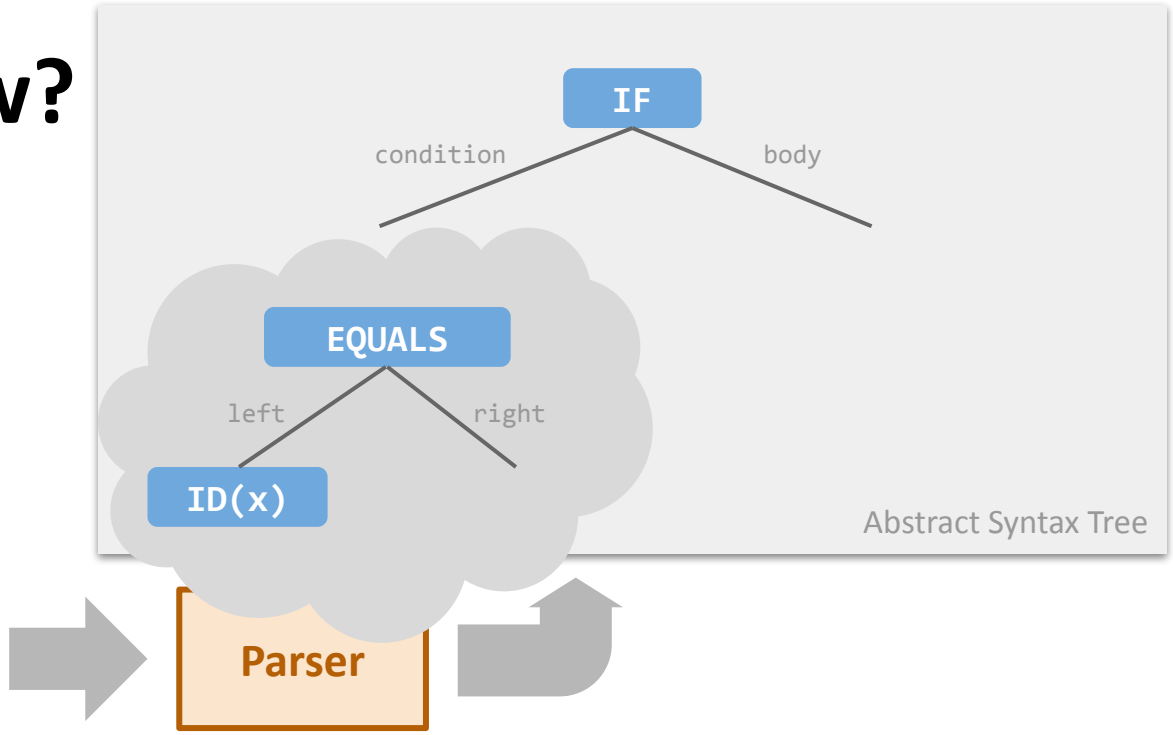
The Parser: How?



- Similar to 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 expect a complete expression
  - So keep reading until we have a complete expression, and attach on the `condition` side of the `IF`

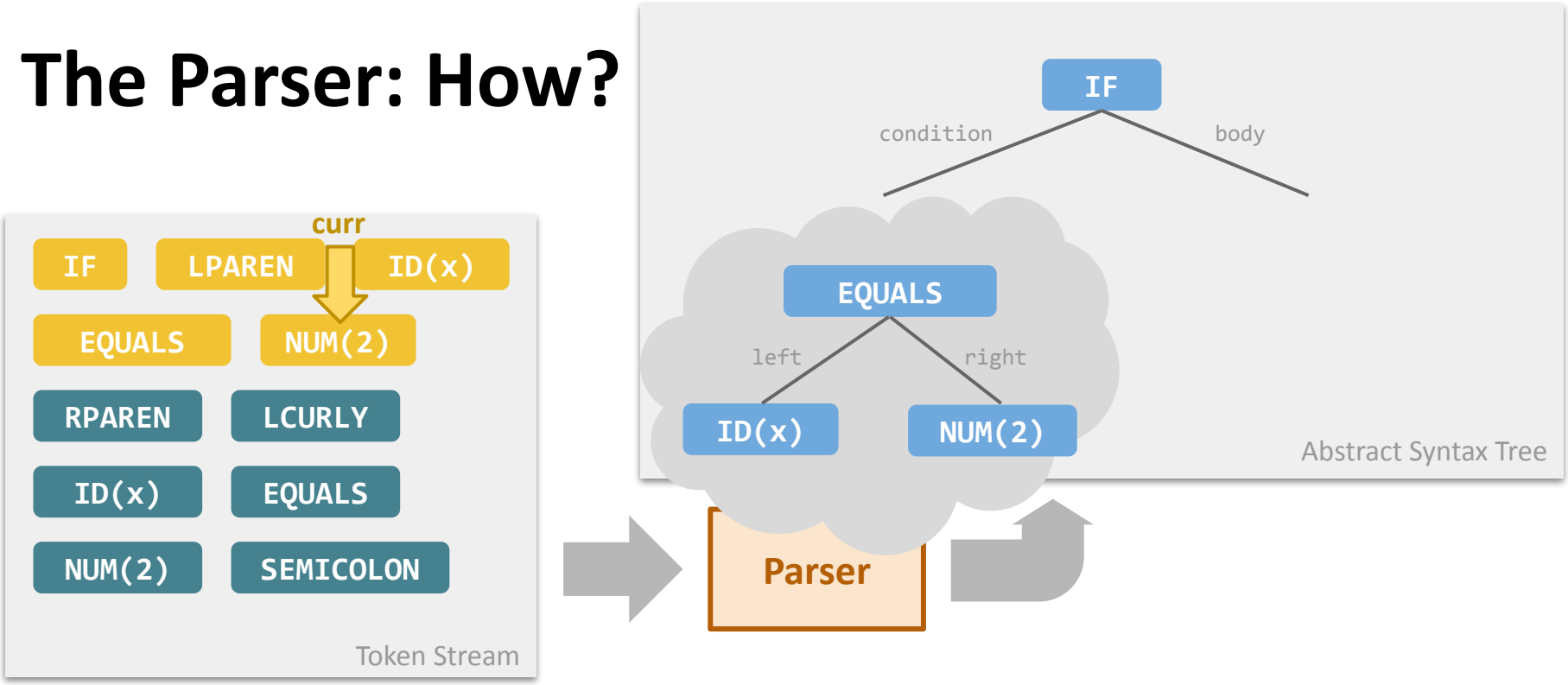
The Parser: How?





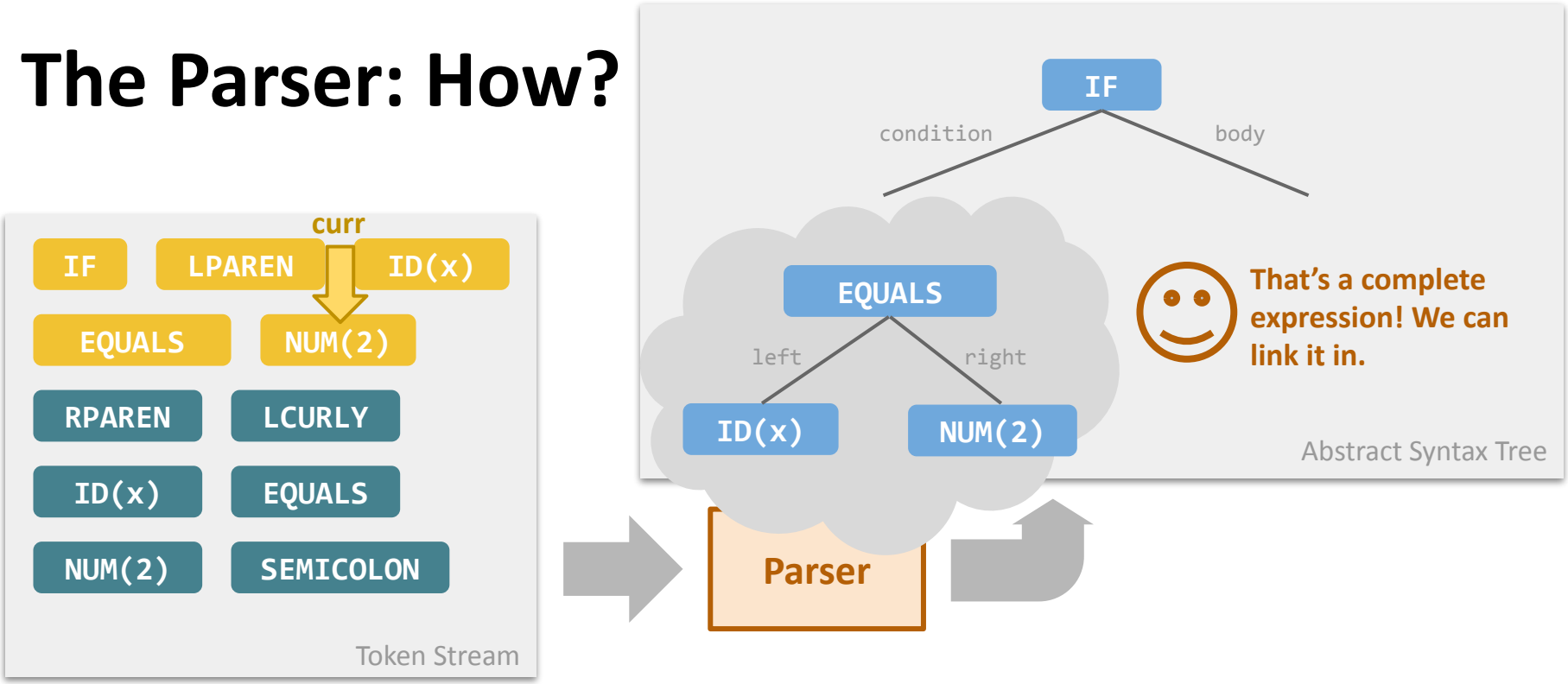
- Similar to scanner: single pass through token stream, building up as we go
- Intuition: If we see `IF` and `LPAREN`, we're entering an `if` statement and next we expect a complete expression
  - So keep reading until we have a complete expression, and attach on the `condition` side of the `IF`

The Parser: How?



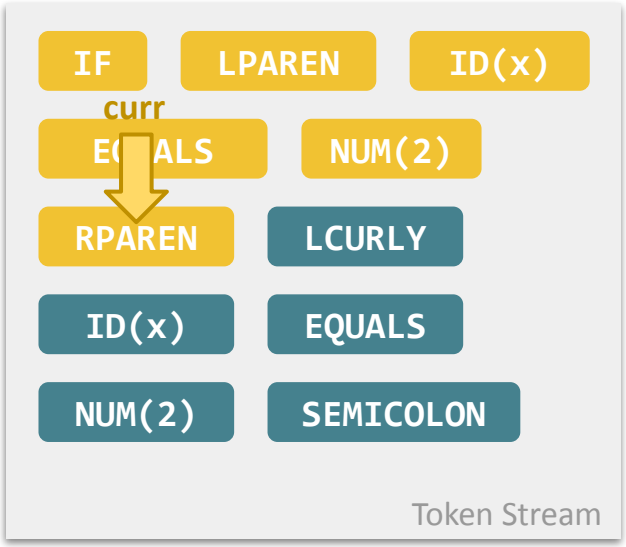
- Similar to scanner: single pass through token stream, building up as we go
- Intuition: If we see `IF` and `LPAREN`, we're entering an `if` statement and next we expect a complete expression
  - So keep reading until we have a complete expression, and attach on the `condition` side of the `IF`

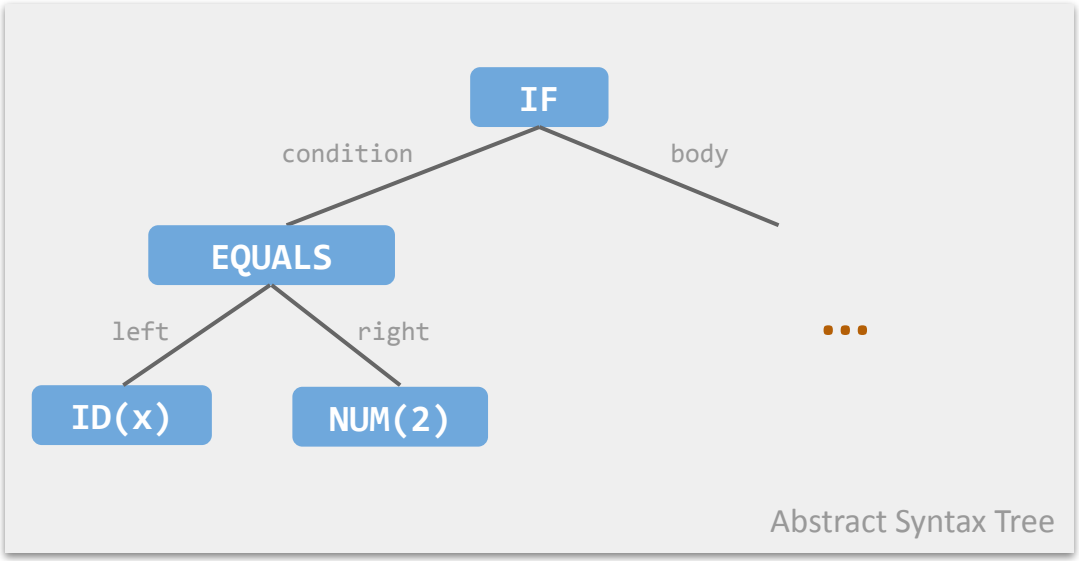
The Parser: How?



- Similar to 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 expect a complete expression
  - So keep reading **until we have a complete expression**, and attach on the `condition` side of the `IF`

The Parser: How?



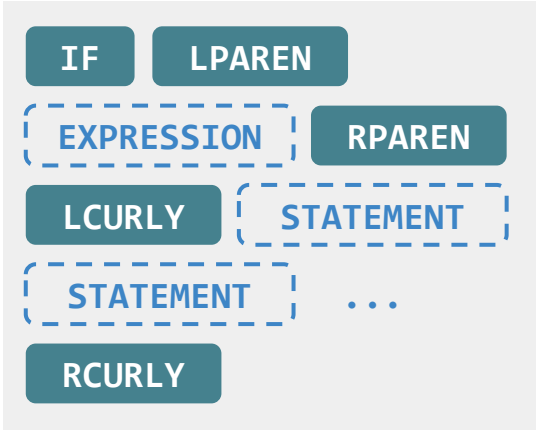


- Similar to scanner: single pass through token stream, building up as we go
- Intuition: If we see `IF` and `LPAREN`, we're entering an `if` statement and next we expect a complete expression
  - So keep reading until we have a complete expression, and attach on the `condition` side of the `IF`

The Parser: How?

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

Token Stream Definition



Pseudocode:

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

    // consumes tokens in expr
    e = parseExpression()

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

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

Agenda

- Reading Q&A
- Introduction to the Compiler
  - Overview
  - The Scanner
  - The Parser
- **Project 6 Overview**

Project 6 Overview

PART I:
Midterm Redo
Due in <u>one</u> 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 <u>two</u> weeks

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

Wrapping Up

What's in store for Week 8?

- More Compiler
- Debugging!
- Project 7 Overview

Reminders

- Project 5 Due tonight 11:59PM PDT
- Schedule your professor meeting ASAP!!!