CS 485: Compilers

Martin Kellogg

Agenda

- Administrivia
  - course policies, webpage, Martin, TAs, etc.
- What is this class about?
  - brief history lesson
  - compiler structure
- Discussion of course difficulty + workload (“is this a hard class?”)
  - and why you should or shouldn’t take the course

Agenda

- **Administrivia**
  - course policies, webpage, Martin, TAs, etc.
- What is this class about?
  - brief history lesson
  - compiler structure
- Discussion of course difficulty + workload (“is this a hard class?”)
  - and why you should or shouldn’t take the course

Course policies

Course policies

- **Most important**: the first time each class you ask or answer a question, I throw candy at you (sorry for poor aim)

Course policies

- **Most important**: the first time each class you ask or answer a question, I throw candy at you (sorry for poor aim)
  - Let's try it now! **Suggested questions**:
    - Why would you do that?
    - Are you just bribing us to pay attention?
    - Does that actually work?
    - Do even silly questions count?

Course staff: TAs

- We have two TAs this semester

Course staff: TAs

- We have two TAs this semester
  - Tomasz (Tom) Brauntsch

Course staff: TAs

- We have two TAs this semester
  - Tomasz (Tom) Brauntsch
  - Hamdi Korreshi

Course staff: TAs

- We have two TAs this semester
  - Tomasz (Tom) Brauntsch
  - Hamdi Korreshi
- Tom and Hamdi **did the assignments for this class last semester** as a CS 488
  - they can help with technical questions about your projects

Course staff: TAs

- We have two TAs this semester
  - Tomasz (Tom) Brauntsch
  - Hamdi Korreshi
- Tom and Hamdi **did the assignments for this class last semester** as a CS 488
  - they can help with technical questions about your projects
- Their OH (all in GITC 4324)
  - Tom: M 1-2, F 1-2
  - Hamdi: M 12-1, R 12-1

Course policies: webpage

- You can find most course policies (and assignments, and lecture slides, etc.) on the **course webpage**
  - https://kelloggm.github.io/martinjkellogg.com/teaching/cs485-sp25/

Course policies: webpage

- You can find most course policies (and assignments, and lecture slides, etc.) on the **course webpage**
  - https://kelloggm.github.io/martinjkellogg.com/teaching/cs485-sp25/
  - don't try to memorize that! instead:
    - search my name + NJIT: "Martin Kellogg NJIT"
    - scroll down to "Teaching"
    - follow the link there

Course policies: webpage

- You can find most course policies (and assignments, and lecture slides, etc.) on the **course webpage**
  - https://kelloggm.github.io/martinjkellogg.com/teaching/cs485-sp25/
  - don't try to memorize that! instead:
    - search my name + NJIT: "Martin Kellogg NJIT"
    - scroll down to "Teaching"
    - follow the link there
- Most important things to read on the webpage:
  - syllabus, calendar, assignment descriptions

Course policies: webpage

- You can find most course policies (and assignments, and lecture slides, etc.) on the **course webpage**
  - https://kelloggm.github.io/martinjkellogg.com/teaching/cs485-sp25/
  - don't try to memorize that! instead:
    - search my name + NJIT: "Martin Kellogg NJIT"
    - scroll down to "Teaching"
    - follow the link there
- Most important things to read on the w
  - syllabus, calendar, assignment descri

We will also make heavy use of the course forum (**Discord** this semester)

Course policies: slides

- The website's calendar page will have **links to the slides** "when they're done"

Course policies: slides

- The website's calendar page will have **links to the slides** "when they're done"
  - which in practice will usually mean "right before class starts"
    - sorry, professors procrastinate too :)

Course policies: slides

- The website's calendar page will have **links to the slides** "when they're done"
  - which in practice will usually mean "right before class starts"
    - sorry, professors procrastinate too :)
- I post the slides to help you study for the exams, but they're not a substitute for **taking notes yourself**

Course policies: slides

- The website's calendar page will have **links to the slides** "when they're done"
  - which in practice will usually mean "right before class starts"
    - sorry, professors procrastinate too :)
- I post the slides to help you study for the exams, but they're not a substitute for **taking notes yourself**
  - when I **make claims** like this one, you're encouraged to **ask me to justify** why I'm making the claim

Aside: on taking notes

“…students who took notes on laptops performed worse on conceptual questions than students who took notes longhand. We show that whereas taking more notes can be beneficial, laptop note takers’ tendency to transcribe lectures verbatim rather than processing information and reframing it in their own words is detrimental to learning.”

[Pam Mueller, Daniel Oppenheimer. The pen is mightier than the keyboard: advantages of longhand over laptop note taking. Psychol. Sci. 2014 Jun; 25(6):Epub 2014 Apr 23.]

Aside: red-bordered slides

- Did you notice the **red border** on the previous slide (and this one)?

Aside: red-bordered slides

- Did you notice the **red border** on the previous slide (and this one)?
- The border indicates material that is **not** fair game for exam questions
  - I often include asides or tangents into lecture material, like the one on taking notes on the previous slides
  - You'll also see this border "mid-class break" slides later :)

Course staff: me

- NJIT assistant professor since 2022
  - I usually teach CS 490 at the undergrad level

Course staff: me

- NJIT assistant professor since 2022
  - I usually teach CS 490 at the undergrad level
- Previously:
  - PhD at University of Washington (Seattle) until June 2022
  - BS at University of Virginia (Charlottesville) in 2016
- My office hours: W 3:30-4:30, GITC 4314

Course staff: me

- NJIT assistant professor since 2022
  - I usually teach CS 490 at the undergrad level
- Previously:
  - PhD at University of Washington (Seattle) until June 2022
  - BS at University of Virginia (Charlottesville) in 2016
- My office hours: W 3:30-4:30, GITC 4314

This class is heavily inspired (with permission) from a course I took as an undergrad! So, I too have done the assignments (many moons ago).

Course staff: me: style

- Qualities that I strive for as an instructor:
  - difficulty level is very high
  - students have to put in a lot of work to succeed
  - students learn a lot
  - students find the course worthwhile

Course staff: me: style

- Qualities that I strive for as an instructor:
  - **difficulty level is very high**
  - **students have to put in a lot of work to succeed**
  - students learn a lot
  - students find the course worthwhile

Course staff: me: the bad

- (all quotes are about CS 490)

"the midterm was **very difficult** and the project was **difficult** as well"

"I think there is a bit **too much content** … this class takes **way more time** than most other classes."

"very straight forward but **very difficult exam**"

"He's **devilish with his intense assignments**, but has the dignity to be upfront about it."

Course staff: me: style

- Qualities that I strive for as an instructor:
  - difficulty level is very high
  - students have to put in a lot of work to succeed
  - **students learn a lot**
  - **students find the course worthwhile**

Course staff: me: the good

- (all quotes are about CS 490)

"I like his teaching style quite a lot and find his **lectures to be quite interesting**."

"how the professor **engages the class to participate** is great"

"**ability to explain complex concepts** in an easy to understand way is a plus….in some cases he does **move a little to fast**."

"capable of **actually explaining** complex and abstract concepts in a non-judgemental way to people unfamiliar to them"

Course staff: me: style

- Quotes highlighted the qualities that I strive for as an instructor:
  - difficulty level is very high
  - students have to put in a lot of work to succeed
  - students learn a lot
  - students find the course worthwhile
- This course, unlike CS 490, is an **elective**

Course staff: me: style

- Quotes highlighted the qualities that I strive for as an instructor:
  - difficulty level is very high
  - students have to put in a lot of work to succeed
  - students learn a lot
  - students find the course worthwhile
- This course, unlike CS 490, is an **elective**
  - that means that these qualities will be **even more extreme**
    - e.g., assignments will be **harder** (!)

Course staff: me: style

- Quotes highlighted the qualities that I strive for as an instructor:
  - difficulty level is very high
  - students have to put in a lot of work to succeed
  - students learn a lot
  - students find the course worthwhile
- This course, unlike CS 490, is an **elective**
  - that means that these qualities will be **even more extreme**
    - e.g., assignments will be **harder** (!)
  - but I hope you'll get even more out of the course!

Course staff: me: why Compilers?

- Foundational for my research work
  - which is about type system design
  - maybe some of you will want to do research with me?

Course staff: me: why Compilers?

- Foundational for my research work
  - which is about type system design
  - maybe some of you will want to do research with me?
- Compilers had a big impact on my career
  - before taking this course as an undergrad, I planned to be a software engineer
  - so I'm excited to try my hand at introducing it to you :)

Course staff: me: why Compilers?

- Foundational for my research work
  - which is about type system design
  - maybe some of you will want to do research with me?
- Compilers had a big impact on my career
  - before taking this course as an undergrad, I planned to be a software engineer
  - so I'm excited to try my hand at introducing it to you :)
    - that said, this is the first time I'm teaching this course. So, please be **patient** when (not if) I make some mistakes…

Agenda

- Administrivia
  - course policies, webpage, Martin, TAs, etc.
- **What is this class about?**
  - brief history lesson
  - compiler structure
- Discussion of course difficulty + workload (“is this a hard class?”)
  - and why you should or shouldn’t take the course

What do I mean by “Compilers”?

What do I mean by “Compilers”?

Definition: a ***compiler*** is a program that translates another program written in one language (the ***source language***) into a ***semantically-equivalent*** program in a different language (the ***target language***)

What do I mean by “Compilers”?

Definition: a ***compiler*** is a program that translates another program written in one language (the ***source language***) into a ***semantically-equivalent*** program in a different language (the ***target language***)

- “***semantically-equivalent***” = “has the same meaning”

What do I mean by “Compilers”?

Definition: a ***compiler*** is a program that translates another program written in one language (the ***source language***) into a ***semantically-equivalent*** program in a different language (the ***target language***)

- “***semantically-equivalent***” = “has the same meaning”
- typically, the source language is “**higher level**” than the target

What do I mean by “Compilers”?

Definition: a ***compiler*** is a program that translates another program written in one language (the ***source language***) into a ***semantically-equivalent*** program in a different language (the ***target language***)

- “***semantically-equivalent***” = “has the same meaning”
- typically, the source language is “**higher level**” than the target
  - e.g., source language is C, target is assembly

What do I mean by “Compilers”?

Definition: a ***compiler*** is a program that translates another program written in one language (the ***source language***) into a ***semantically-equivalent*** program in a different language (the ***target language***)

- “***semantically-equivalent***” = “has the same meaning”
- typically, the source language is “**higher level**” than the target
  - e.g., source language is C, target is assembly
  - “higher level” here refers to a higher level **of abstraction**

What do I mean by “Compilers”?

Definition: a ***compiler*** is a program that translates another program written in one language (the ***source language***) into a ***semantically-equivalent*** program in another language (the ***target language***)

Note that “higher-level” here is in the eye of the beholder. Sometimes, a compiler whose source and target languages are similarly abstract is called a ***transpiler*** instead.

- “***semantically-equivalent***”
- typically, the source language is “**higher level**” than the target
  - e.g., source language is C, target is assembly
  - “higher level” here refers to a higher level **of abstraction**

What do I mean by “Compilers”?

- Compilers are a **fundamental technology** for modern software engineering. Key benefits include:

What do I mean by “Compilers”?

- Compilers are a **fundamental technology** for modern software engineering. Key benefits include:
  - enable programmers to **write code more quickly**

History: writing code more quickly

- 1953: IBM develops the 701 “Defense Calculator”

History: writing code more quickly

- 1953: IBM develops the 701 “Defense Calculator”
  - cf. 1952: US formally ends occupation of Japan
  - 1954: Brown v. Board of Education of Topeka, Kansas

History: writing code more quickly

- 1953: IBM develops the 701 “Defense Calculator”
  - cf. 1952: US formally ends occupation of Japan
  - 1954: Brown v. Board of Education of Topeka, Kansas
- All programming done in **assembly**

History: writing code more quickly

- 1953: IBM develops the 701 “Defense Calculator”
  - cf. 1952: US formally ends occupation of Japan
  - 1954: Brown v. Board of Education of Topeka, Kansas
- All programming done in **assembly**
- **Problem**: Software costs exceeded hardware costs!

History: writing code more quickly

- 1953: IBM develops the 701 “Defense Calculator”
  - cf. 1952: US formally ends occupation of Japan
  - 1954: Brown v. Board of Education of Topeka, Kansas
- All programming done in **assembly**
- **Problem**: Software costs exceeded hardware costs!
- John Backus: “Speedcoding”

History: writing code more quickly

- 1953: IBM develops the 701 “Defense Calculator”
  - cf. 1952: US formally ends occupation of Japan
  - 1954: Brown v. Board of Education of Topeka, Kansas
- All programming done in **assembly**
- **Problem**: Software costs exceeded hardware costs!
- John Backus: “Speedcoding”
  - An ***interpreter*** (more on this in a few minutes)
    - Ran 10-20x slower than hand-written assembly

History: writing code more quickly

- 1953: IBM develops the 701 “Defense Calculator”
  - cf. 1952: US formally ends occupation of Japan
  - 1954: Brown v. Board of Education of Topeka, Kansas
- All programming done in **assembly**
- **Problem**: Software costs exceeded hardware costs!
- John Backus: “Speedcoding”
  - An ***interpreter*** (more on this in a few minutes)
    - Ran 10-20x slower than hand-written assembly
  - but cut development time dramatically (2 weeks -> 2 hours)

What do I mean by “Compilers”?

- Compilers are a **fundamental technology** for modern software engineering. Key benefits include:
  - enable programmers to **write code more quickly**
    - **downside**: program is slower

History: mitigating the downside

- 1954: IBM develops the 704

History: mitigating the downside

- 1954: IBM develops the 704
- John Backus's next idea, based on the success of the Speedcoding project: **translate** high-level code to assembly

History: mitigating the downside

- 1954: IBM develops the 704
- John Backus's next idea, based on the success of the Speedcoding project: **translate** high-level code to assembly
  - Many thought this impossible

History: mitigating the downside

- 1954: IBM develops the 704
- John Backus's next idea, based on the success of the Speedcoding project: **translate** high-level code to assembly
  - Many thought this impossible
- 1954-7 FORTRAN I project

History: mitigating the downside

- 1954: IBM develops the 704
- John Backus's next idea, based on the success of the Speedcoding project: **translate** high-level code to assembly
  - Many thought this impossible
- 1954-7 FORTRAN I project
- By 1958, >50% of all software is in FORTRAN

History: mitigating the downside

- 1954: IBM develops the 704
- John Backus's next idea, based on the success of the Speedcoding project: **translate** high-level code to assembly
  - Many thought this impossible
- 1954-7 FORTRAN I project
- By 1958, >50% of all software is in FORTRAN
  - Compilers can **save as much programmer time** as an interpreter, but produce **much faster code**

What do I mean by “Compilers”?

- Compilers are a **fundamental technology** for modern software engineering. Key benefits include:
  - enable programmers to **write code more quickly**
    - **downside**: program is slower
  - enable code to be **shared between different machines**

History: sharing code between machines

- Early assembly programs were **machine-dependent**

History: sharing code between machines

- Early assembly programs were **machine-dependent**
- Compilers enabled **code reuse** between machines:

History: sharing code between machines

- Early assembly programs were **machine-dependent**
- Compilers enabled **code reuse** between machines:
  - write your programs in a high-level language like FORTRAN

History: sharing code between machines

- Early assembly programs were **machine-dependent**
- Compilers enabled **code reuse** between machines:
  - write your programs in a high-level language like FORTRAN
  - when you buy a new computer, instead of having to rewrite many programs in the new assembly language, you only need to **write a new FORTRAN compiler** (1 program!)

History: sharing code between machines

- Early assembly programs were **machine-dependent**
- Compilers enabled **code reuse** between machines:
  - write your programs in a high-level language like FORTRAN
  - when you buy a new computer, instead of having to rewrite many programs in the new assembly language, you only need to **write a new FORTRAN compiler** (1 program!)
  - as languages became standardized, vendors started distributing compilers (e.g., for FORTRAN or C) with the machine

History: sharing code between machines

- Early assembly programs were **machine-dependent**
- Compilers enabled **code**
  - write your programs
  - when you buy a new
    many programs in th
    to **write a new FORTRAN compiler** (1 program!)
  - as languages became standardized, vendors started distributing compilers (e.g., for FORTRAN or C) with the machine

Implication for this class: writing a compiler itself is still **relatively machine-dependent**, even in the modern era. You'll need to use the same OS version etc. as our grading server!

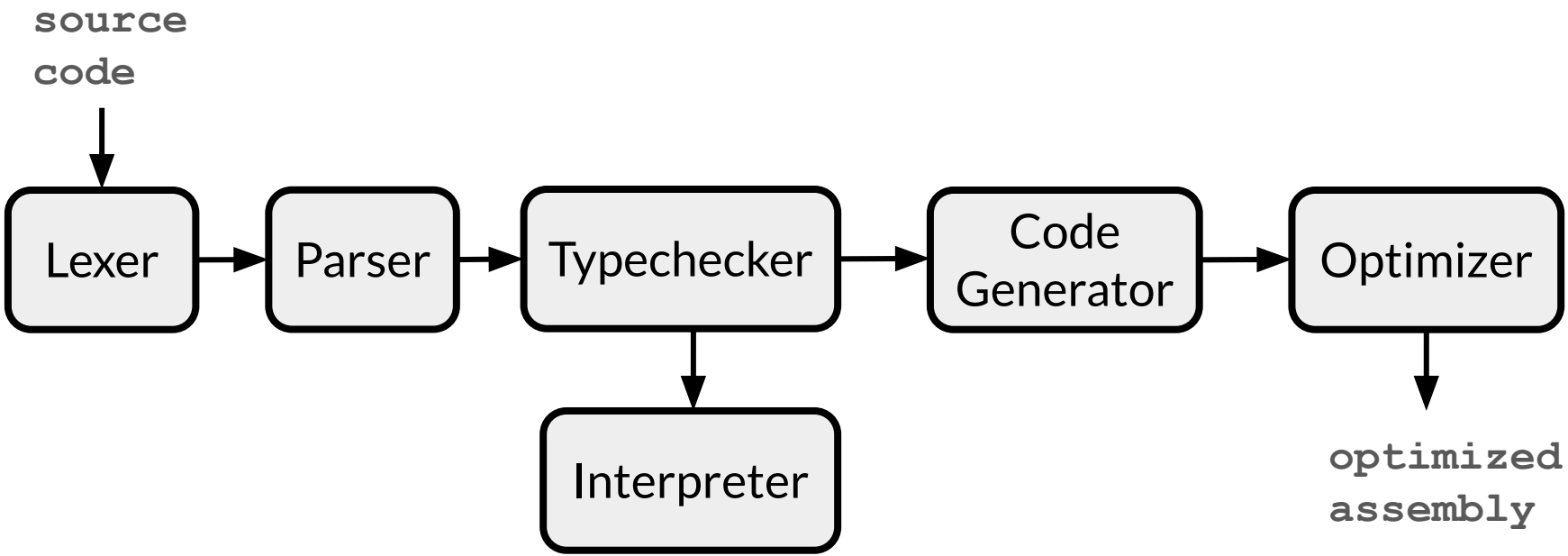
Trivia Break: Computer Science

This notation is used to describe the syntax of both programming languages and other formal languages. It can be described as a metasyntax for context-free grammars. It is typically used whenever an exact language description is needed, such as in official language specifications, in manuals, or in textbooks on programming language theory. It is named after its co-creators, who are also famous for leading the development of the early programming languages FORTRAN and Algol, respectively.

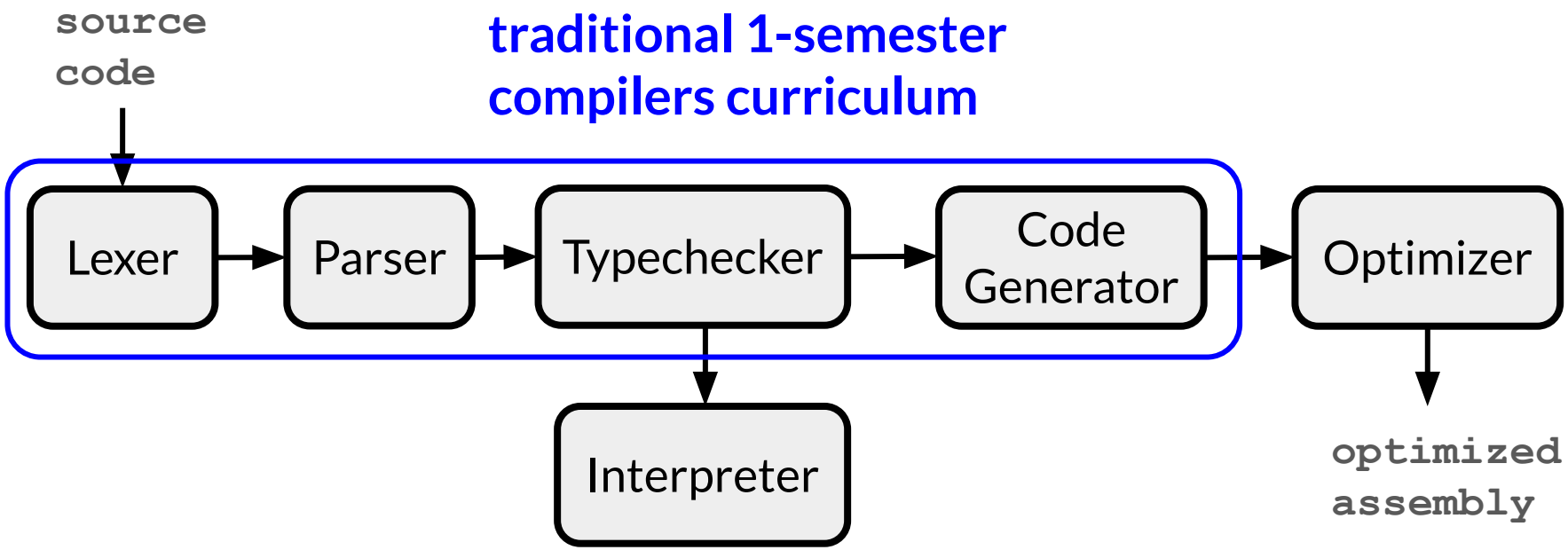
Trivia Break: Real-life Languages

46% (3.2 billion people) of the world's population speaks a language from this language family as their native language - by far the highest of any language family. It is divided into 17 branches, of which 9 are extinct and 8 contain one or more living languages: Albanian, Armenian, Balto-Slavic, Celtic, Germanic, Hellenic, Indo-Iranian, and Italic. The languages from this family with the most native speakers today are English, Spanish, Portuguese, Russian, Hindustani, Bengali, French, and German.

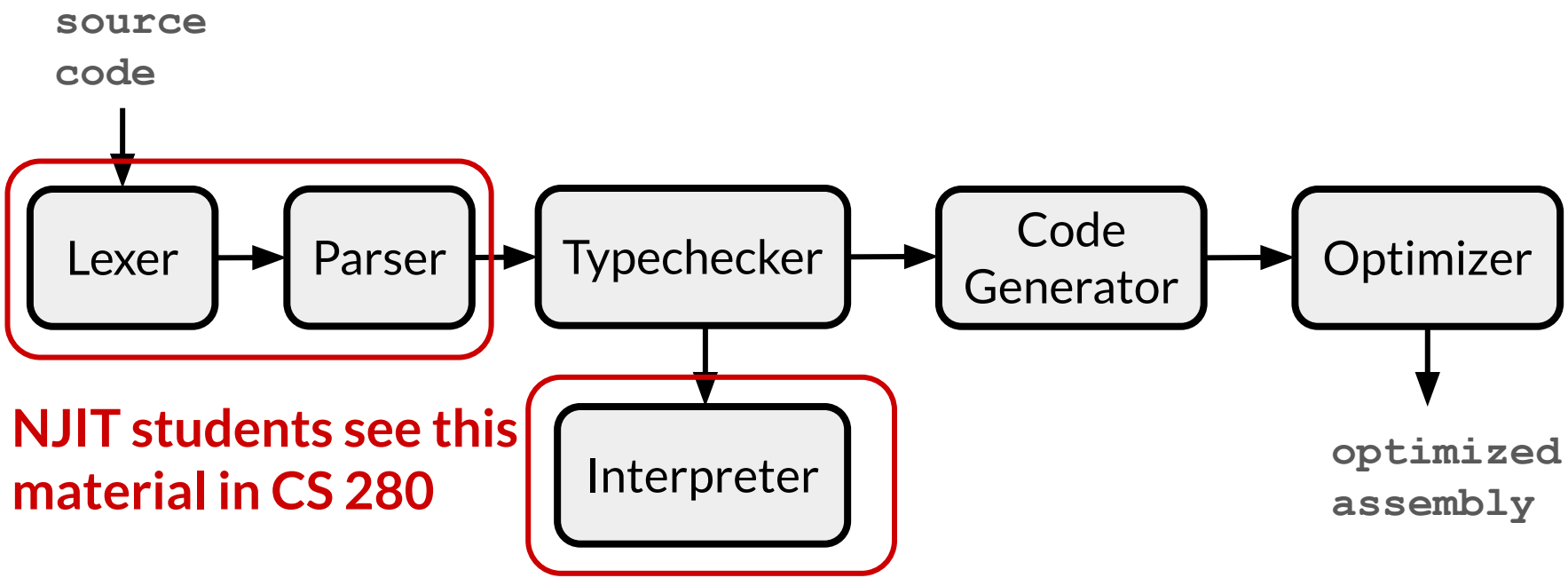
Traditional compiler/interpreter structure



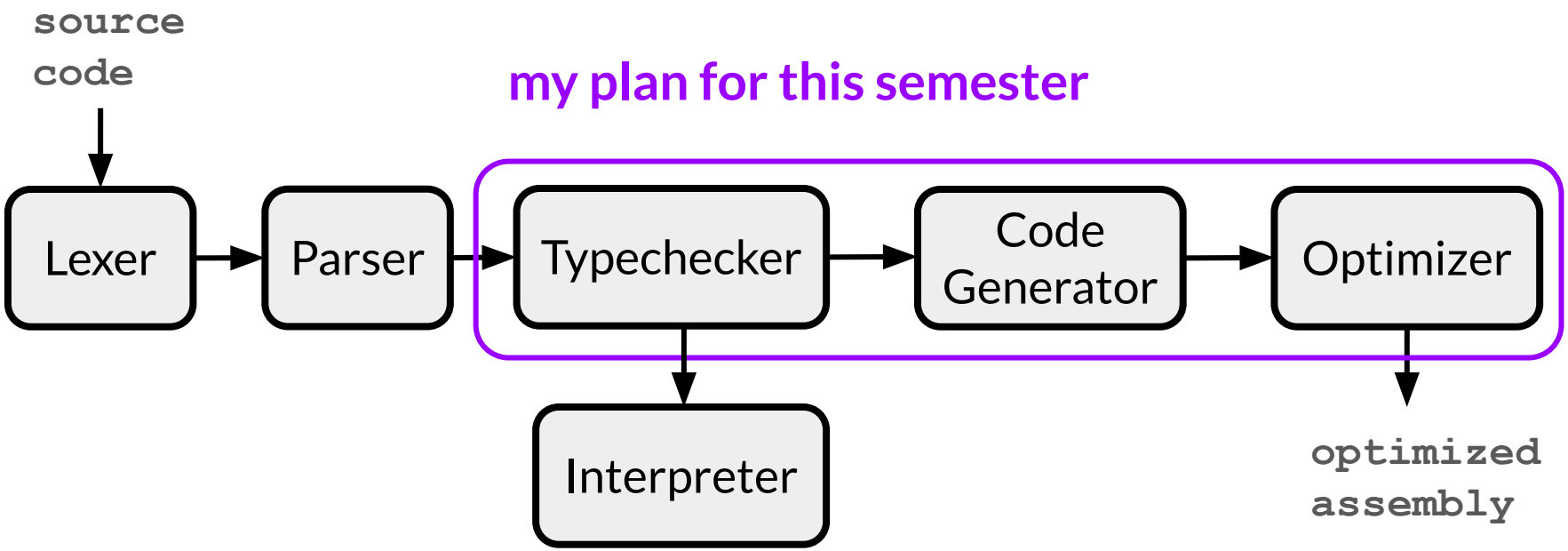
Traditional compiler/interpreter structure



Traditional compiler/interpreter structure



Traditional compiler/interpreter structure



Interpreters vs Compilers

Definition: an ***interpreter*** is a program that executes its input program.

Interpreters vs Compilers

Definition: an ***interpreter*** is a program that executes its input program.

- interpreters and compilers are closely related

Interpreters vs Compilers

Definition: an ***interpreter*** is a program that executes its input program.

- interpreters and compilers are closely related
  - share same **frontend**: lexing and parsing

Interpreters vs Compilers

Definition: an ***interpreter*** is a program that executes its input program.

- interpreters and compilers are closely related
  - share same **frontend**: lexing and parsing
  - but differ on the **backend**:

Interpreters vs Compilers

Definition: an ***interpreter*** is a program that executes its input program.

- interpreters and compilers are closely related
  - share same **frontend**: lexing and parsing
  - but differ on the **backend**:
    - interpreters execute the parsed code
    - compilers instead generate code that itself will execute the parsed code

Interpreters vs Compilers

Definition: an ***interpreter*** is a program that executes its input program.

- interpreters and compilers are closely related
  - share same **frontend**: lexing and parsing
  - but differ on the **backend**:
    - interpreters execute the parsed code
    - compilers instead generate code that itself will execute the parsed code
- Not all modern languages are compiled (e.g., Python!)

Interpreters vs Compilers

Interpreters:

Compilers:

Interpreters vs Compilers

Interpreters:

- Lexical analysis

Compilers:

- Lexical analysis

Interpreters vs Compilers

Interpreters:

- Lexical analysis
- Parsing

Compilers:

- Lexical analysis
- Parsing

Interpreters vs Compilers

Interpreters:

- Lexical analysis
- Parsing
- Semantic analysis

Compilers:

- Lexical analysis
- Parsing
- Semantic analysis

Interpreters vs Compilers

Interpreters:

- Lexical analysis
- Parsing
- Semantic analysis
- (optionally) Optimization

Compilers:

- Lexical analysis
- Parsing
- Semantic analysis
- (optionally) Optimization

Interpreters vs Compilers

Interpreters:

- Lexical analysis
- Parsing
- Semantic analysis
- (optionally) Optimization
- Run the program

Compilers:

- Lexical analysis
- Parsing
- Semantic analysis
- (optionally) Optimization
- Generate machine code

Interpreters vs Compilers

Interpreters:

- Lexical analysis
- Parsing
- Semantic analysis
- (optionally) Optimization
- Run the program

Compilers:

- Lexical analysis
- Parsing
- Semantic analysis
- (optionally) Optimization
- Generate machine code

The first 3, at least, can be understood by analogy to how humans comprehend natural languages like English.

Lexical analysis

- First step: **recognize words**

Lexical analysis

- First step: **recognize words**
  - Smallest unit above letters

Lexical analysis

- First step: **recognize words**
  - Smallest unit above letters
- Consider this example:

This is a sentence.

Lexical analysis

- First step: **recognize words**
  - Smallest unit above letters
- Consider this example:

This is a sentence.

- Note the:

Lexical analysis

- First step: **recognize words**
  - Smallest unit above letters
- Consider this example:

This is a sentence.

- Note the:
  - Capital letter "**T**" (symbol for **start** of a sentence)

Lexical analysis

- First step: **recognize words**
  - Smallest unit above letters
- Consider this example:

This is a sentence.

- Note the:
  - Capital letter "**T**" (symbol for **start** of a sentence)
  - Spaces between words (symbol for **word separator**)

Lexical analysis

- First step: **recognize words**
  - Smallest unit above letters
- Consider this example:

This is a sentence.

- Note the:
  - Capital letter "**T**" (symbol for **start** of a sentence)
  - Spaces between words (symbol for **word separator**)
  - Period at the end (symbol for **end** of sentence)

Lexical analysis

- Lexical analysis is not trivial. Consider:

  How d'you break "this" up?

Lexical analysis

- Lexical analysis is not trivial. Consider:

  How d'you break "this" up?

- Plus, programming languages are typically ***more cryptic*** than English:

  ***p->f += -.12345e-6**

Lexical analysis

- A ***lexical analyzer*** (or ***lexer***) divides program text into "words" or ***tokens***

Lexical analysis

- A ***lexical analyzer*** (or ***lexer***) divides program text into "words" or ***tokens***
  - "Token" is just a technical term for the "words" in a programming language

Lexical analysis

- A ***lexical analyzer*** (or ***lexer***) divides program text into "words" or ***tokens***
  - "Token" is just a technical term for the "words" in a programming language
- For example, consider:

  if x == y then z = 1; else z = 2;

Lexical analysis

- A ***lexical analyzer*** (or ***lexer***) divides program text into "words" or ***tokens***
  - "Token" is just a technical term for the "words" in a programming language
- For example, consider:

  if x == y then z = 1; else z = 2;

- A lexer would break this up into:

  if, x, ==, y, then, z, =, 1, ;, else, z, =, 2, ;

Parsing

- Once words are understood, the next step is to understand **sentence structure**

Parsing

- Once words are understood, the next step is to understand **sentence structure**
- ***Parsing*** is like **diagramming sentences**

Parsing

- Once words are understood, the next step is to understand **sentence structure**
- ***Parsing*** is like **diagramming sentences**
  - the diagram is a tree
  - often annotated with additional information

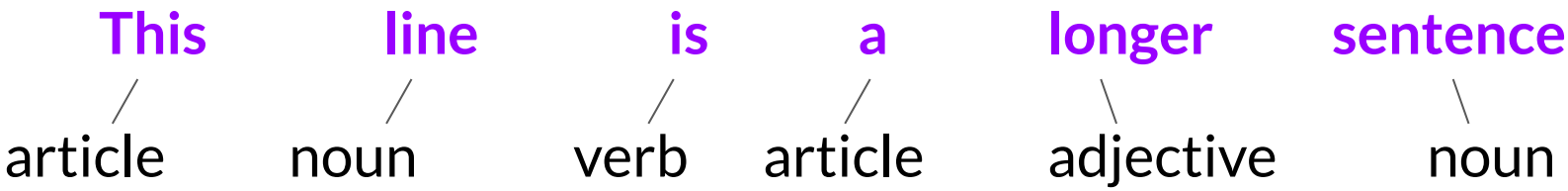
Sentence Diagramming

This line is a longer sentence

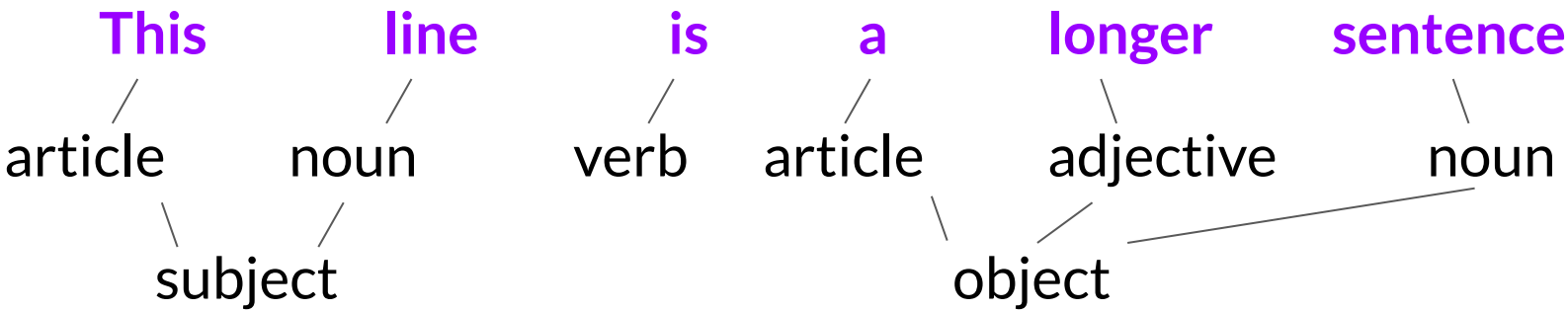
Sentence Diagramming

Sentence Diagramming

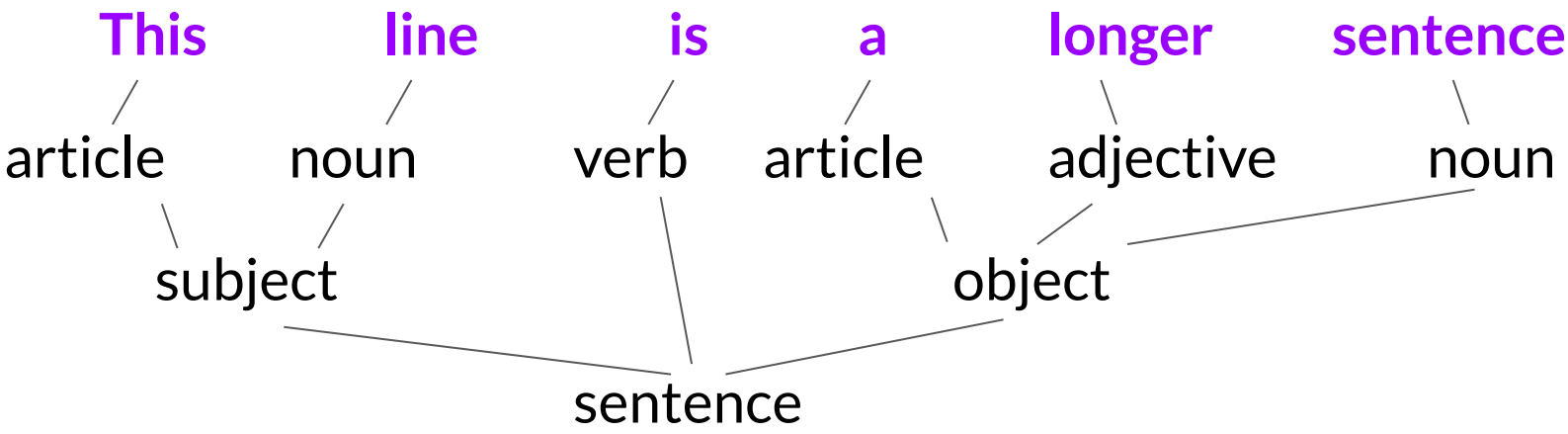
Sentence Diagramming



Sentence Diagramming



Sentence Diagramming



Parsing Programs

- Parsing program expressions is the same

Parsing Programs

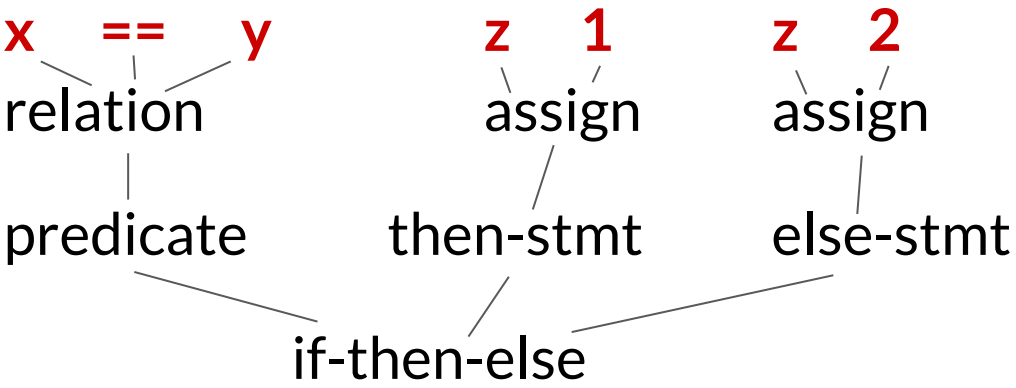
- Parsing program expressions is the same
- Consider:

if x == y then z = 1; else z = 2;

Parsing Programs

- Parsing program expressions is the same
- Consider:

  if x == y then z = 1; else z = 2;
- Diagrammed:



Semantic Analysis

- Once sentence structure is understood, we can try to understand “meaning”

Semantic Analysis

- Once sentence structure is understood, we can try to understand “meaning”
  - But meaning is **too hard for compilers**

Semantic Analysis

- Once sentence structure is understood, we can try to understand “meaning”
  - But meaning is **too hard for compilers**
- Compilers perform limited analysis to **catch inconsistencies**

Semantic Analysis

- Once sentence structure is understood, we can try to understand “meaning”
  - But meaning is **too hard for compilers**
- Compilers perform limited analysis to **catch inconsistencies**
  - Goal: **reject bad programs early**!

Semantic Analysis

- Once sentence structure is understood, we can try to understand “meaning”
  - But meaning is **too hard for compilers**
- Compilers perform limited analysis to **catch inconsistencies**
  - Goal: **reject bad programs early**!
- Some do more analysis to **improve the performance** of the program

Semantic Analysis

- Once sentence structure is understood, we can try to understand “meaning”
  - But meaning is **too hard for compilers**
- Compilers perform limited analysis to **catch inconsistencies**
  - Goal: **reject bad programs early**!
- Some do more analysis to **improve the performance** of the program
  - This is ***optimization***

Semantic Analysis in English

- Example:

  Tom said that Hamdi had tested his code.

Semantic Analysis in English

- Example:

  Tom said that Hamdi had tested his code.

- What does “his” refer to? Tom? Or Hamdi?

Semantic Analysis in English

- Example:

  Tom said that Hamdi had tested his code.

- What does “his” refer to? Tom? Or Hamdi?
- It can get even worse:

  Tom said that Tom had tested his code.

Semantic Analysis in English

- Example:

  Tom said that Hamdi had tested his code.

- What does “his” refer to? Tom? Or Hamdi?
- It can get even worse:

  Tom said that Tom had tested his code.

- How many Toms are there? Which one tested his code?

Semantic Analysis in English

- Example:

  Tom said that Hamdi had tested his code.

- What does “his” refer to? Tom? Or Hamdi?
- It can get even worse:

  Tom said that Tom had tested his code.

It’s **context-sensitive**!

- How many Toms are there? Which one tested his code?

Semantic Analysis in Programming

- Programming languages define **strict rules** to avoid such ambiguities

Semantic Analysis in Programming

- Programming languages define **strict rules** to avoid such ambiguities
  - e.g., the C++ code on the right prints “4”, because C++ always uses the most-recent definition of a variable

```
{
    int s = 3;
    {
        int s = 4;
        cout << s;
    }
}
```

Semantic Analysis in Programming

- Programming languages define **strict rules** to avoid such ambiguities
  - e.g., the C++ code on the right prints “4”, because C++ always uses the most-recent definition of a variable

```
{
    int s = 3;
    {
        int s = 4;
        cout << s;
    }
}
```

We’ll discuss this kind of ***scoping*** issue in a later lecture.

Semantic Analysis in Programming

- Compilers perform many other kinds of **semantic checks** besides variable bindings

Semantic Analysis in Programming

- Compilers perform many other kinds of **semantic checks** besides variable bindings
- For example, ***type systems*** enforce data separation:

Semantic Analysis in Programming

- Compilers perform many other kinds of **semantic checks** besides variable bindings
- For example, ***type systems*** enforce data separation:
  - ensures that valid operands are present for binary operations (e.g., that in `x + y` both `x` and `y` are `Int`s)

Semantic Analysis in Programming

- Compilers perform many other kinds of **semantic checks** besides variable bindings
- For example, ***type systems*** enforce data separation:
  - ensures that valid operands are present for binary operations (e.g., that in `x + y` both `x` and `y` are `Int`s)
  - ensures that methods being called actually exist on the relevant object
  - etc.

Optimization

- No strong counterpart in English, but akin to editing (cf. poems, short stories)

Optimization

- No strong counterpart in English, but akin to editing (cf. poems, short stories)
- **Automatically modify** programs so that they:

Optimization

- No strong counterpart in English, but akin to editing (cf. poems, short stories)
- **Automatically modify** programs so that they:
  - Run faster

Optimization

- No strong counterpart in English, but akin to editing (cf. poems, short stories)
- **Automatically modify** programs so that they:
  - Run faster
  - Use less memory

Optimization

- No strong counterpart in English, but akin to editing (cf. poems, short stories)
- **Automatically modify** programs so that they:
  - Run faster
  - Use less memory
  - In general, conserve some resource

Issues

- Compiling and interpreting are almost this simple, but there are **many pitfalls**

Issues

- Compiling and interpreting are almost this simple, but there are **many pitfalls**
  - Example: How are **bad programs** handled?

Issues

- Compiling and interpreting are almost this simple, but there are **many pitfalls**
  - Example: How are **bad programs** handled?
- Language design has big impact on compiler

Issues

- Compiling and interpreting are almost this simple, but there are **many pitfalls**
  - Example: How are **bad programs** handled?
- Language design has big impact on compiler
  - Determines what is easy and hard to compile

Issues

- Compiling and interpreting are almost this simple, but there are **many pitfalls**
  - Example: How are **bad programs** handled?
- Language design has big impact on compiler
  - Determines what is easy and hard to compile
  - Course theme: **trade-offs** in language design

Languages today

- The **overall structure** of almost every compiler & interpreter follows our outline

Languages today

- The **overall structure** of almost every compiler & interpreter follows our outline
- The **proportions have changed** since FORTRAN

Languages today

- The **overall structure** of almost every compiler & interpreter follows our outline
- The **proportions have changed** since FORTRAN
  - Early: lexing, parsing most complex, expensive

Languages today

- The **overall structure** of almost every compiler & interpreter follows our outline
- The **proportions have changed** since FORTRAN
  - Early: lexing, parsing most complex, expensive
  - Today: **optimization dominates** all other phases; lexing and parsing are cheap and standardized

Languages today

- The **overall structure** of almost every compiler & interpreter follows our outline
- The **proportions have changed** since FORTRAN
  - Early: lexing, parsing most complex, expensive
  - Today: **optimization dominates** all other phases; lexing and parsing are cheap and standardized
  - Thus: this course avoids ancient parsing techniques (e.g., LL, LALR) that you'd find in e.g., the Dragon book
    - instead, we focus on **semantic analysis and optimization**

Languages today

- The **overall structure** of almost every compiler & interpreter follows our outline
- The **proportions have changed** since FORTRAN
  - Early: lexing, parsing most complex, expensive
  - Today: **optimization dominates** all other phases; lexing and parsing are cheap and standa
  - Thus: this course avoids ancie LALR) that you'd find in e.g., t
    - instead, we focus on **semantic analysis and optimization**

This boils down to studying how to build compilers that generate **correct and fast** programs!

Agenda

- Administrivia
  - course policies, webpage, Martin, TAs, etc.
- What is this class about?
  - brief history lesson
  - compiler structure
- **Discussion of course difficulty + workload (“is this a hard class?”)**
  - and why you should or shouldn’t take the course

Course difficulty

Course difficulty

- Unhappiness is often related to unrealized desires
  - cf. “Must take this to graduate”, “Must have good grades”, “Must have free time”

Course difficulty

- Unhappiness is often related to unrealized desires
  - cf. “Must take this to graduate”, “Must have good grades”, “Must have free time”
- That is, **your expectations matter a lot** when thinking about whether you’ll find something worthwhile

Course difficulty

- Unhappiness is often related to unrealized desires
  - cf. “Must take this to graduate”, “Must have good grades”, “Must have free time”
- That is, **your expectations matter a lot** when thinking about whether you’ll find something worthwhile
  - I want everyone in this room to **understand what they’re getting into** when taking this course

Course difficulty

- Unhappiness is often related to unrealized desires
  - cf. “Must take this to graduate”, “Must have good grades”, “Must have free time”
- That is, **your expectations matter a lot** when thinking about whether you’ll find something worthwhile
  - I want everyone in this room to **understand what they’re getting into** when taking this course
    - I wish NJIT’s drop date was later, so you had time to experience the course fully before you have to decide

Course difficulty

- Unhappiness is often related to unrealized desires
  - cf. “Must take this to graduate”, “Must have good grades”, “Must have free time”
- That is, **your expectations matter a lot** when thinking about whether you’ll find something worthwhile
  - I want everyone in this room to **understand what they’re getting into** when taking this course
    - I wish NJIT’s drop date was later, so you had time to experience the course fully before you have to decide
    - but I don’t control the academic calendar :(

So, is this a hard course?

So, is this a hard course?

Unequivocally, the answer is ***YES!***

So, is this a hard course?

Unequivocally, the answer is ***YES!***

- the material is dense, and most students will find some parts boring

So, is this a hard course?

Unequivocally, the answer is ***YES!***

- the material is dense, and most students will find some parts boring
- the assignments are both difficult and time-consuming

So, is this a hard course?

Unequivocally, the answer is ***YES!***

- the material is dense, and most students will find some parts boring
- the assignments are both difficult and time-consuming
- however, I hope to give high grades to everyone who finishes them

So, is this a hard course?

Unequivocally, the answer is ***YES!***

- the material is dense, and most students will find some parts boring
- the assignments are both difficult and time-consuming
- however, I hope to give high grades to everyone who finishes them
    - so, it'll be a slog but if you put in the effort you'll be rewarded

So, what’s the payoff?

So, what's the payoff?

- Increase capacity of expression
  - See what is possible
- Improve understanding of program behavior
  - Know how things work "under the hood"
- Increase ability to learn new programming languages
- Learn to build a large and reliable system
  - Compiler is an excellent portfolio entry
- See many basic CS concepts at work
- Computers are one of the only tools that can increase cognitive power, so learn to control them

So, what's the payoff?

- Increase capacity of expression
  - See what is possible
- Improve understanding of program behavior
  - Know how things work "under the hood"
- Increase ability to learn new programming languages
- Learn to build a large and reliable system
  - Compiler is an excellent portfolio entry
- See many basic CS concepts at work
- Computers are one of the **only tools that can increase cognitive power**, so learn to control them

The Computer is a Unique Machine

- The computer is one of the few “machines” (e.g., lever, pulley, etc.) in that it magnifies our **mental** rather than **physical** force

The Computer is a Unique Machine

- The computer is one of the few “machines” (e.g., lever, pulley, etc.) in that it magnifies our **mental** rather than **physical** force
  - Computers can assist with decision making, model and predict outcomes, etc.

The Computer is a Unique Machine

- The computer is one of the few “machines” (e.g., lever, pulley, etc.) in that it magnifies our **mental** rather than **physical** force
  - Computers can assist with decision making, model and predict outcomes, etc.
- The computer is (arguably) the **defining technology** of our era
  - cf. “Information Age”

The Computer is a Unique Machine

- The computer is one of the few “machines” (e.g., lever, pulley, etc.) in that it magnifies our **mental** rather than **physical** force
  - Computers can assist with decision making, model and predict outcomes, etc.
- The computer is (arguably) the **defining technology** of our era
  - cf. “Information Age”
- **Programming Languages** are the mechanism for communicating with and commanding the computer

The Computer is a Unique Machine

- The computer is one of the few “machines” (e.g., lever, pulley, etc.) in that it magnifies our **mental** rather than **physical** force
  - Computers can assist with decision making, model and predict outcomes, etc.
- The computer is (arguably) the **defining technology** of our era
  - cf. “Information Age”
- **Programming Languages** are the mechanism for communicating with and commanding the computer
  - AI interfaces (e.g., ChatGPT) are becoming popular

The Computer is a Unique Machine

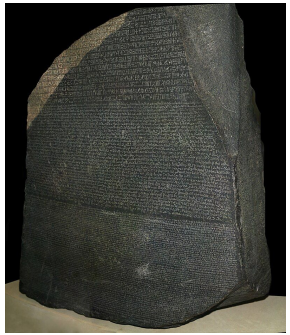
- The computer is one of the few “machines” (e.g., lever, pulley, etc.) in that it magnifies our **mental** rather than **physical** force
  - Computers can assist with decision making, model and predict outcomes, etc.
- The computer is (arguably) the **defining technology** of our era
  - cf. “Information Age”
- **Programming Languages** are the mechanism for communicating with and commanding the computer
  - AI interfaces (e.g., ChatGPT) are becoming popular
  - but they will always be **imprecise**, unlike a real PL

Compilers Todos

- **Most important**: start PA1 (“Rosetta Stone”)

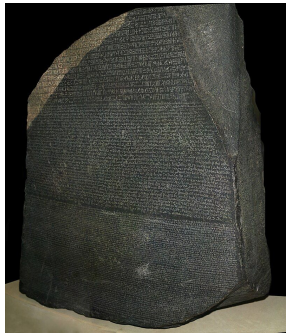
Rosetta Stone

- The first programming assignment (“PA1”) involves writing the same “simple” (50-75 line) program in **four languages**:



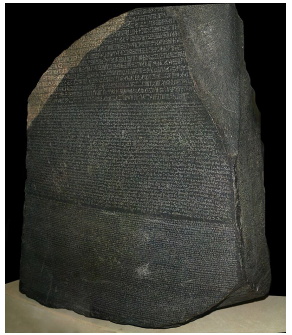
Rosetta Stone

- The first programming assignment (“PA1”) involves writing the same “simple” (50-75 line) program in **four languages**:
  - a language you already know from class



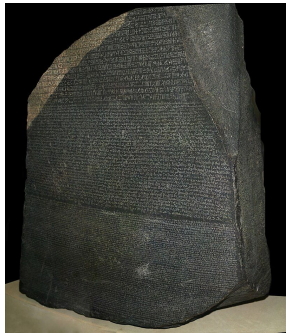
Rosetta Stone

- The first programming assignment (“PA1”) involves writing the same “simple” (50-75 line) program in **four languages**:
  - a language you already know from class
  - a language with an interesting type system



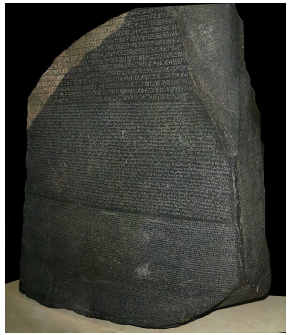
Rosetta Stone

- The first programming assignment (“PA1”) involves writing the same “simple” (50-75 line) program in **four languages**:
  - a language you already know from class
  - a language with an interesting type system
  - a functional language



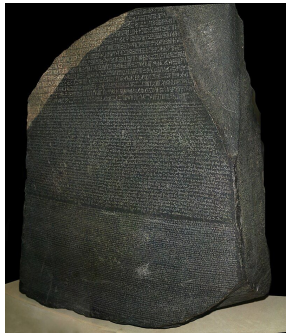
Rosetta Stone

- The first programming assignment (“PA1”) involves writing the same “simple” (50-75 line) program in **four languages**:
  - a language you already know from class
  - a language with an interesting type system
  - a functional language
  - COOL (“Classroom Object-Oriented Language”)
    - this is the source language for the course project



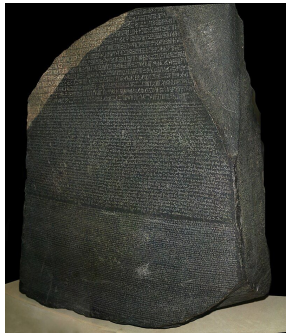
Rosetta Stone

- The first programming assignment (“PA1”) involves writing the same “simple” (50-75 line) program in **four languages**:
  - a language you already know from class
  - a language with an interesting type system
  - a functional language
  - COOL (“Classroom Object-Oriented Language”)
    - this is the source language for the course project
- PA1c1, which requires **only the first**, is **due Monday**



Rosetta Stone

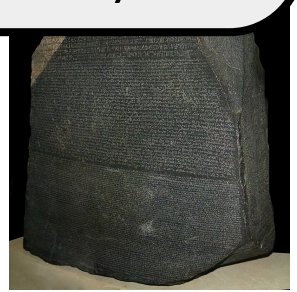
- The first programming assignment (“PA1”) involves writing the same “simple” (50-75 line) program in **four languages**:
  - a language you already know from class
  - a language with an interesting type system
  - a functional language
  - COOL (“Classroom Object-Oriented Language”)
    - this is the source language for the course project



- PA1c1, which requires **only the first**, is **due Monday**
  - if you struggle with getting PA1c1 done by Monday, consider whether this class is right for you: assignments only get harder

Rosetta Stone

Rest due soon:
- one more due next Thursday
- other 2 due Monday 2/3



- The first programming assignment ("PA1") involv
  same "simple" (50-75 line) program in **four langu**
  - a language you already know from class
  - a language with an interesting type system
  - a functional language
  - COOL ("Classroom Object-Oriented Language")
    - this is the source language for the course project
- PA1c1, which requires **only the first**, is **due Monday**
  - if you struggle with getting PA1c1 done by Monday, consider whether this class is right for you: assignments only get harder

Compilers Todos

- **Most important**: start PA1 (“Rosetta Stone”)

Compilers Todos

- **Most important**: start PA1 (“Rosetta Stone”)
- Read the **course website**, especially the syllabus
  - Decide if this class matches your goals
    - if you’re unsure or have more questions, my office hours are this afternoon (3:30-4:30pm, GITC 4314)
    - can also ask on Discord (DMs okay)

Compilers Todos

- **Most important**: start PA1 (“Rosetta Stone”)
- Read the **course website**, especially the syllabus
  - Decide if this class matches your goals
    - if you’re unsure or have more questions, my office hours are this afternoon (3:30-4:30pm, GITC 4314)
    - can also ask on Discord (DMs okay)
- Read the **Cool Reference Manual** (CRM)
  - There **will** be a quiz on Cool next week
    - anything in the syllabus is fair game, too