Compiler Structure

Martin Kellogg

Today’s Agenda

- **Social contract of a compiler**
- Compiler frontends
  - Lexing
  - Parsing
    - our first intermediate representation: abstract syntax trees

Surprise Bioethics

“Primum non nocere”

Surprise Bioethics

“Primum non nocere” - “First, do no harm.”

Surprise Bioethics

“Primum non nocere” - “First, do no harm.”

- one of the key tenets of bioethics
  - e.g., the Hippocratic Oath that doctors take

Surprise Bioethics

“Primum non nocere” - “First, do no harm.”

- one of the key tenets of bioethics
  - e.g., the Hippocratic Oath that doctors take
- also a key tenet in compilers:

First Rule of Compilers: don’t change semantics

Surprise Bioethics

“Primum non nocere” - “First, do no harm.”

- one of the key tenets of bioethics
  - e.g., the Hippocratic Oath that doctors take
- also a key tenet in compilers:

> **First Rule of Compilers**: don’t change semantics

- i.e., don’t change the meaning of the program

Surprise Bioethics

“Primum non nocere” - “First, do no harm.”

- one of the key tenets of bioethics
  - e.g., the Hippocratic Oath that doctors take
- also a key tenet in compilers:

> **First Rule of Compilers**: don’t change semantics

- i.e., don’t change the meaning of the program
- this is the **contract** that a compiler promises to its users

Syntax and Semantics

Definition: the ***syntax*** of a programming language is how programs are written down

- roughly “spelling and grammar”

Syntax and Semantics

Definition: the ***syntax*** of a programming language is how programs are written down

- roughly “spelling and grammar”

Definition: the ***semantics*** of a programming language explains what a given program **means**

- Two relevant kinds of semantics (for now):

Syntax and Semantics

Definition: the ***syntax*** of a programming language is how programs are written down

- roughly “spelling and grammar”

Definition: the ***semantics*** of a programming language explains what a given program **means**

- Two relevant kinds of semantics (for now):
  - ***static semantics***: roughly, the state space of the program

Syntax and Semantics

Definition: the ***syntax*** of a programming language is how programs are written down

- roughly “spelling and grammar”

Definition: the ***semantics*** of a programming language explains what a given program **means**

- Two relevant kinds of semantics (for now):
  - ***static semantics***: roughly, the state space of the program
  - ***dynamic semantics***: in some specific execution, what does this program actually evaluate to?

Why Two Semantics?

Why Two Semantics?

- A **static** semantics is useful for proving that certain behaviors do or do not occur

Why Two Semantics?

- A **static** semantics is useful for proving that certain behaviors do or do not occur
  - e.g., typechecking implies a particular static semantics

Why Two Semantics?

- A **static** semantics is useful for proving that certain behaviors do or do not occur
  - e.g., typechecking implies a particular static semantics
  - one way to think about this: the static semantics is the **union** of the program's dynamic semantics (for all possible inputs)

Why Two Semantics?

- A **static** semantics is useful for proving that certain behaviors do or do not occur
  - e.g., typechecking implies a particular static semantics
  - one way to think about this: the static semantics is the **union** of the program's dynamic semantics (for all possible inputs)
- We want some **dynamic** semantics to be impossible!

Why Two Semantics?

- A **static** semantics is useful for proving that certain behaviors do or do not occur
  - e.g., typechecking implies a particular static semantics
  - one way to think about this: the static semantics is the **union** of the program's dynamic semantics (for all possible inputs)
- We want some **dynamic** semantics to be impossible!
  - e.g., `1 + "hello"` should not have a dynamic semantics

Why Two Semantics?

- A **static** semantics is useful for proving that certain behaviors do or do not occur
  - e.g., typechecking implies a particular static semantics
  - one way to think about this: the static semantics is the **union** of the program's dynamic semantics (for all possible inputs)
- We want some **dynamic** semantics to be impossible!
  - e.g., `1 + "hello"` should not have a dynamic semantics
    - typechecking **prevents** this "program" from ever being evaluated!

Rules of Compilers

Rules of Compilers

1. **First, don’t change the semantics**

Rules of Compilers

1. **First, don't change the semantics**
    - from the textbook:

The compiler must preserve the meaning of the program being compiled.

Rules of Compilers

1. **First, don't change the semantics**
   - from the textbook:

The compiler must preserve the meaning of the program being compiled.

This Rule is referring to the program's ***static*** semantics: we must not change the program's behavior for **any** input!

Rules of Compilers

1. **First, don't change the semantics**
   - from the textbook:

The compiler must preserve the meaning of the program being compiled.

This Rule is referring to the program's ***static*** semantics: we must not change the program's behavior for **any** input!

2. **Second, try to improve the program ("try to help")**

Rules of Compilers

1. **First, don't change the semantics**
   - from the textbook:

The compiler must preserve the meaning of the program being compiled.

This Rule is referring to the program's ***static*** semantics: we must not change the program's behavior for **any** input!

2. **Second, try to improve the program ("try to help")**
   - from the textbook:

The compiler must improve the input program in some discernible way.

Rules of Compilers

1. **First, don't change the semantics**
   - from the textbook:

The compiler must preserve the meaning of the program being compiled.

This Rule is referring to the program's ***static*** semantics: we must not change the program's behavior for **any** input!

2. **Second, try to improve th**
   - from the textbook:

The compiler must improve the *ay.*

If there is ever a **conflict** between these Rules, which one takes precedence?

Rules of Compilers

1. **First, don't change the semantics**
   - from the textbook:

The compiler must preserve the meaning of the program being compiled.

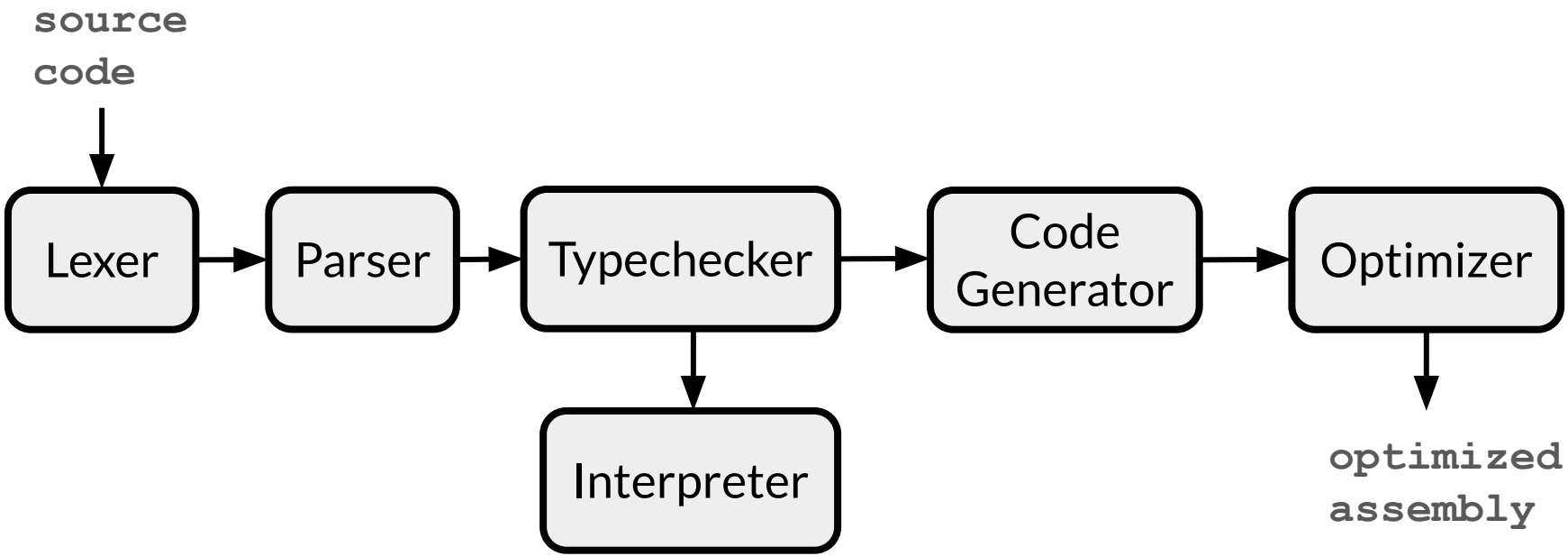
This Rule is referring to the program's ***static*** semantics: we must not change the program's behavior for **any** input!

2. **Second, try to improve th**
   - from the textbook:

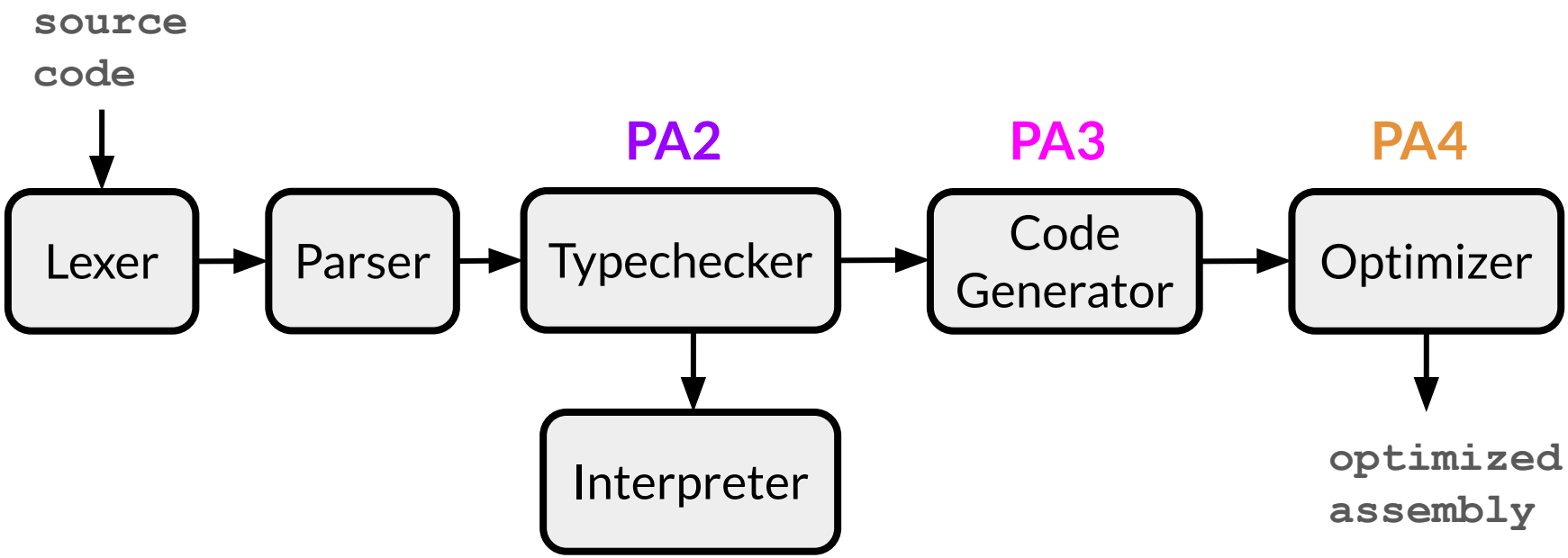
The compiler must improve the *ay.*

If there is ever a **conflict** between these Rules, which one takes precedence?
The First Rule!

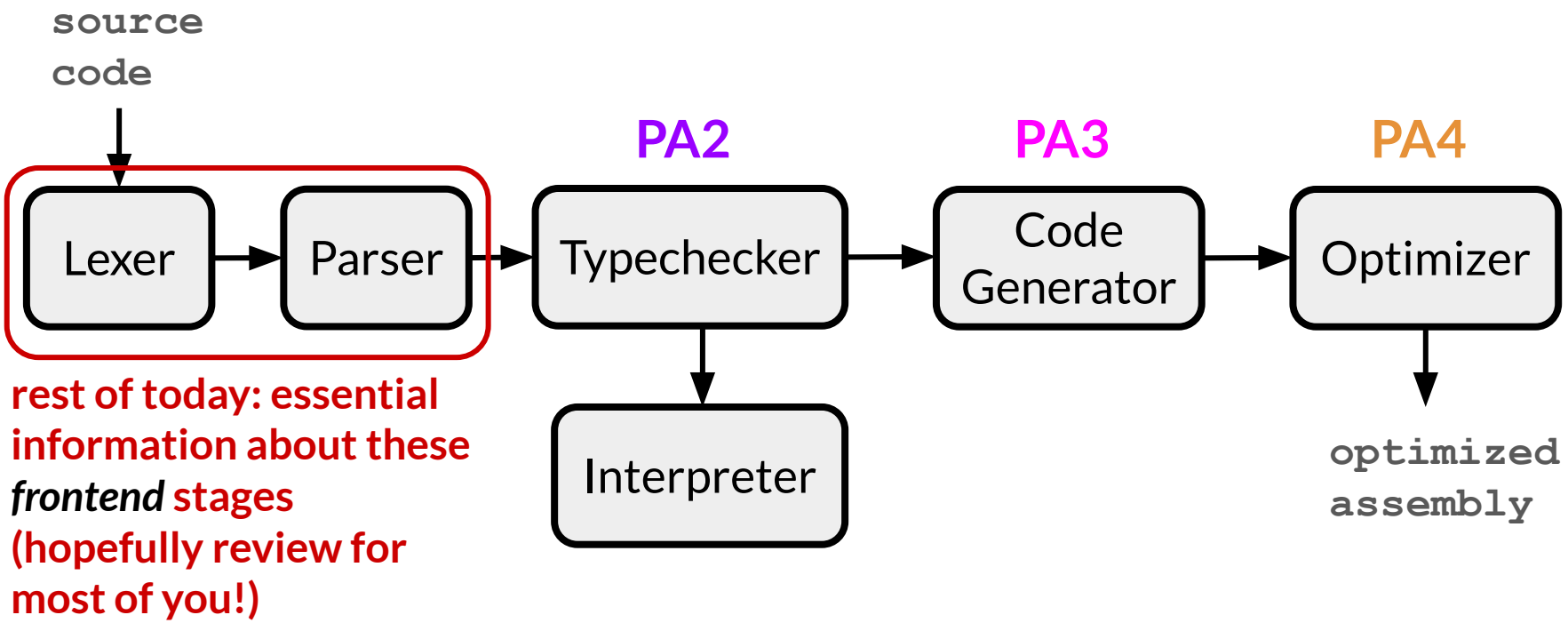
Traditional compiler/interpreter structure



Traditional compiler/interpreter structure



Traditional compiler/interpreter structure



Today’s Agenda

- Social contract of a compiler
- **Compiler frontends**
  - **Lexing**
  - Parsing
    - our first intermediate representation: abstract syntax trees

Lexical analysis

- A ***lexical analyzer*** (or ***lexer***) divides program text into “tokens”

Lexical analysis

- A ***lexical analyzer*** (or ***lexer***) divides program text into “tokens”
  - i.e., it is a function of type `string -> token list`

Lexical analysis

- A ***lexical analyzer*** (or ***lexer***) divides program text into “tokens”
  - i.e., it is a function of type `string -> token list`
- A ***token*** is a **syntactic category**

Lexical analysis

- A ***lexical analyzer*** (or ***lexer***) divides program text into “tokens”
  - i.e., it is a function of type `string -> token list`
- A ***token*** is a **syntactic category**
  - in English:
    - “noun”, “verb”, “adjective”

Lexical analysis

- A ***lexical analyzer*** (or ***lexer***) divides program text into “tokens”
  - i.e., it is a function of type `string -> token list`
- A ***token*** is a **syntactic category**
  - in English:
    - “noun”, “verb”, “adjective”
  - in a programming language:
    - identifier, integer constant, keyword, whitespace

Lexical analysis

- A ***lexical analyzer*** (or ***lexer***) divides program text into “tokens”
  - i.e., it is a function of type `string -> token list`
- A ***token*** is a **syntactic category**
  - in English:
    - “noun”, “verb”, “adjective”
  - in a programming language:
    - identifier, integer constant, keyword, whitespace
- Parsers rely on token distinctions
  - e.g., identifiers are treated differently than keywords

Lexical analysis: tokens

- Note that tokens correspond to **sets** of strings

Lexical analysis: tokens

- Note that tokens correspond to **sets** of strings
  - e.g., an **identifier** token is any string of letters or digits that starts with a letter (in most languages)

Lexical analysis: tokens

- Note that tokens correspond to **sets** of strings
  - e.g., an **identifier** token is any string of letters or digits that starts with a letter (in most languages)
- A lexer needs to do two things:
  - recognize substrings that correspond to tokens

Lexical analysis: tokens

- Note that tokens correspond to **sets** of strings
  - e.g., an **identifier** token is any string of letters or digits that starts with a letter (in most languages)
- A lexer needs to do two things:
  - recognize substrings that correspond to tokens
  - return the ***lexeme*** of the token
    - the lexeme is the specific substring

Lexical analysis: tokens

- Note that tokens correspond to **sets** of strings
  - e.g., an **identifier** token is any string of letters or digits that starts with a letter (in most languages)
- A lexer needs to do two things:
  - recognize substrings that correspond to tokens
  - return the ***lexeme*** of the token
    - the lexeme is the specific substring
    - e.g., in Cool, `myVar` is an identifier token whose lexeme is “myVar”

Lexical analysis: implementation

- Lexers usually **discard** “uninteresting” tokens that don’t contribute to parsing
  - e.g., whitespace or comments

Lexical analysis: implementation

- Lexers usually **discard** “uninteresting” tokens that don’t contribute to parsing
  - e.g., whitespace or comments
- Lexing a string is typically implemented by reading the string from start to end, **one token** at a time

Lexical analysis: implementation

- Lexers usually **discard** “uninteresting” tokens that don’t contribute to parsing
  - e.g., whitespace or comments
- Lexing a string is typically implemented by reading the string from start to end, **one token** at a time
  - Lexers generally require some amount of ***lookahead*** to successfully partition a string
  - e.g., “i” vs “if”

Lexical analysis: implementation

- Lexers usually **discard** “uninteresting” tokens that don’t contribute to parsing
  - e.g., whitespace or comments
- Lexing a string is typically implemented by reading the string from start to end, **one token** at a time
  - Lexers generally require some amount of ***lookahead*** to successfully partition a string
  - e.g., “i” vs “if”

looks like an identifier at first…

Lexical analysis: implementation

- Lexers usually **discard** “uninteresting” tokens that don’t contribute to parsing
  - e.g., whitespace or comments
- Lexing a string is typically implemented by reading the string from start to end, **one token** at a time
  - Lexers generally require some amount of ***lookahead*** to successfully partition a string
  - e.g., “i” vs “if”

looks like an identifier at first…but it’s actually a keyword

Lexical analysis: theoretical grounding

Lexical analysis: theoretical grounding

Definition: Let Σ (“sigma”) be a set of characters. A ***language over Σ*** is a set of strings of characters drawn from Σ. Σ is called the ***alphabet***.

Lexical analysis: theoretical grounding

Definition: Let Σ ("sigma") be a set of characters. A ***language over Σ*** is a set of strings of characters drawn from Σ. Σ is called the ***alphabet***.

- e.g., alphabet = English characters, language = valid English sentences

Lexical analysis: theoretical grounding

Definition: Let Σ ("sigma") be a set of characters. A ***language over Σ*** is a set of strings of characters drawn from Σ. Σ is called the ***alphabet***.

- e.g., alphabet = English characters, language = valid English sentences
  - note that **not** every string of English characters is a valid sentence! E.g., "xagea' iwej"

Lexical analysis: theoretical grounding

Definition: Let Σ ("sigma") be a set of characters. A ***language over Σ*** is a set of strings of characters drawn from Σ. Σ is called the ***alphabet***.

- e.g., alphabet = English characters, language = valid English sentences
  - note that **not** every string of English characters is a valid sentence! E.g., "xagea' iwej"
- for lexing, we care about the ***regular languages***, which you might remember from your CS theory class

Lexical analysis: regular languages

Definition: a ***regular language*** is a language that can be recognized by a ***regular expression*** or, equivalently, by a ***deterministic finite automaton***.

Lexical analysis: regular languages

Definition: a ***regular language*** is a language that can be recognized by a ***regular expression*** or, equivalently, by a ***deterministic finite automaton***.

- More formally, the set of regular languages over some alphabet Σ is defined as:
  - the empty language Ø is regular
  - for each *s* in Σ, the singleton set { *s* } is regular
  - if *A* is a regular language, then A^* is a regular language
  - if *A* and *B* are regular languages, then $A \cup B$ and $A \bullet B$ are regular

Lexical analysis: regular languages

Definition: a ***regular language*** is a language that can be recognized by a ***regular expression*** or, equivalently, by a ***deterministic***

- More formally, the set of regular languages over Σ is defined as:
  - the empty language $\emptyset$ is regular
  - for each s in Σ, the singleton set $\{ s \}$ is regular
  - if A is a regular language, then A^* is a regular language
  - if A and B are regular languages, then $A \cup B$ and $A \bullet B$ are regular

This ***Kleene Star*** operator indicates repetition ("zero or more times")

Lexical analysis: regular languages

Definition: a ***regular language*** is a language that can be recognized by a ***regular expression*** or, equivalently, by a ***deterministic finite automaton***.

- More formally, the set of regular languages over some alphabet Σ is defined as:
  - the empty language ∅
  - for each *s* in Σ, the si… regular
  - if *A* is a regular language, then *A** is a regular language
  - if *A* and *B* are regular languages, then $A \cup B$ and $A \bullet B$ are regular

Union ("all the strings in both *A* and *B*")

Lexical analysis: regular languages

Definition: a ***regular language*** is a language that can be recognized by a ***regular expression*** or, equivalently, by a ***deterministic finite automaton***.

- More formally, the set of regular languages over some alphabet Σ is defined as:
  - the empty language ∅
  - for each *s* in Σ, the si
  - if *A* is a regular language, then *A** is a regular language
  - if *A* and *B* are regular languages, then $A \cup B$ and $A \bullet B$ are regular

Concatenation ("each string in *A* followed by each string in *B*")

Lexical analysis: in practice

Lexical analysis: in practice

- In practice, we use a ***lexical analysis generator*** like `ply`, `flex`, or `ocamllex` to build lexers for us

Lexical analysis: in practice

- In practice, we use a ***lexical analysis generator*** like `ply`, `flex`, or `ocamllex` to build lexers for us
  - these tools take regular expressions as input…

Lexical analysis: in practice

- In practice, we use a ***lexical analysis generator*** like `ply`, `flex`, or `ocamllex` to build lexers for us
  - these tools take regular expressions as input…
  - …and then convert into **finite automata**, which they implement using **lookup tables**…

Lexical analysis: in practice

- In practice, we use a ***lexical analysis generator*** like `ply`, `flex`, or `ocamllex` to build lexers for us
  - these tools take regular expressions as input…
  - …and then convert into **finite automata**, which they implement using **lookup tables**…
  - to eventually generate performant lexing code

Trivia Break: Computing History

Around 825 CE, this Persian scientist and polymath wrote *kitāb al-ḥisāb al-hindī* (“Book of Indian computation”) and *kitab al-jam' wa'l-tafriq al-ḥisāb al-hindī* (“Addition and subtraction in Indian arithmetic”). Despite these works later having a major influence on computer science, he is best known for his popularizing treatise on algebra (the “*Al-Jabr*”). The word “algorithm” is derived from his name.

Today’s Agenda

- Social contract of a compiler
- **Compiler frontends**
  - Lexing
  - **Parsing**
    - our first intermediate representation: abstract syntax trees

Parsing

Definition: A ***parser*** takes a sequence of tokens as input. If the input is valid, it produces a ***abstract syntax tree*** (or ***derivation*** or ***parse tree***).

Parsing

Definition: A ***parser*** takes a sequence of tokens as input. If the input is valid, it produces a ***abstract syntax tree*** (or ***derivation*** or ***parse tree***).

- Otherwise, it produces an error
  - e.g., “parse error on line 3”

Parsing

Definition: A ***parser*** takes a sequence of tokens as input. If the input is valid, it produces a ***abstract syntax tree*** (or ***derivation*** or ***parse tree***).

- Otherwise, it produces an error
  - e.g., “parse error on line 3”
- Comparison of lexing vs. parsing:

Phase	*Input*	*Output*
Lexer		
Parser		

Parsing

Definition: A ***parser*** takes a sequence of tokens as input. If the input is valid, it produces a ***abstract syntax tree*** (or ***derivation*** or ***parse tree***).

- Otherwise, it produces an error
  - e.g., “parse error on line 3”
- Comparison of lexing vs. parsing:

Phase	*Input*	*Output*
Lexer	Sequence of characters	**Sequence of tokens**
Parser		

Parsing

Definition: A ***parser*** takes a sequence of tokens as input. If the input is valid, it produces a ***abstract syntax tree*** (or ***derivation*** or ***parse tree***).

- Otherwise, it produces an error
  - e.g., “parse error on line 3”
- Comparison of lexing vs. parsing:

Phase	*Input*	*Output*
Lexer	Sequence of characters	**Sequence of tokens**
Parser	**Sequence of tokens**	Parse tree

Parser Output: Abstract Syntax Trees (ASTs)

- Implicit in our discussion of compilation so far: programs in the compiler's source language are **data**, from the perspective of the compiler

Parser Output: Abstract Syntax Trees (ASTs)

- Implicit in our discussion of compilation so far: programs in the compiler's source language are **data**, from the perspective of the compiler
  - this is a powerful perspective: you know how to write programs that **manipulate** data already

Parser Output: Abstract Syntax Trees (ASTs)

- Implicit in our discussion of compilation so far: programs in the compiler's source language are **data**, from the perspective of the compiler
  - this is a powerful perspective: you know how to write programs that **manipulate** data already
  - fundamentally, your compiler is just another program that manipulates data - the only difference is that the data, in this case, are also programs

Parser Output: Abstract Syntax Trees (ASTs)

- Implicit in our discussion of compilation so far: programs in the compiler's source language are **data**, from the perspective of the compiler
  - this is a powerful perspective: you know how to write programs that **manipulate** data already
  - fundamentally, your compiler is just another program that manipulates data - the only difference is that the data, in this case, are also programs
- So, how do we **represent programs as data**?

Parser Output: Abstract Syntax Trees (ASTs)

- Implicit in our discussion of compilation so far: programs in the compiler's source language are **data**, from the perspective of the compiler
  - this is a powerful perspective: you know how to write programs that **manipulate** data already
  - fundamentally, your compiler is just another program that manipulates data - the only difference is that the data, in this case, are also programs
- So, how do we **represent programs as data**?
  - there are many ways, but today we'll discuss two…

Treating programs as data: two ways

#1: treat the program **as a string**

Treating programs as data: two ways

#1: treat the program **as a string**

- allows us to easily decide **syntactic** properties

Treating programs as data: two ways

#1: treat the program **as a string**

- allows us to easily decide **syntactic** properties
  - for example, checking if a program contains the text “foo”

Treating programs as data: two ways

#1: treat the program **as a string**

- allows us to easily decide **syntactic** properties
  - for example, checking if a program contains the text “foo”
- key downside: cannot use the program’s **semantics**

Treating programs as data: two ways

#1: treat the program **as a string**

- allows us to easily decide **syntactic** properties
  - for example, checking if a program contains the text “foo”
- key downside: cannot use the program’s **semantics**
  - semantics are relevant for properties related to **context** - that is, where the question to be decided depends on the rest of the program

Treating programs as data: two ways

#2: treat the program **as a tree**

Treating programs as data: two ways

#2: treat the program **as a tree**

Definition: an ***abstract syntax tree*** (or ***AST***) is a tree-based representation of a program's syntactic structure

Treating programs as data: two ways

#2: treat the program **as a tree**

Definition: an ***abstract syntax tree*** (or ***AST***) is a tree-based representation of a program's syntactic structure

- usually produced by a **parser**

Treating programs as data: two ways

#2: treat the program **as a tree**

Definition: an ***abstract syntax tree*** (or ***AST***) is a tree-based representation of a program's syntactic structure

- usually produced by a **parser**
- nodes in the tree represent syntactic constructs

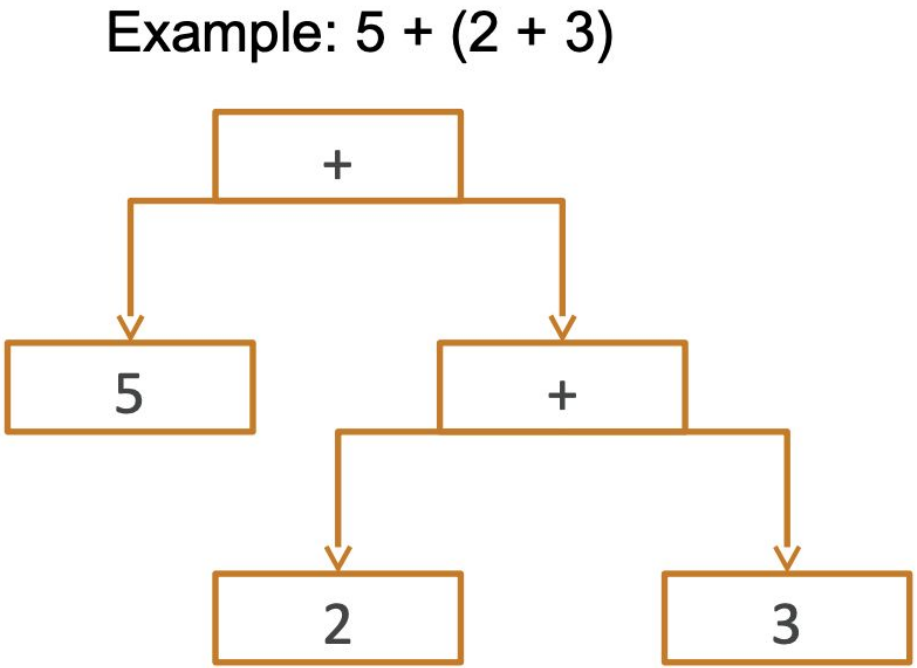
Treating programs as data: two ways

#2: treat the program **as a tree**

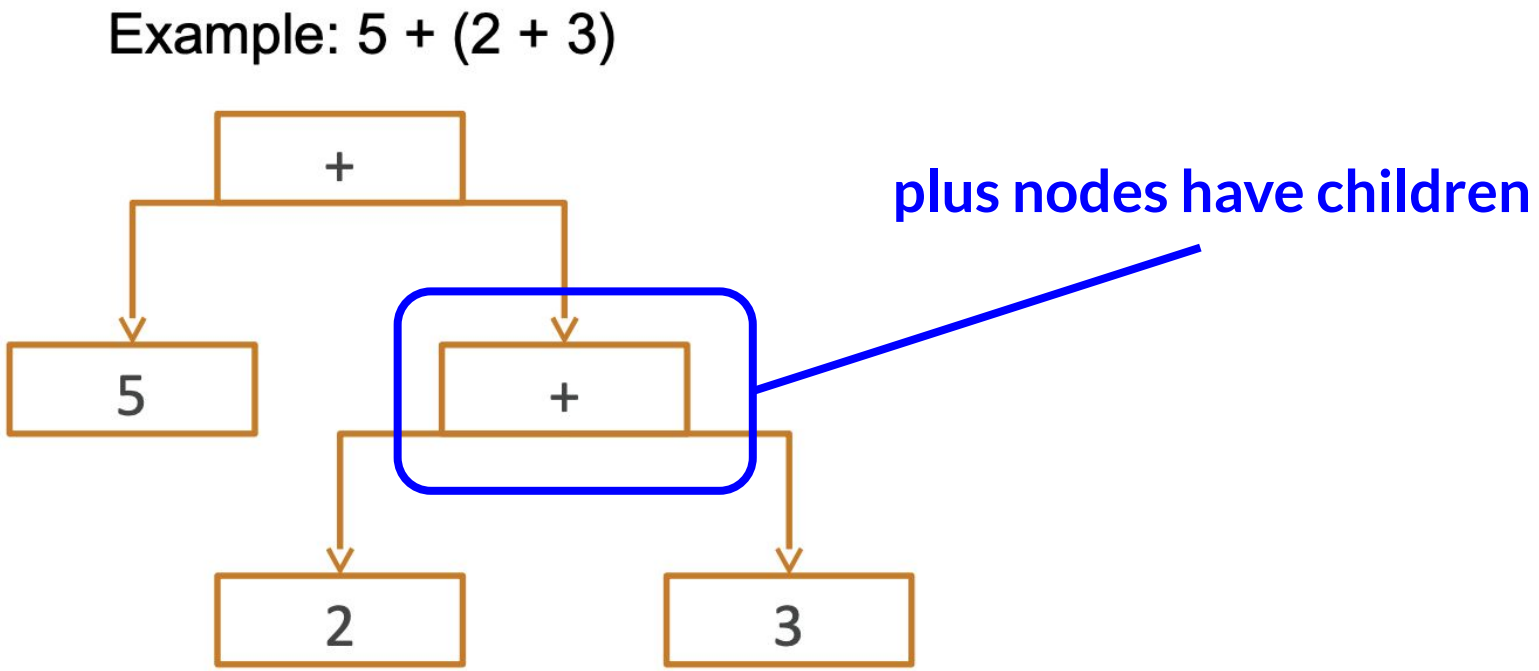
Definition: an ***abstract syntax tree*** (or ***AST***) is a tree-based representation of a program's syntactic structure

- usually produced by a **parser**
- nodes in the tree represent syntactic constructs
  - parent-child relationships in the AST represent **compound expressions** in the source code (e.g., a "plus node" might have two children: the left and right side expressions)

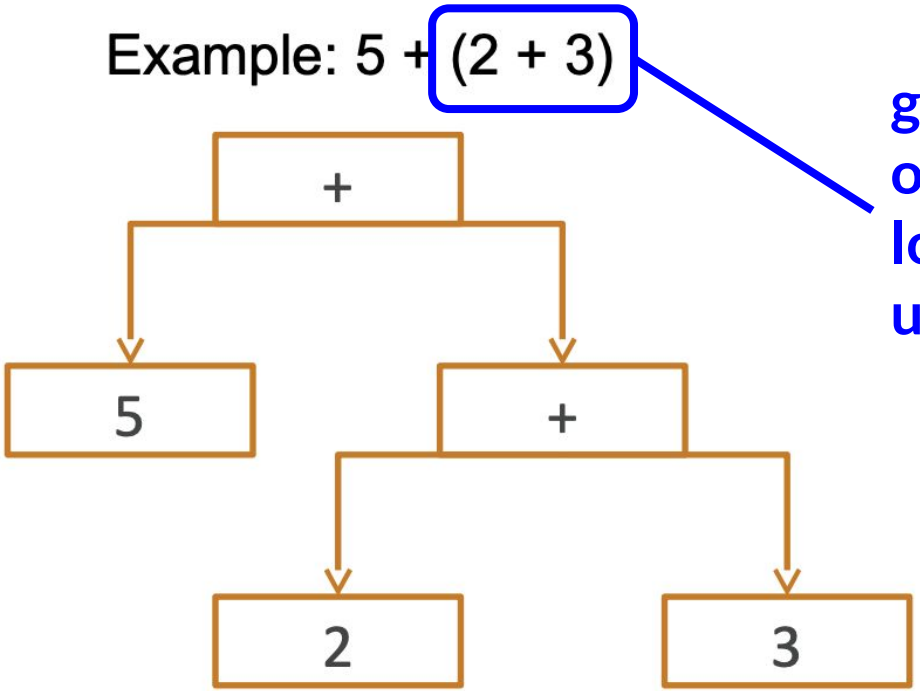
Treating programs as data: AST example



Treating programs as data: AST example

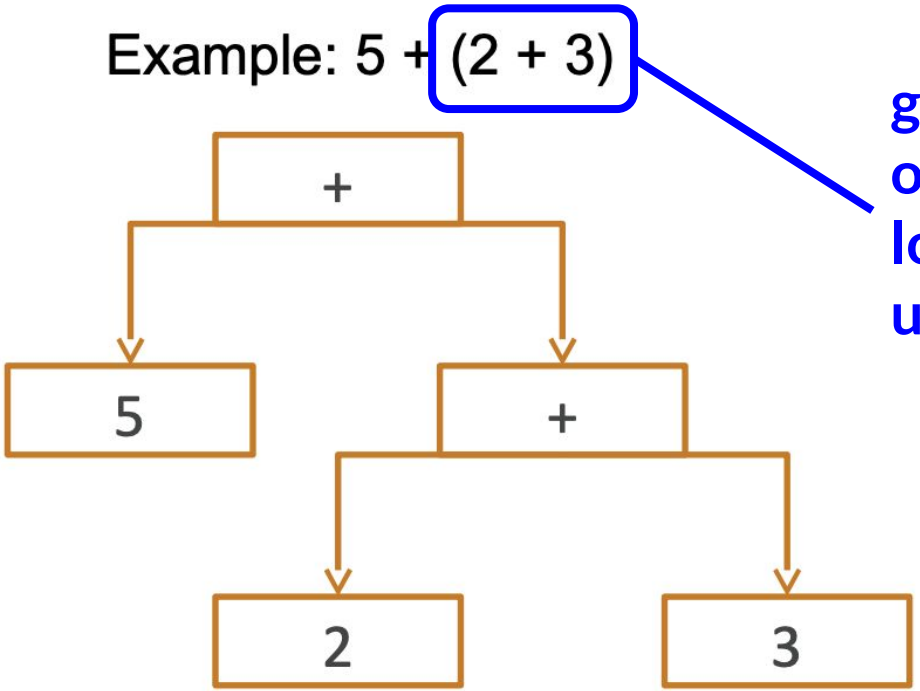


Treating programs as data: AST example



grouping parentheses and other disambiguation is no longer necessary (AST is unambiguous, unlike text)

Treating programs as data: AST example



grouping parentheses and other disambiguation is no longer necessary (AST is unambiguous, unlike text)

In PA2, you will take this structure as input and **annotate** each node with a type

Parsing: errors

- Not all sequences of tokens are programs and correspond to an AST. E.g.:
  - `then x * / + 3 while x ; y z then`

Parsing: errors

- Not all sequences of tokens are programs and correspond to an AST. E.g.:
  - `then x * / + 3 while x ; y z then`
- The parser must distinguish between **valid** and **invalid** sequences of tokens

Parsing: errors

- Not all sequences of tokens are programs and correspond to an AST. E.g.:
  - `then x * / + 3 while x ; y z then`
- The parser must distinguish between **valid** and **invalid** sequences of tokens
- We need:
  - A language or **formalism** to describe valid sequences of tokens

Parsing: errors

- Not all sequences of tokens are programs and correspond to an AST. E.g.:
  - `then x * / + 3 while x ; y z then`
- The parser must distinguish between **valid** and **invalid** sequences of tokens
- We need:
  - A language or **formalism** to describe valid sequences of tokens
  - A method (i.e., an **algorithm**) for distinguishing between valid and invalid token sequences

Aside: computability theory

Definition: an ***algorithm*** is a finite sequence of mathematically rigorous instructions, typically used to solve a class of specific problems or to perform a computation. [Wikipedia's *Algorithm*]

Aside: computability theory

Definition: an ***algorithm*** is a finite sequence of mathematically rigorous instructions, typically used to solve a class of specific problems or to perform a computation. [Wikipedia's *Algorithm*]

- when we talk about algorithms in Computer Science, we generally are referring to the **computable functions**, which are the formalized analogue of the intuitive notion above

Aside: computability theory

Definition: an ***algorithm*** is a finite sequence of mathematically rigorous instructions, typically used to solve a class of specific problems or to perform a computation. [Wikipedia's *Algorithm*]

- when we talk about algorithms in Computer Science, we generally are referring to the **computable functions**, which are the formalized analogue of the intuitive notion above
  - a function is ***computable*** if there exists an algorithm that can do the job of the function which we can execute in one of our models of computation (e.g., the lambda calculus or a Turing machine)

Parsing Theory

- So, what's the **formalism** for parsing?

Parsing Theory

- So, what's the **formalism** for parsing?
  - Regular expressions/DFAs (as we used for lexing) don't work

Parsing Theory

- So, what's the **formalism** for parsing?
  - Regular expressions/DFAs (as we used for lexing) don't work
    - because parsing requires **context**!

Parsing Theory

- So, what's the **formalism** for parsing?
  - Regular expressions/DFAs (as we used for lexing) don't work
    - because parsing requires **context**!
    - e.g., we need to check if parentheses are matched

Parsing Theory

- So, what's the **formalism** for parsing?
  - Regular expressions/DFAs (as we used for lexing) don't work
    - because parsing requires **context**!
    - e.g., we need to check if parentheses are matched
- Perhaps you recall from your CS theory class what kind of formalism is appropriate for checking the language of balanced parentheses?

Parsing Theory

- So, what's the **formalism** for parsing?
  - Regular expressions/DFAs (as we used for lexing) don't work
    - because parsing requires **context**!
    - e.g., we need to check if parentheses are matched
- Perhaps you recall from your CS theory class what kind of formalism is appropriate for checking the language of balanced parentheses?
  - the ***context-free grammars*** or ***pushdown automata***

Parsing Theory: Context-free Grammars

- How do we know? Programming languages have **recursive structure**

Parsing Theory: Context-free Grammars

- How do we know? Programming languages have **recursive structure**
- E.g., consider the language of arithmetic expressions with integers, +, *, and ( )

Parsing Theory: Context-free Grammars

- How do we know? Programming languages have **recursive structure**
- E.g., consider the language of arithmetic expressions with integers, +, *, and ( )
- An **expression** in this language is either:
  - an integer
  - an **expression** followed by “+” followed by an **expression**
  - an **expression** followed by “*” followed by an **expression**
  - a ‘(‘ followed by an **expression** followed by ‘)’

Parsing Theory: Context-free Grammars

- How do we know? Programming languages have **recursive structure**
- E.g., consider the language of arithmetic expressions with integers, +, *, and ( )
- An **expression** in this language is either:
  - an integer
  - an **expression** followed by "+" followed by an **expression**
  - an **expression** followed by "*" followed by an **expression**
  - a '(' followed by an **expression** followed by ')'
- 5 , 5+3, (5+3)*7 are expressions in this language

Parsing Theory: Context-free Grammars

- An alternative notation for the language on the last slide:
  - E -> int
  - E -> E + E
  - E -> E * E
  - E -> (E)

Parsing Theory: Context-free Grammars

- An alternative notation for the language on the last slide:
  - **E** -> int
  - **E** -> **E** + **E**
  - **E** -> **E** * **E**
  - **E** -> (**E**)
- We can view these rules as **rewrite rules**

Parsing Theory: Context-free Grammars

- An alternative notation for the language on the last slide:
  - **E** -> int
  - **E** -> **E** + **E**
  - **E** -> **E** * **E**
  - **E** -> (**E**)
- We can view these rules as **rewrite rules**
  - We start with **E** and replace occurrences of **E** with some right-hand side

Parsing Theory: Context-free Grammars

- An alternative notation for the language on the last slide:
  - **E** -> int
  - **E** -> **E** + **E**
  - **E** -> **E** * **E**
  - **E** -> (**E**)
- We can view these rules as **rewrite rules**
  - We start with **E** and replace occurrences of **E** with some right-hand side
  - Eventually, we'll derive an actual program in the language

Parsing Theory: Context-free Grammars

- An alternative notation for the language on the last slide:
  - **E** -> int
  - **E** -> **E** + **E**
  - **E** -> **E** * **E**
  - **E** -> (**E**)
- We can view these rules as **rewrite rules**
  - We start with **E** and replace occurrences of **E** with some right-hand side
  - Eventually, we'll derive an actual program in the language

Note that there is no series of rewrites that allows us to obtain an **invalid** program (e.g., with mismatched parentheses). Why?

Parsing Theory: Context-free Grammars

- The notation on the previous slide is exactly a ***context-free grammar***.

Parsing Theory: Context-free Grammars

- The notation on the previous slide is exactly a ***context-free grammar***.
- Officially, a context-free grammar consists of:

Parsing Theory: Context-free Grammars

- The notation on the previous slide is exactly a ***context-free grammar***.
- Officially, a context-free grammar consists of:
  - A set of ***non-terminals*** N
    - Written in uppercase in these notes

Parsing Theory: Context-free Grammars

- The notation on the previous slide is exactly a ***context-free grammar***.
- Officially, a context-free grammar consists of:
  - A set of ***non-terminals*** N
    - Written in uppercase in these notes
  - A set of ***terminals*** T
    - Lowercase or punctuation in these notes

Parsing Theory: Context-free Grammars

- The notation on the previous slide is exactly a ***context-free grammar***.
- Officially, a context-free grammar consists of:
  - A set of ***non-terminals*** N
    - Written in uppercase in these notes
  - A set of ***terminals*** T
    - Lowercase or punctuation in these notes
  - A ***start symbol*** S (a non-terminal)

Parsing Theory: Context-free Grammars

- The notation on the previous slide is exactly a ***context-free grammar***.
- Officially, a context-free grammar consists of:
  - A set of ***non-terminals*** N
    - Written in uppercase in these notes
  - A set of ***terminals*** T
    - Lowercase or punctuation in these notes
  - A ***start symbol*** S (a non-terminal)
  - A set of ***productions*** (rewrite rules)

Parsing Theory: Context-free Grammars

- Context-free grammar example:
  - E -> int
  - E -> E + E
  - E -> E * E
  - E -> (E)

Parsing Theory: Context-free Grammars

- Context-free grammar example:
  - **E** -> int
  - **E** -> **E** + **E**
  - **E** -> **E** * **E**
  - **E** -> (**E**)
- **E** is the only ***non-terminal***

Parsing Theory: Context-free Grammars

- Context-free grammar example:
  - **E** -> int
  - **E** -> **E** + **E**
  - **E** -> **E** * **E**
  - **E** -> (**E**)
- **E** is the only ***non-terminal***
- int, +, *, (, and) are the ***terminals***
  - called terminals because they are never replaced

Parsing Theory: Context-free Grammars

- Context-free grammar example:
  - **E** -> int
  - **E** -> **E** + **E**
  - **E** -> **E** * **E**
  - **E** -> (**E**)
- **E** is the only ***non-terminal***
- int, +, *, (, and) are the ***terminals***
  - called terminals because they are never replaced
- each bullet above is one of the rewrite rules (***productions***)

Parsing Theory: Context-free Grammars

- Context-free grammar example:
  - **E** -> int
  - **E** -> **E** + **E**
  - **E** -> **E** * **E**
  - **E** -> (**E**)
- **E** is the only ***non-terminal***
- int, +, *, (, and) are the ***terminals***
  - called terminals because they are never replaced
- each bullet above is one of the rewrite rules (***productions***)
- by convention, the first non-terminal in the first production is the ***start symbol***

Parsing Theory: Derivations

- A ***derivation*** is a sequence of productions, starting from the start symbol of the context-free grammar, that produces a string of only terminals

Parsing Theory: Derivations

- A ***derivation*** is a sequence of productions, starting from the start symbol of the context-free grammar, that produces a string of only terminals
  - a derivation can be **drawn as a tree**

Parsing Theory: Derivations

- A ***derivation*** is a sequence of productions, starting from the start symbol of the context-free grammar, that produces a string of only terminals
  - a derivation can be **drawn as a tree**
    - start symbol is the root

Parsing Theory: Derivations

- A ***derivation*** is a sequence of productions, starting from the start symbol of the context-free grammar, that produces a string of only terminals
  - a derivation can be **drawn as a tree**
    - start symbol is the root
    - for each production, add children for each symbol on the right-hand side of the rule as children of the left-hand side non-terminal

Parsing Theory: Derivations

- A ***derivation*** is a sequence of productions, starting from the start symbol of the context-free grammar, that produces a string of only terminals
  - a derivation can be **drawn as a tree**
    - start symbol is the root
    - for each production, add children for each symbol on the right-hand side of the rule as children of the left-hand side non-terminal
  - this derivation **exactly corresponds** to the AST!

Parser History

- In the early days of computing, parsing was the **most expensive** part of a compiler

Parser History

- In the early days of computing, parsing was the **most expensive** part of a compiler
  - e.g., consider original FORTRAN:
    - no semantic analysis or typechecking
    - little optimization
    - code generation is ***syntax-directed***

Parser History

- In the early days of computing, parsing was the **most expensive** part of a compiler
  - e.g., consider original FORTRAN:
    - no semantic analysis or typechecking
    - little optimization
    - code generation is ***syntax-directed***
- Therefore, a lot of early **compilers research** was focused on improving parsing

Parser History

- In the early days of computing, parsing was the **most expensive** part of a compiler
  - e.g., consider original FORTRAN:
    - no semantic analysis or typechecking
    - little optimization
    - code generation is ***syntax-directed***
- Therefore, a lot of early **compilers research** was focused on improving parsing
  - e.g., LL(1) vs LR grammars, Earley parsing, etc.
    - we aren't going to cover these in this class!

Parser History

- These days, parsing is considered a **mostly-solved problem**

Parser History

- These days, parsing is considered a **mostly-solved problem**
  - semantic analysis and optimization times dominate

Parser History

- These days, parsing is considered a **mostly-solved problem**
  - semantic analysis and optimization times dominate
  - parsing is fast enough that it’s not worth optimizing further

Parser History

- These days, parsing is considered a **mostly-solved problem**
  - semantic analysis and optimization times dominate
  - parsing is fast enough that it's not worth optimizing further
- Automatic ***parser generators*** (e.g., `bison` or `ocamlyacc`) can produce a parser implementation from a context-free grammar

Parser History

- These days, parsing is considered a **mostly-solved problem**
  - semantic analysis and optimization times dominate
  - parsing is fast enough that it's not worth optimizing further
- Automatic ***parser generators*** (e.g., `bison` or `ocamlyacc`) can produce a parser implementation from a context-free grammar
  - production compilers, though, typically use a custom ***recursive descent*** parser

Parser History

- These days, parsing is considered a **mostly-solved problem**
  - semantic analysis and optimization times dominate
  - parsing is fast enough that it's not worth optimizing further
- Automatic ***parser generators*** (e.g., `bison` or `ocamlyacc`) can produce a parser implementation from a context-free grammar
  - production compilers, though, typically use a custom ***recursive descent*** parser
    - that is, compiler engineers prioritize **simplicity** over having a fast parser (the speed gains are no longer worthwhile for the technical debt)

Parser History

- These days, parsing is considered a **mostly-solved problem**
  - semantic analysis and optimization times do
  - parsing is fast enough that it's not worth op
- Automatic ***parser generators*** (e.g., `bison` or `oca`
  produce a parser implementation from a contex
  - production compilers, though, typically use a custom ***recursive descent*** parser
    - that is, compiler engineers prioritize **simplicity** over having a fast parser (the speed gains are no longer worthwhile for the technical debt)

Takeaway for you: you can assume that it's **relatively easy** to get an AST

Course Announcements

- PA1 (all four languages!) **due today**

Course Announcements

- PA1 (all four languages!) **due today**
- My OH this week are modified:
  - I will hold OH **today** instead of Wednesday, 4-5pm
  - if you need to meet later in the week, send me an email and we'll arrange something

Course Announcements

- PA1 (all four languages!) **due today**
- My OH this week are modified:
  - I will hold OH **today** instead of Wednesday, 4-5pm
  - if you need to meet later in the week, send me an email and we'll arrange something
- Don't forget: PA2c1 is due **Friday**
  - this is a testing assignment: you'll just write Cool programs

Course Announcements

- PA1 (all four languages!) **due today**
- My OH this week are modified:
  - I will hold OH **today** instead of Wednesday, 4-5pm
  - if you need to meet later in the week, send me an email and we'll arrange something
- Don't forget: PA2c1 is due **Friday**
  - this is a testing assignment: you'll just write Cool programs
- The course staff has become aware of a bug in the Apple Silicon builds for Cool related to newlines
  - No fix is available. Use Ubuntu instead.