# **6.110** Computer Language Engineering

## **Recitation 1:** Project overview/phase 1

February 9, 2024

# Before we get started…

- Recitations are new this year
- We’d appreciate your feedback! Here are some ways to give us feedback:
    - Weekly check-in forms
    - Piazza posts (can be fully anonymous)

**Announcements ←**

Weekly updates

Project overview

Phase 1 details

# Re-lectures

- Re-lectures will be **Wednesdays 4-6pm**, starting this upcoming Wednesday.
- Re-lectures will be recorded.
- Location TBD, look for an announcement on Piazza by Monday.

# Office Hours

- **Monday 4-6pm:** Tarushii
- **Thursday 4-6pm:** Yoland
- **Friday 2-4pm:** Pleng
- **Friday 4-7pm:** Krit

Rooms TBD, will be posted on Piazza as soon as we get room confirmations.

Announcements

**Weekly updates ←**

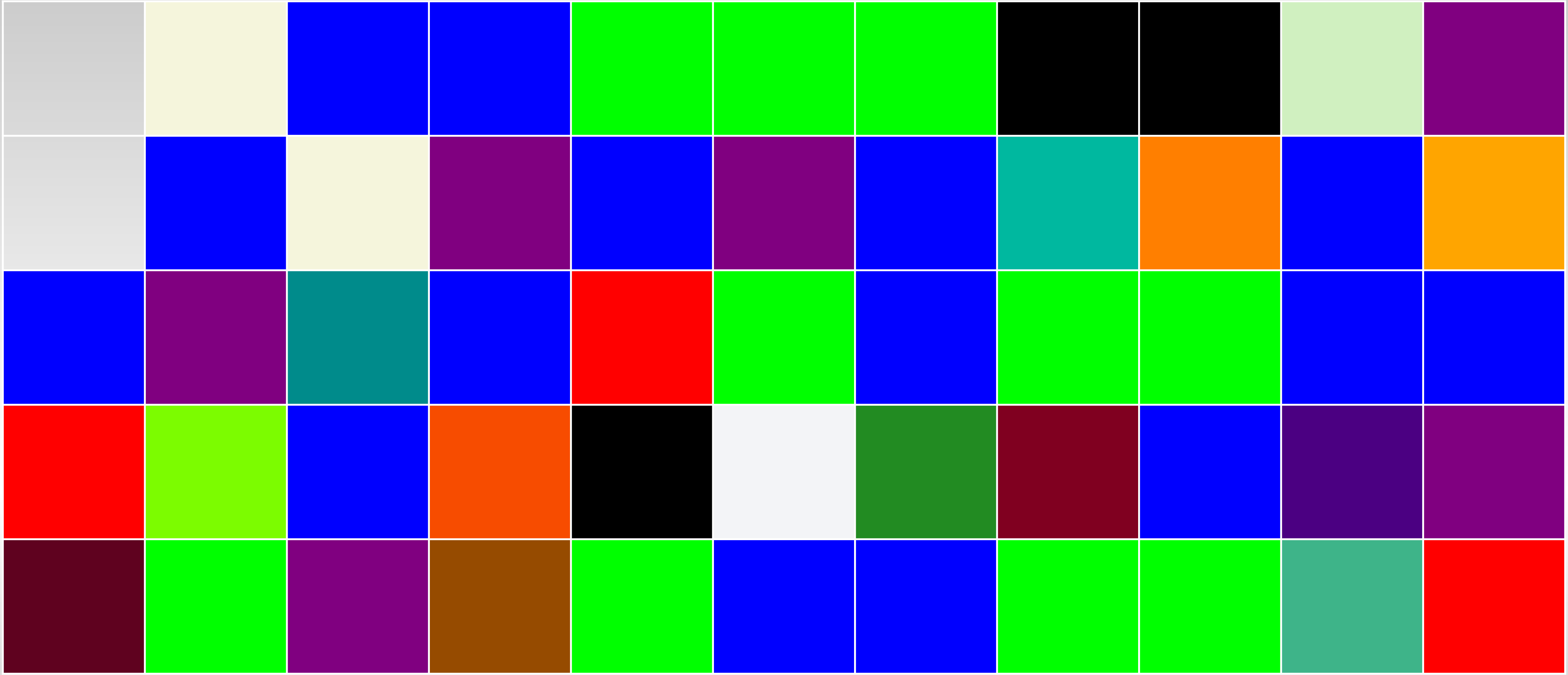
Project overview

Phase 1 details

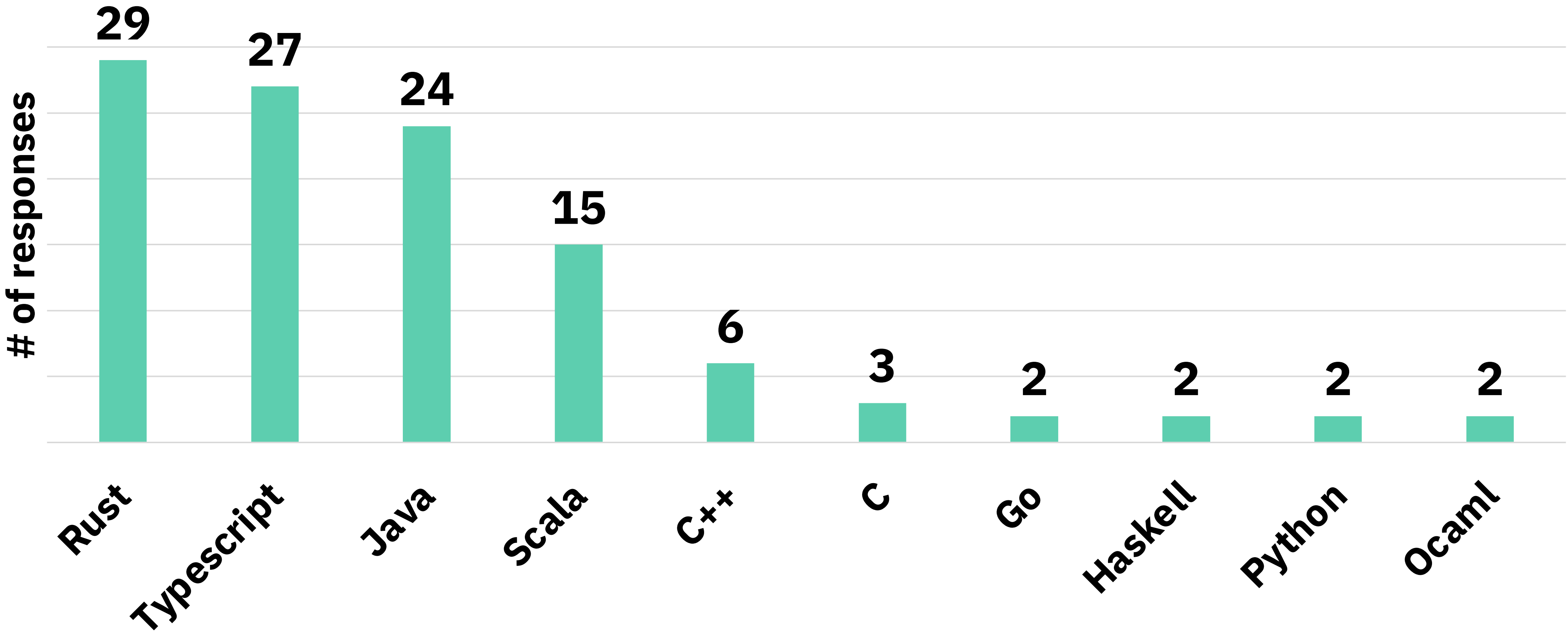
# Fresh off the press

- Project phase 1: due **Friday, February 23**
- Mini-quiz 1 and Weekly Check-in 2: due **Thursday, February 15**
- If you haven't submitted Weekly Check-in 1 yet, **please do so ASAP**.
  - We need your GitHub account to create your phase 1 repository.
  - Future assignments must be submitted on time!

# Check-in 1: Colors



# Check-in 1: Languages



# Coming up soon... **Week 2**

| **Mon**<br>**2/12** | **Tue**<br>**2/13** | **Wed**<br>**2/14** | **Thu**<br>**2/15** | **Fri**<br>**2/16** |
|---|---|---|---|---|
| **Lecture**<br>Top-down parsing | **Lecture** | **Lecture** | **Lecture** | **Recitation**<br>Scanning and parsing a toy language |
|  |  | **Re-lecture**<br>for Week 1 lectures | **Due:** Mini-quiz, weekly check-in |  |

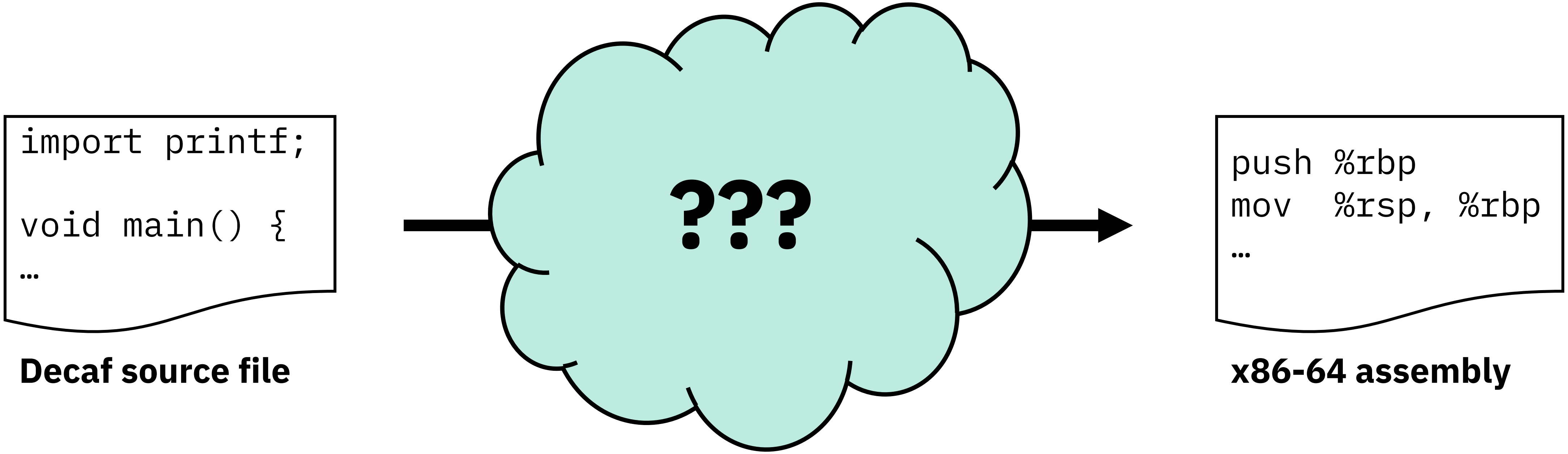
Announcements

Weekly updates

**Project overview ←**

Phase 1 details

# Project overview



```
import printf;

void main() {
…
```

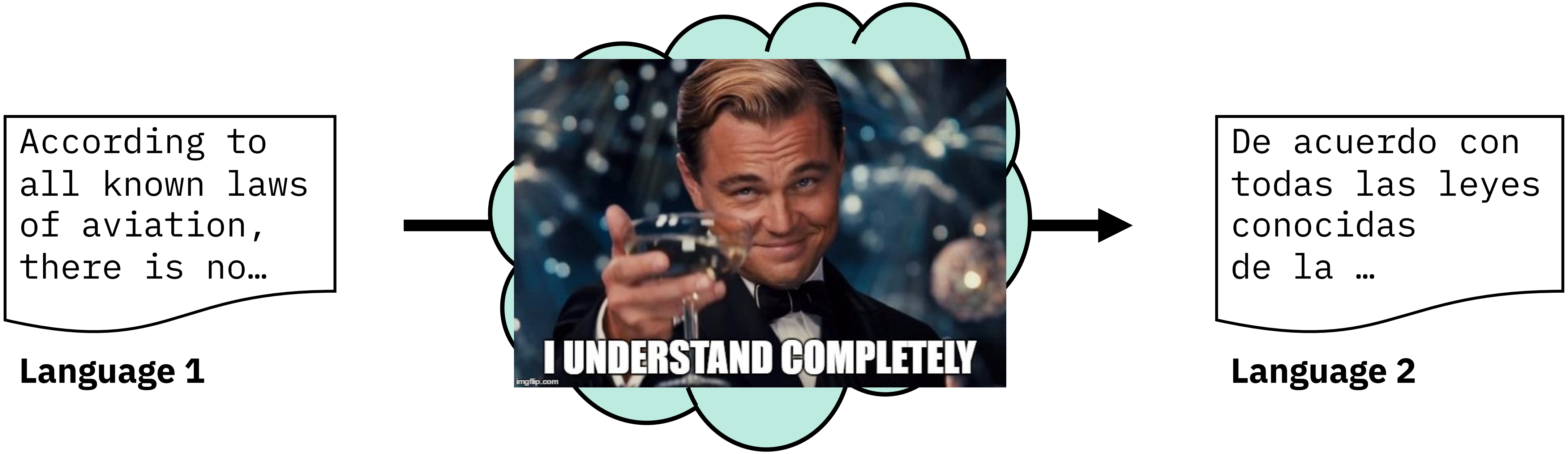
**Decaf source file**

???

```
push %rbp
mov  %rsp, %rbp
…
```

**x86-64 assembly**

# Project overview



According to all known laws of aviation, there is no…

**Language 1**

De acuerdo con todas las leyes conocidas de la …

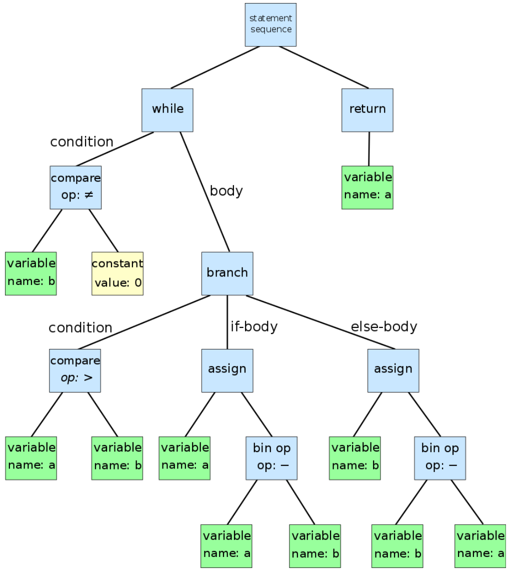
**Language 2**

# Project overview

```
import printf;

void main() {
…
```

**Decaf source file**



**Internal representation**

```
push %rbp
mov  %rsp, %rbp
…
```

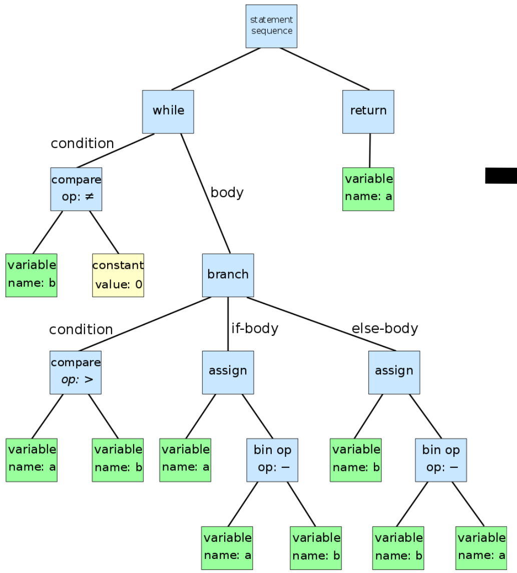
**x86-64 assembly**

# Project overview

```
import printf;

void main() {
…
```

**Decaf source file**



**Internal representation**

```
push %rbp
mov  %rsp, %rbp
…
```

**x86-64 assembly**

**Phase 1.** Does it have the right structure? (syntax)

**Phase 2.** Does it make sense? (semantics)

# Project overview

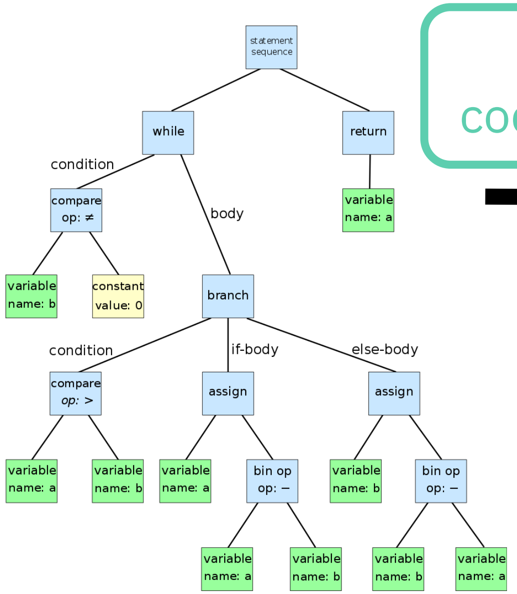
```
import printf;

void main() {
…
```

**Decaf source file**

**Phase 1.** Does it have the right structure? (syntax)

**Phase 2.** Does it make sense? (semantics)



**Internal representation**

**Phase 3**
code generation

```
push %rbp
mov  %rsp, %rbp
…
```

**x86-64 assembly**

# Project overview

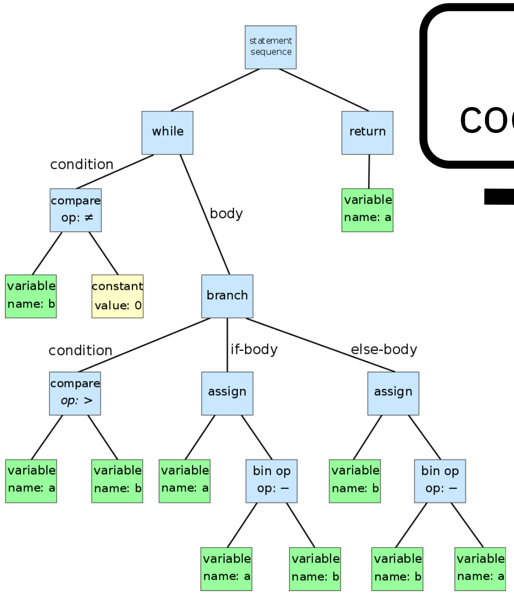
```
import printf;

void main() {
…
```

**Decaf source file**

**Phase 1.** Does it have the right structure? (syntax)

**Phase 2.** Does it make sense? (semantics)



**Internal representation**

**Phase 4.** What can we learn about the program? (dataflow analysis)

**Phase 3**
code generation

```
push %rbp
mov  %rsp, %rbp
…
```

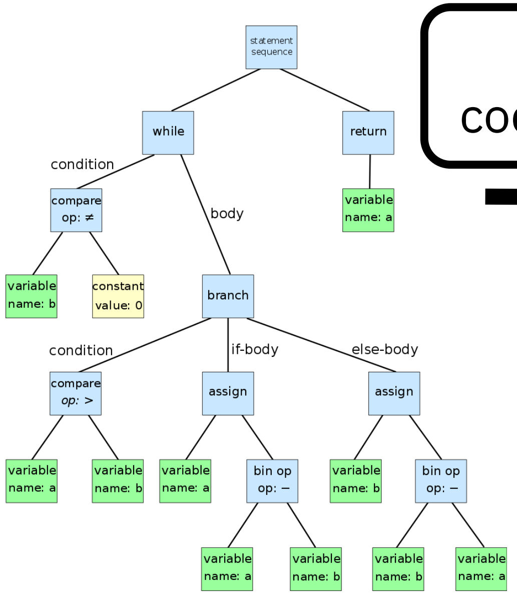
**x86-64 assembly**

# Project overview

```
import printf;

void main() {
…
```

**Decaf source file**



**Internal representation**

**Phase 3**
code generation

```
push %rbp
mov  %rsp, %rbp
…
```

**x86-64 assembly**

**Phase 5.** How can we make the output code faster?

**Phase 1.** Does it have the right structure? (syntax)

**Phase 2.** Does it make sense? (semantics)

**Phase 4.** What can we learn about the program? (dataflow analysis)

# Things we specify for you:

- Input language (Decaf)
- Output language (x86-64 assembly)
- General design (scanning → parsing → semantic checking → code generation)
- Command line interface

# Features of Decaf

- Imperative language, watered down version of C — name stands for Decaffeinated C.
- Follows C semantics and calling convention.
- Types: **`int`**, **`bool`**.
- Operations (arithmetic / boolean / comparison)
- Constant-sized arrays
- Functions

# Example Decaf program

```
import printf;
int array[100];
void main ( ) {
  int i, sum = 0;
  for ( i = 0; i < len(array); i++ ) {
     sum += i;
  }
  printf ( "%d\n", sum );
}
```

# Command line interface

- `./build.sh` builds your compiler
- `./run.sh filename [options]` runs your compiler, must support the following options:

| | |
|---|---|
| `-t \| --target <stage>` | Specify compilation stage: **`scan`**, **`parse`**, **`inter`**, or **`assembly`** |
| `-o \| --output <outname>` | Write output to the specified file name. (If blank, output to stdout) |
| `-O \| --opt [optimizations,…]` | Perform the listed optimizations. **`all`** means all optimizations, **`-optname`** removes optname. |
| `-d \| --debug` | Prints debug information |

Announcements

Weekly updates

Project overview

**Phase 1 details ←**

# Phase 1 overview

- **Goal:** have a working program that can determine whether each input Decaf code is *syntactically* valid or not.
  - We split this into two subtasks: **scanning** and **parsing.**
  - What this phase *doesn't* cover: semantics. Things like type checking, bounds checking, etc. will be done in the next phase.

# Scanner

- **Input:** Decaf code, essentially a string
- **Output:** A list of tokens
- Example:

```
print("Hello, World!");
```

→

- `print`
- `(`
- `"Hello, World!"`
- `)`
- `;`

# Scanner specifications

- When running **`./run.sh <filename> -t scan`** on **a lexically valid input file:**
    - Exit with return code 0 (OK)
    - Outputs tokens, one per line.
    - For identifiers and literals, also output the token type:

```
IDENTIFIER print
(
STRINGLITERAL "Hello, World!"
)
;
```

# Scanner specifications

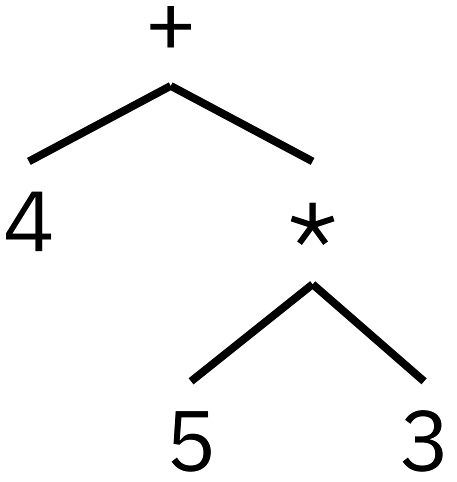
- When running `./run.sh <filename> -t scan` on **a lexically invalid input file:**
  - Exit with a nonzero return code (i.e. error)
- The autograder doesn't check the output, but it's nice to output an error message.

# Parser

- **Input:** A list of tokens
- **Output:** A *parse tree*, which is a data structure that encapsulates the syntactic structure of the program
- Example:

```
INTLITERAL 4
+
INTLITERAL 5
*
INTLITERAL 3
```

➔



# Parser specifications

- When running `./run.sh <filename> -t scan` on **a syntactically valid input file:**
  - Exits with return code 0 (OK)
  - Produce no output
- You can decide how you want to implement your parse trees

# Parser specifications

- When running `./run.sh <filename> -t scan` on **a syntactically invalid input file:**
  - Exit with nonzero return code (i.e. error)
- Again, the autograder doesn’t check the output, but it’s nice to output an error message.

# Submission and grading

- Phase 1 is worth **5%** of the overall grade, due **Friday, February 23.**
- Three items to be submitted on Gradescope
  - Code submission (autograded)
    - Scanner tests: **2%**
    - Parser tests: **2%**
  - Short report (1-2 paragraphs): **1%**
  - LLM questionnaire: **0%** (due 3 days after deadline)

# Getting started

- You should have received an invite to join the course organization (`6110-sp24`).
- We created a repo `<your-kerb>_phase1` for you.
  - If you don’t have access to it, let us know ASAP.
- Make sure to accept the invite for both the organization and the repo!

# Getting started

- We have starting skeletons for Java, Scala, Rust, and Typescript.
  - The skeletons come with a build system and a barebones implementation of the CLI.
  - To use the skeletons, follow the instructions on the Project Skeletons page on the course website.
- You’re also welcome to start from scratch if you’d like to use a different build system or language (but let us know so we can support it on the autograder!)

# Testing

- **Unit tests:** the skeletons come with unit-testing frameworks. (ex. Mocha for Typescript)
  - It's good practice to write your own unit tests for each function/module you're writing. The scanner/parser can get pretty complex, and the test cases we provide are only end-to-end.
- **End-to-end tests:** we provide public test cases in the `public-tests` repository.
  - You should write your own script to run these tests

# Testing

- You can also submit your code on Gradescope to see feedback on the private tests (you'll see the test names and whether you passed or failed them).
  - We suggest doing this if you edit `./build.sh` or `./run.sh` to verify that the autograder can successfully build your code.
  - There is no rate limit, but **try not to overuse this.**
  - Try to use this only for verification purposes, and don't submit every single commit, for example.
  - Don't blindly try to increase your # of private tests passed.

# Words of advice

- **Start early!**
  - The project deadlines in this class are spaced out, so it’s easy to feel like you have a lot of time … until you don’t.
- **You’ll face a lot of design decisions.**
  - One specific example: do you want to use the same token datatypes for both the scanner output and the parse tree?
  - A lot of of the time, it’s usually okay either way. But if you made a choice and got really stuck, maybe step back and reconsider design choices.

# Words of advice

- **Start with a subset of the Decaf grammar.**
  - Dealing with the whole grammar at once can be intimidating. Try picking a self-contained subset of it (ex. arithmetic expressions only, or pure expressions only)
- **Keep source location information.**
  - While we don’t require this in Phase 1, this will be required in the next phase, and it’ll also make debugging a lot easier.

# Words of advice

- **Consider using existing libraries to help.**
  - Regex libraries are allowed and very helpful for scanning.
  - If you’re interested, also check out scanner/parser generators. Our general advice is use these if you already knew the language well, it might be a good learning experience to use them.

# Words of advice

- **The course staff is here to help!**
  - Come to office hours or ask on Piazza!
  - We know that this project can feel pretty intimidating.
  - We can give you suggestions on how to start, and we will try to help you debug issues with your parser and scanner.
  - (Note that we give you a lot of freedom on how to approach the project, and so we might not be able to give very specific guidance in some cases.)