CSE 390B, Spring 2022

Building Academic Success Through Bottom-Up Computing

Midterm Debrief & Compilers

Midterm Debrief, Introduction to Compilers, Project 7 Overview

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

Midterm Debrief

- Challenging midterm topics with only 60 minutes
- Opportunity to discover your strengths and what your gaps in knowledge are
- Measure of your *performance*, not your *capability*
- Practice fostering a **growth mindset**
  - "My academic performance in CSE 390B can be improved through effort and persistence…"
  - "I'm not where I want to be in this class *yet*…"

Midterm Debrief Discussion

- Both metacognitively and technically, in what areas did you perform strongly in on the midterm? Which areas could be improved?

- What is your plan for making the most out of the experience from this midterm?

- How will you practice fostering a **growth mindset**?
  - "My academic performance in CSE 390B can be improved through effort and persistence…"
  - "I'm not where I want to be in this class *yet*…"

Midterm Debrief Discussion

- Both metacognitively and technically, in what areas did you perform strongly in on the midterm? Which areas could be improved?
- What is your plan for making the most out of the experience from this midterm?
- How will you practice fostering a **growth mindset**?

Midterm Debrief

- Exams are not an objective measure of your learning or abilities
  - They are one type of evaluation and favor certain learning styles
  - They reflect priorities of instructor and are one view of material
  - Performance on an exam doesn't determine your capability
- Yet, our educational system frequently use high-stakes, time-pressured exams
- Our goal is to help you improve your skills related to preparing, taking, and reflecting on exams

Midterm Grading Observations

- Question 1: Boolean Logic & Circuit Design
  - Good job overall on the truth table
  - Use Boolean Function Synthesis to determine Boolean expression
- Question 2: Number Representation & Circuit Design
  - Equals: construct equivalent behavior using Not, And, and Or gates
  - Minimum: make sure to use recommended gates
- Question 3: Sequential Logic Circuit Design
  - Flipping the Mux inputs or passing in flipped select bit

Midterm Grading Observations

- Question 4: Assembly Programming
  - Only allowed to use computations available to the ALU
- Question 5: Assembly Debugging
  - Tricky program to trace through in ~10 minutes

Midterm Next Steps

- 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 the grading
  - This is a great learning opportunity for us all

- You will have a chance to regain points midterm corrections as part of Project 7

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

Vote at https://pollev.com/cse390b

In which phase of the compiler would we check that parentheses takes precedence over arithmetic operations?

A. Scanner
B. Parser
C. Analysis (Type Checking and Optimization)
D. Code Generation
E. We're lost…

Software Overview

High-Level Language

Java
Python
C/C++
Jack

Compiler

Compiler

(Project 8)

Intermediate Language(s)

Java Byte Code
Jack VM Code

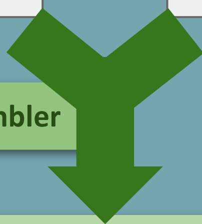
Compiler (VM Translator)

Assembly Language

x86, x86-64
ARM
RISC-V
HACK

Operating System

Windows
Mac
Unix/Linux
Android
Hack OS



Assembler

Machine Code

SOFTWARE

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

Scanner → Parser → Type Checker → Optimizer → Code Generator

Break string into discrete **tokens**:

`IF` `(` `ID(n)` `==` `NUM(0)` etc.

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

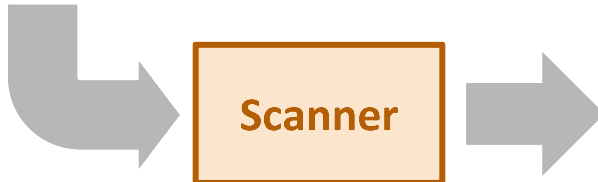
```
function void main() {
  var int a, bar;
  let bar = 10;
}

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

Jack

The Scanner

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

Jack



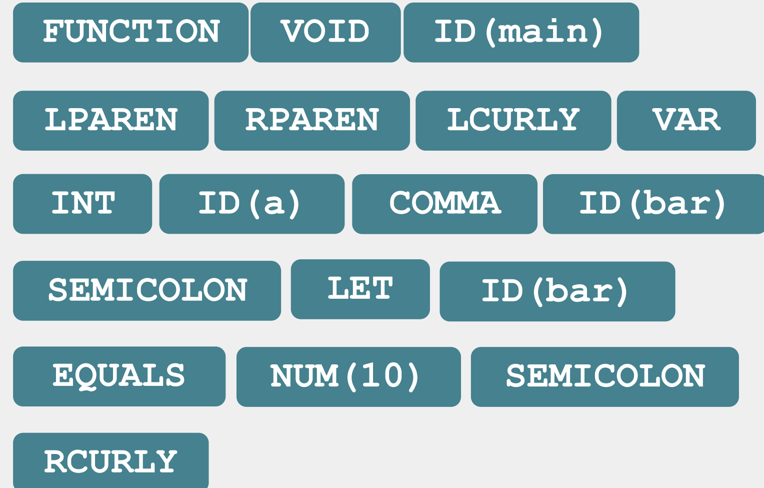
FUNCTION VOID ID(main)
LPAREN RPAREN LCURLY VAR
INT ID(a) COMMA ID(bar)
SEMICOLON LET ID(bar)
EQUALS NUM(10) SEMICOLON
RCURLY

Token Stream

The Scanner

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

Jack



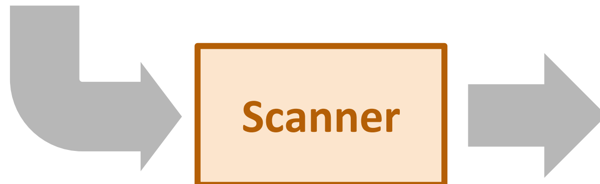
Token Stream

- 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

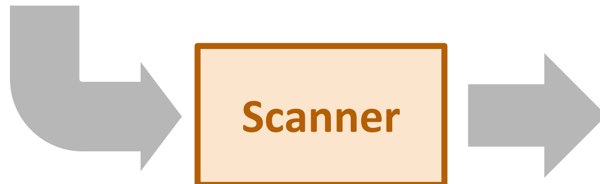
Token Stream

- 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

```
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?

curr

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

Jack

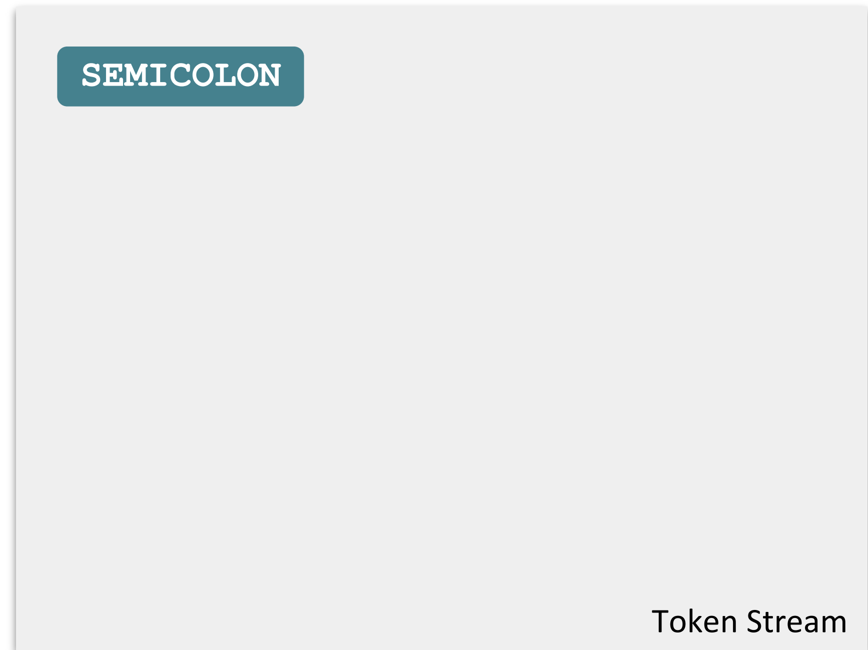
Accumulated: ;

Token Stream

- 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”)?
  - When token is done, check against list of keywords

The Scanner: How?

Accumulated:



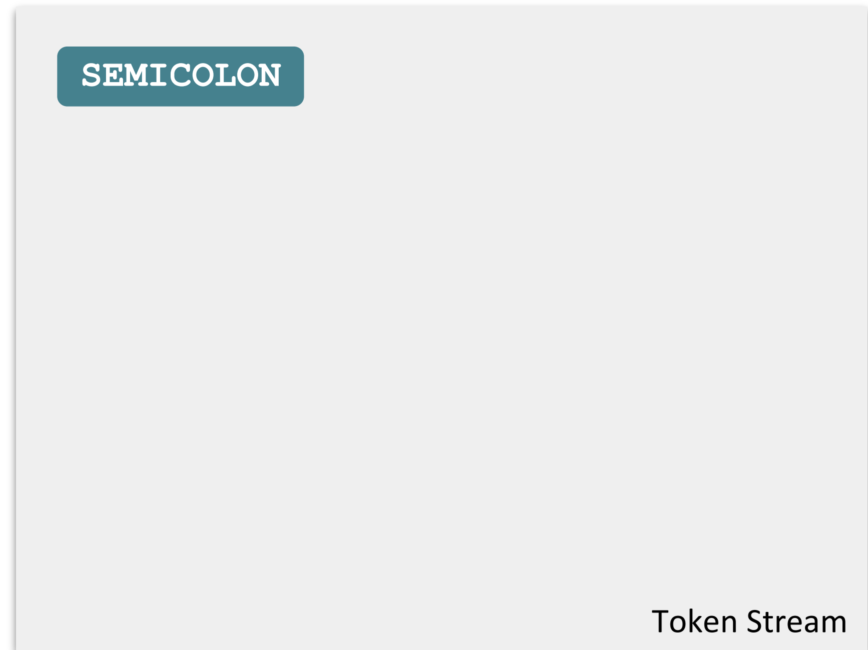
- Observation: many tokens have disjoint starting characters
- Keep cursor on current char
  - If the char *could* be part of this token, accumulate it
  - If not, complete the current token
- How to distinguish built-in keywords (e.g., "let") from identifiers (e.g., "bar")?
  - Simple: when token is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

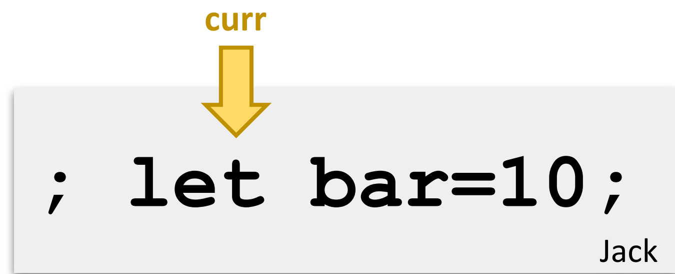
The Scanner: How?

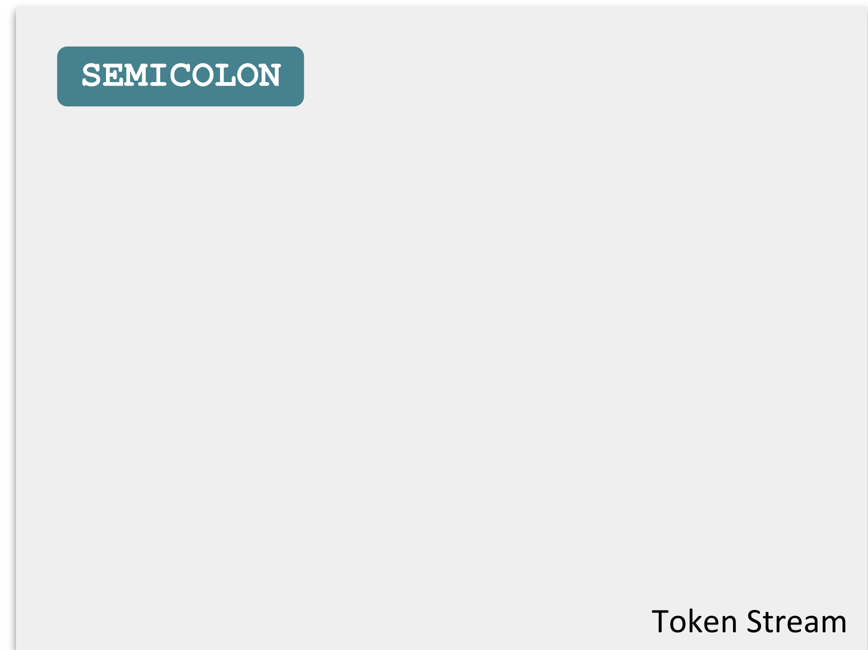
Accumulated: le



- 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")?
  - Simple: when token is done, check against list of keywords

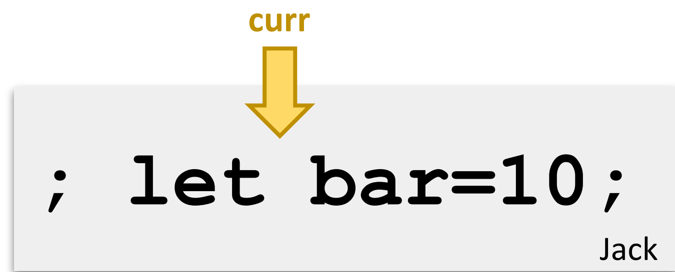
The Scanner: How?

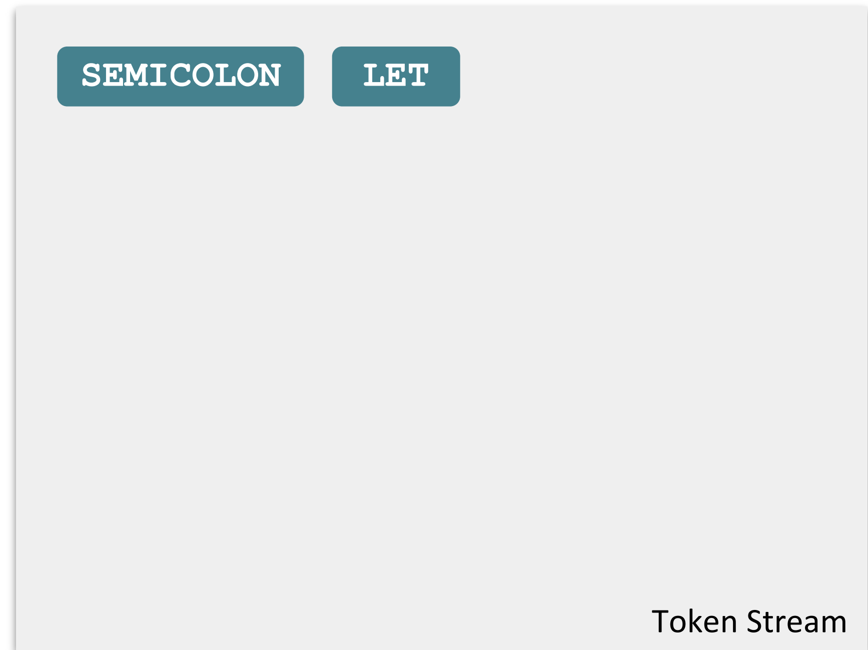




- Observation: many tokens have disjoint starting characters
- Keep cursor on current char
  - If the char *could* be part of this token, accumulate it
  - If not, complete the current token
- How to distinguish built-in keywords (e.g., "let") from identifiers (e.g., "bar")?
  - Simple: when token is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

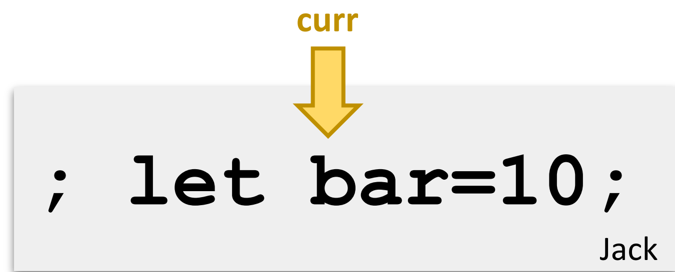
The Scanner: How?

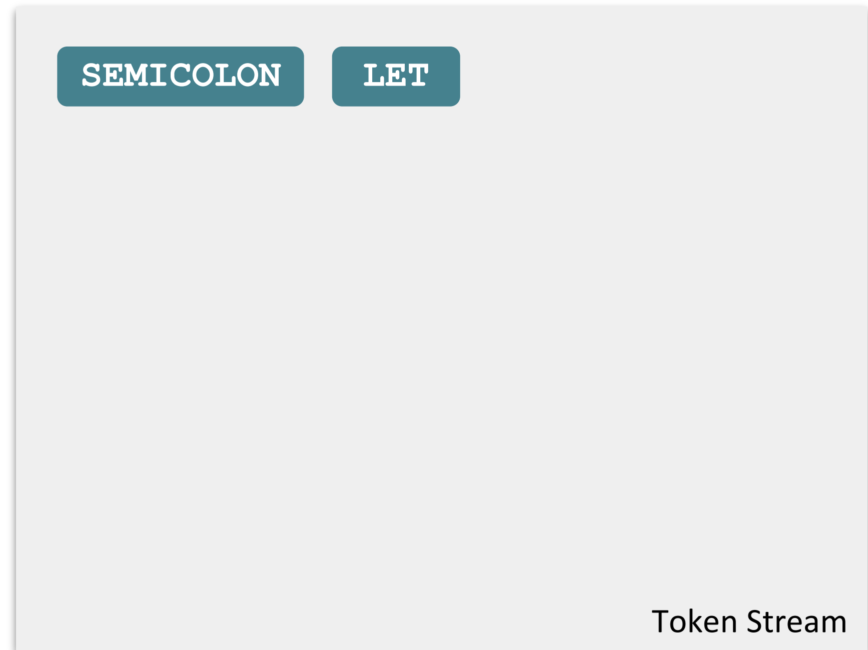




- Observation: many tokens have disjoint starting characters
- Keep cursor on current char
  - If the char *could* be part of this token, accumulate it
  - If not, complete the current token
- How to distinguish built-in keywords (e.g., "let") from identifiers (e.g., "bar")?
  - Simple: when token is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

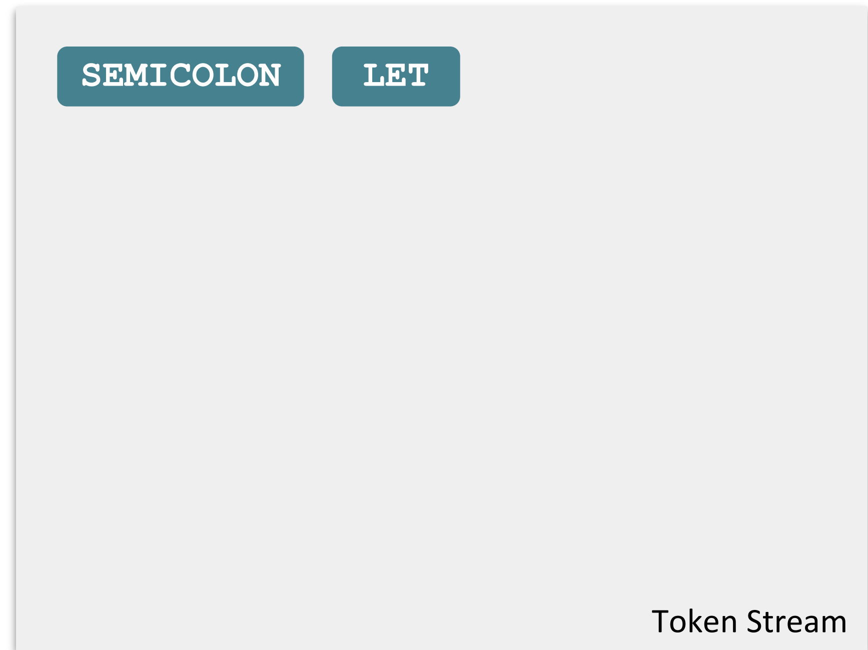
The Scanner: How?

curr

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

Jack

Accumulated: `ba`



- 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”)?
  - Simple: when token is done, check against list of keywords

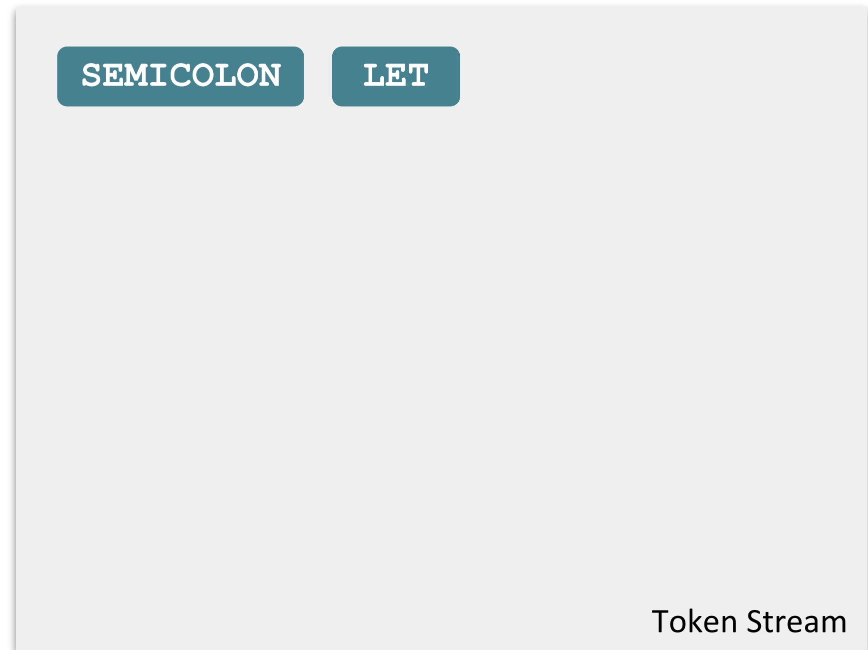
The Scanner: How?

curr

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

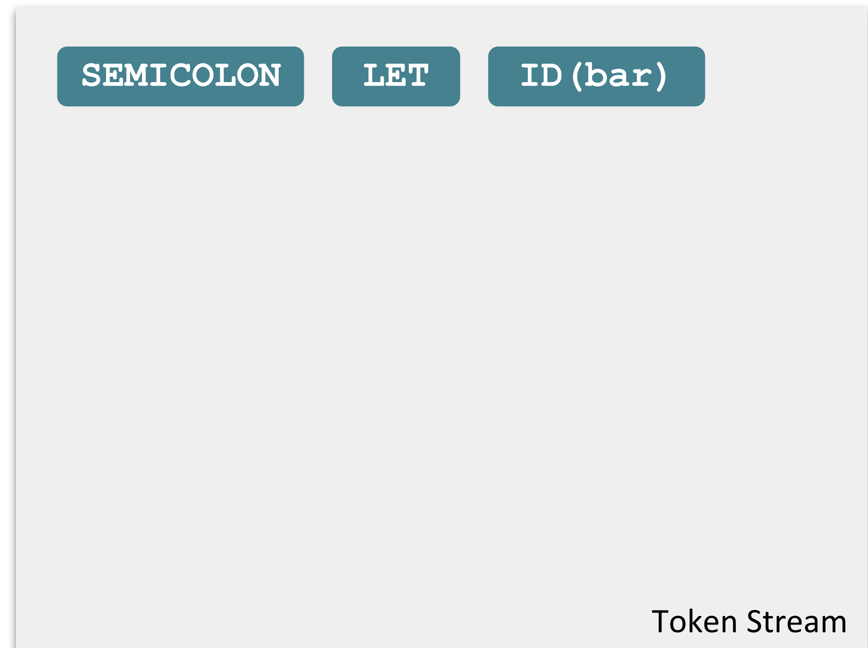
Jack

Accumulated: `bar`



- 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")?
  - Simple: 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”)?
  - Simple: when token is done, check against list of keywords

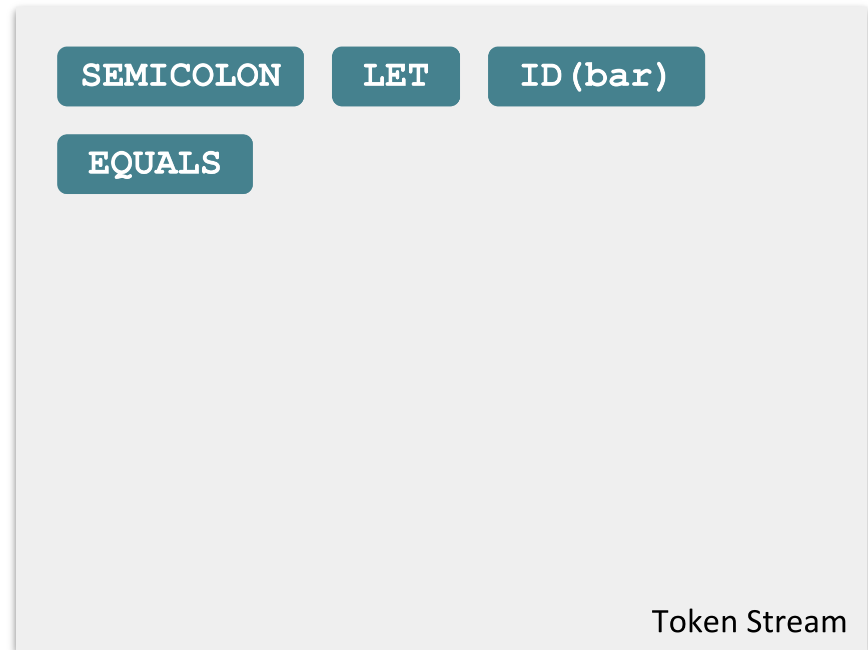
The Scanner: How?

curr

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

Jack

Accumulated: 1



- 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")?
  - Simple: when token is done, check against list of keywords

Why Have a Scanner?

- 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

Five-minute Break!

- Feel free to stand up, stretch, use the restroom, drink some water, review your notes, or ask questions

- We'll be back at: 3:30pm

- Research shows mid-lecture breaks reduce the decline of attention in the middle of lecture (Olmsted, 1999)

Olmsted III, John. "The Mid-Lecture Break: When Less Is More." *Journal of Chemical Education* (1999). https://pubs.acs.org/doi/abs/10.1021/ed076p525.

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

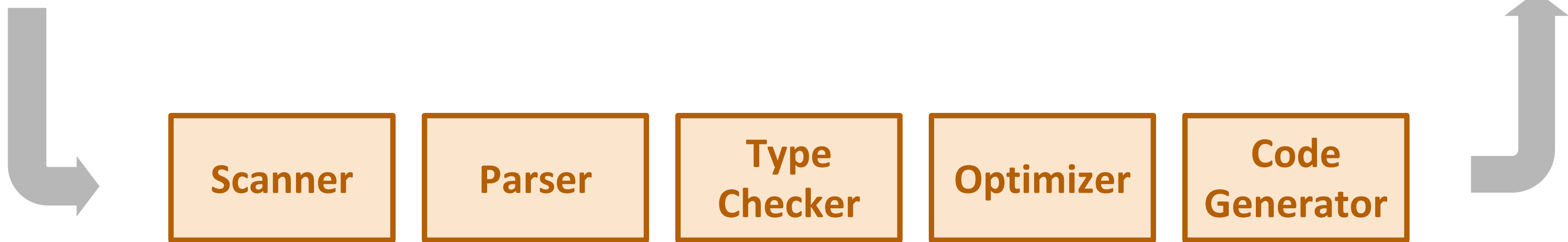
The Compiler: Implementation

```
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
```

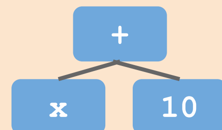
Assembly Language



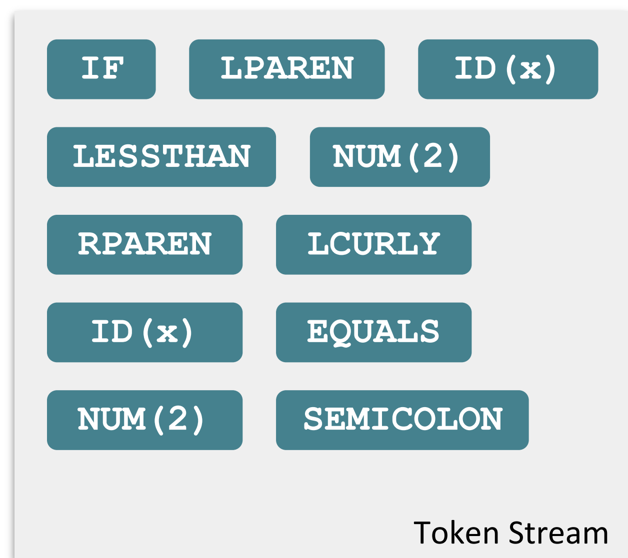
Break string into discrete **tokens**:

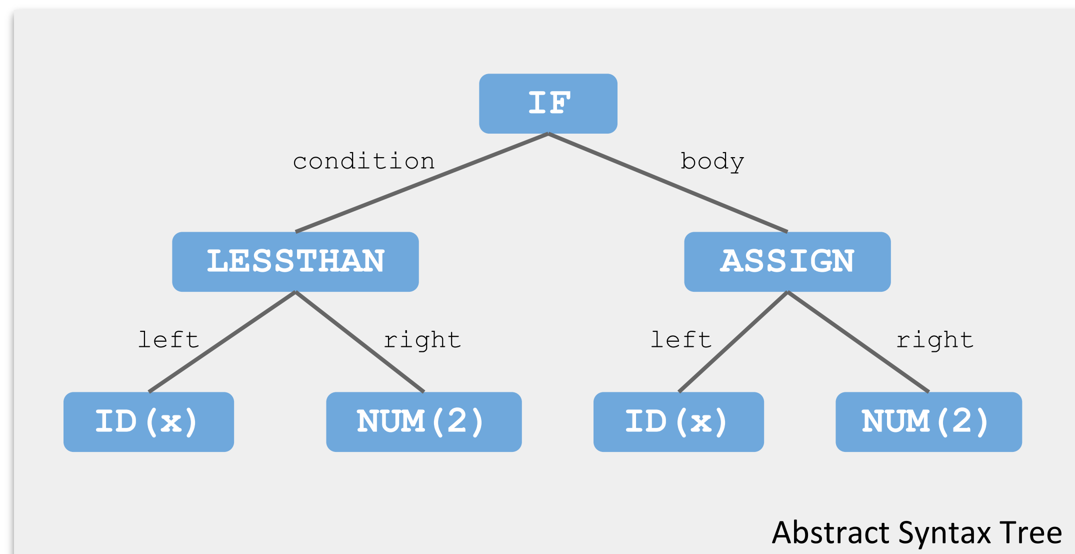
IF | (| ID(n)
== | NUM(0) etc.

Arrange tokens into **syntax tree**:



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
  - Explored in 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

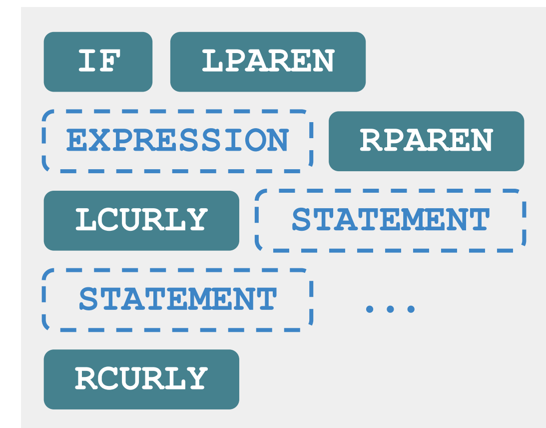
Symbolic Example

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

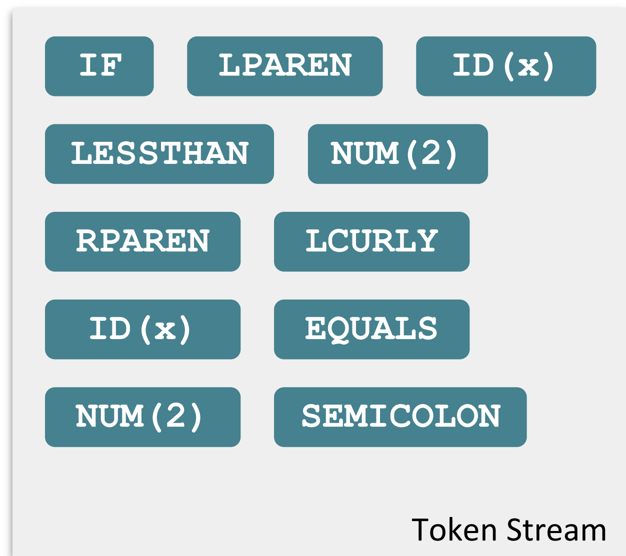
General Definition of an `if` Statement

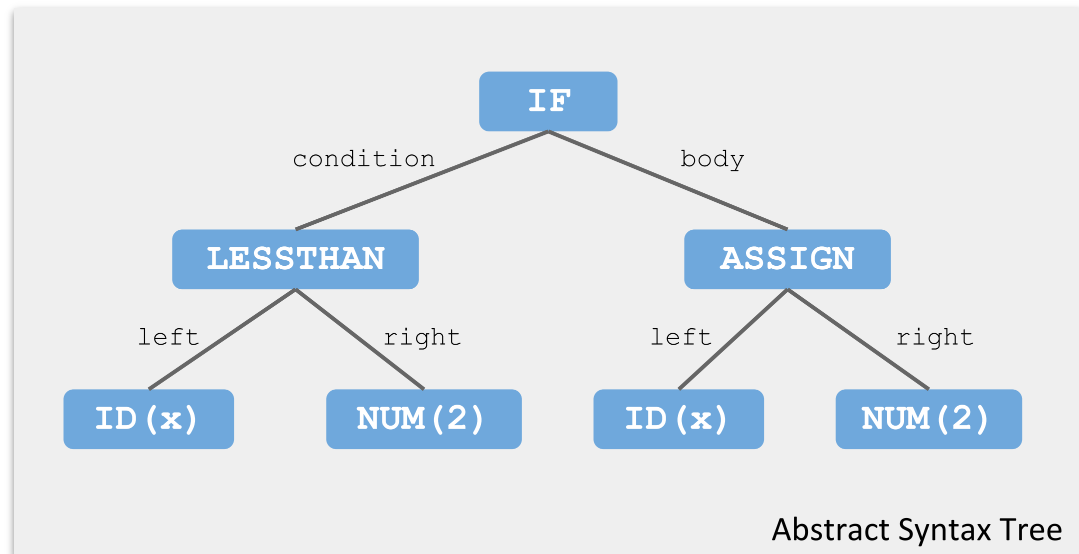
```
if ( EXPRESSION )
{
    STATEMENT
    STATEMENT
    ...
}
```

Token Stream Definition



The Parser: How?



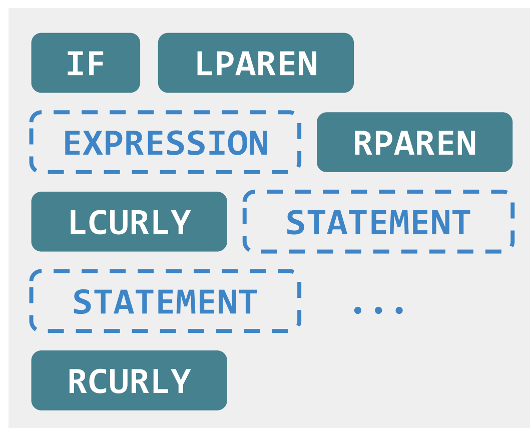


- Like scanner: single pass-through token stream, building up as we go
- Intuition: If we see `IF` and `LPAREN`, we are entering an if statement and next we must see a complete expression
  - Keep reading until we have a complete expression (recursively parse that) and attach on the condition side of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

The Parser: How?

- Parser implementation: 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 stmt
    s = parseStatement()
    ...
    return new If(e, s)
  }
  ...
}
```

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

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
```

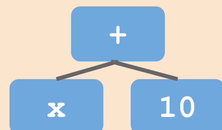
Assembly Language

Scanner → Parser → Type Checker → Optimizer → Code Generator

Break string into discrete **tokens**:

IF | (| ID(n) | == | NUM(0) etc.

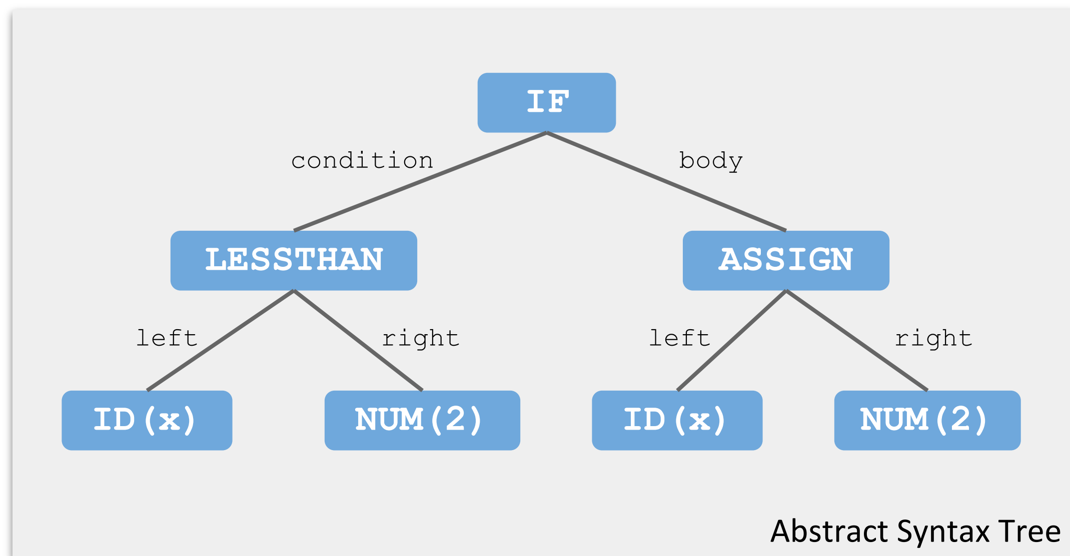
Arrange tokens into **syntax tree**:



Verify the syntax tree is **semantically correct**

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



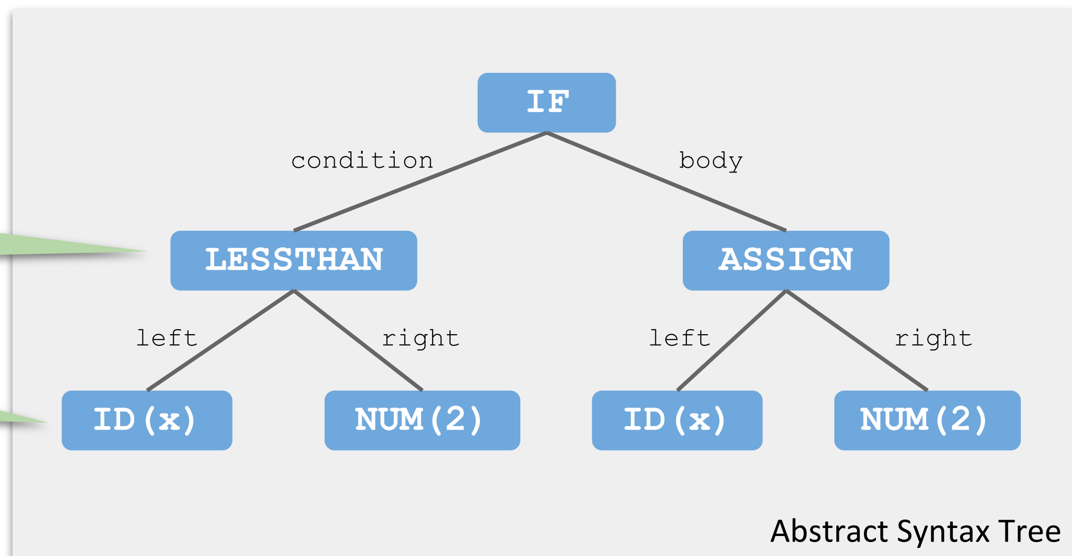
Abstract Syntax Tree

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?



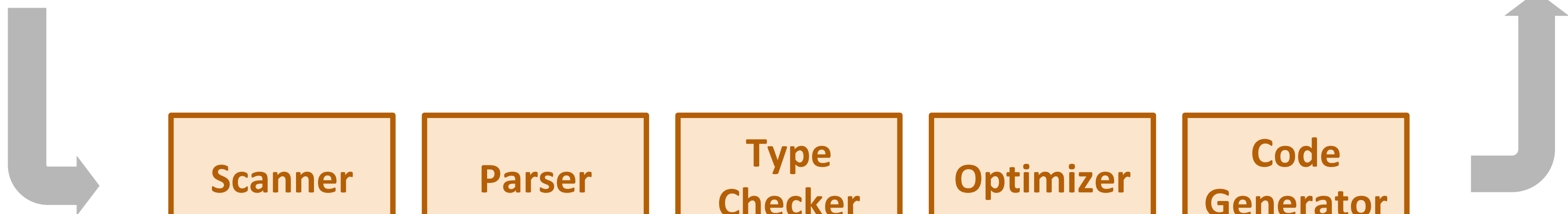
The Compiler: Implementation

```
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
```

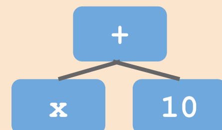
Assembly Language



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**

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

The Compiler: Implementation

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

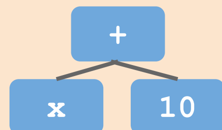
High-Level Language

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

Assembly Language

Scanner → **Parser** → **Type Checker** → **Optimizer** → **Code Generator**

- **Scanner:** Break string into discrete **tokens**: `IF` `(` `ID(n)` `==` `NUM(0)` etc.
- **Parser:** Arrange tokens into **syntax tree**:



- **Type Checker:** Verify the syntax tree is **semantically correct**
- **Optimizer:** Rearrange the code to be **more efficient**
- **Code Generator:** Convert the syntax tree to the **target language**

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

- Many engineering details
  - Example: if you want a stack frame and calling conventions for function calls, we must implement them ourselves via instructions generated by the compiler every time it sees a function call

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 5/17 at 11:59pm PDT (no late days can be used on this part)
  - Open-note, open-tool
  - 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
  - Utilize the course staff for support

- Part II: Professor Meeting Report
  - Due in two weeks on 5/24 at 11:59pm PDT
  - Schedule the meeting early
  - Please do not tell your professor this is for an assignment

Lecture 14 Wrap-up

- Exciting topics for Week 8!
  - Metacognitive Subjects: Debugging and Student Well-being
  - Technical Subject: Code Generation and Two-Tier Compilation

- Project Reminders
  - **Project 6 due tonight (5/12) at 11:59pm PST**
  - Project 7, Part I (Midterm Corrections) due 5/17 (no late days)
  - Project 7, Part II (Professor Meeting Report) due 5/24
  - Schedule your professor meeting ASAP!

- Come to office hours for midterm questions and advice for meeting with a professor—we're happy to help!