Lecture 10

COMPILER DESIGN

Announcements

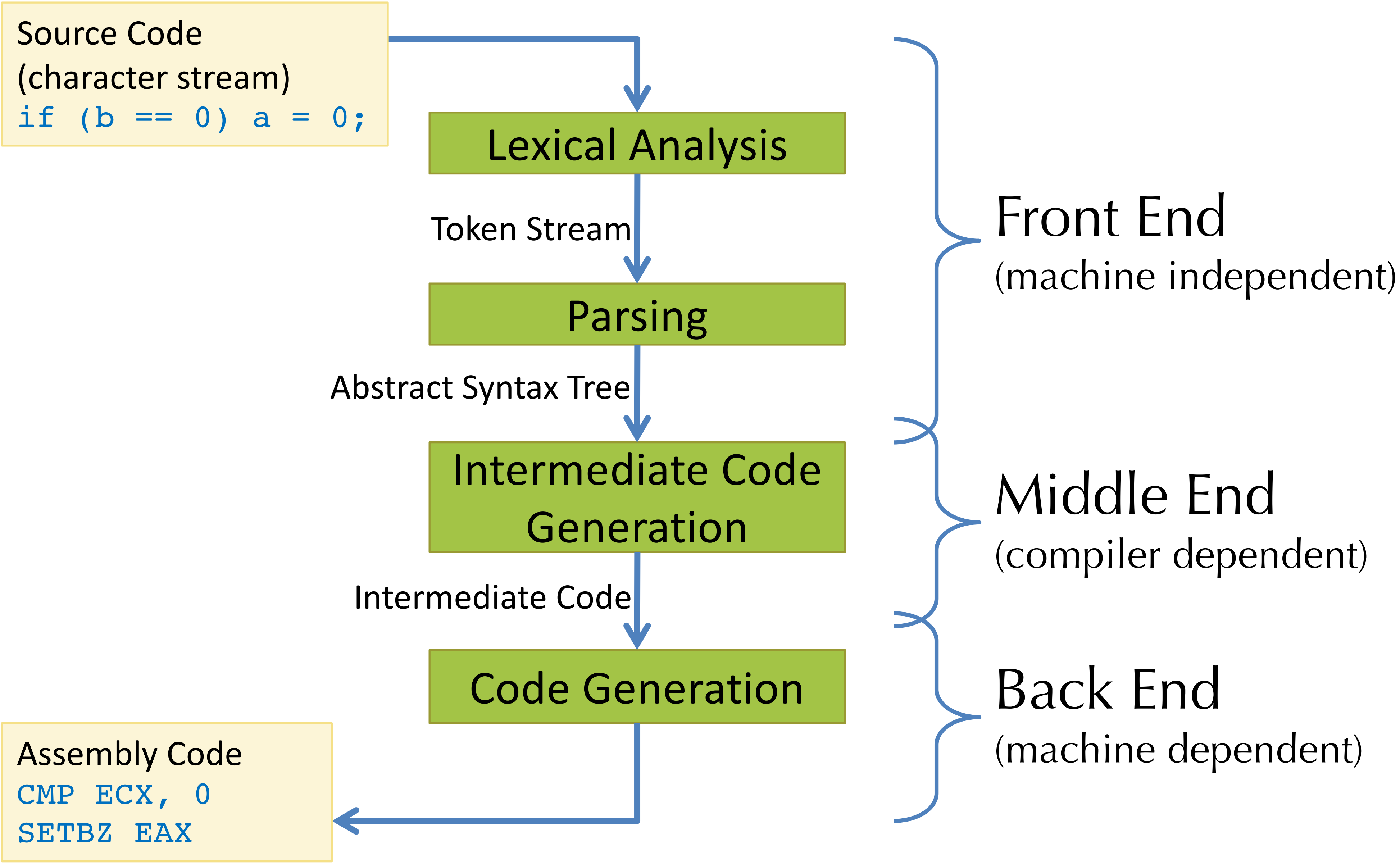
- **HW2:** due soon

- **HW3:** Compiling LLVMlite
  - Familiarize yourself with (a subset of) the LLVM IR
  - Implement a translation down to (inefficient) X86lite

Creating an abstract representation of program syntax

PARSING

(Simplified) Compiler Structure



Source Code
(character stream)
`if (b == 0) a = 0;`

Lexical Analysis

Token Stream

Parsing

Abstract Syntax Tree

Intermediate Code Generation

Intermediate Code

Code Generation

Assembly Code

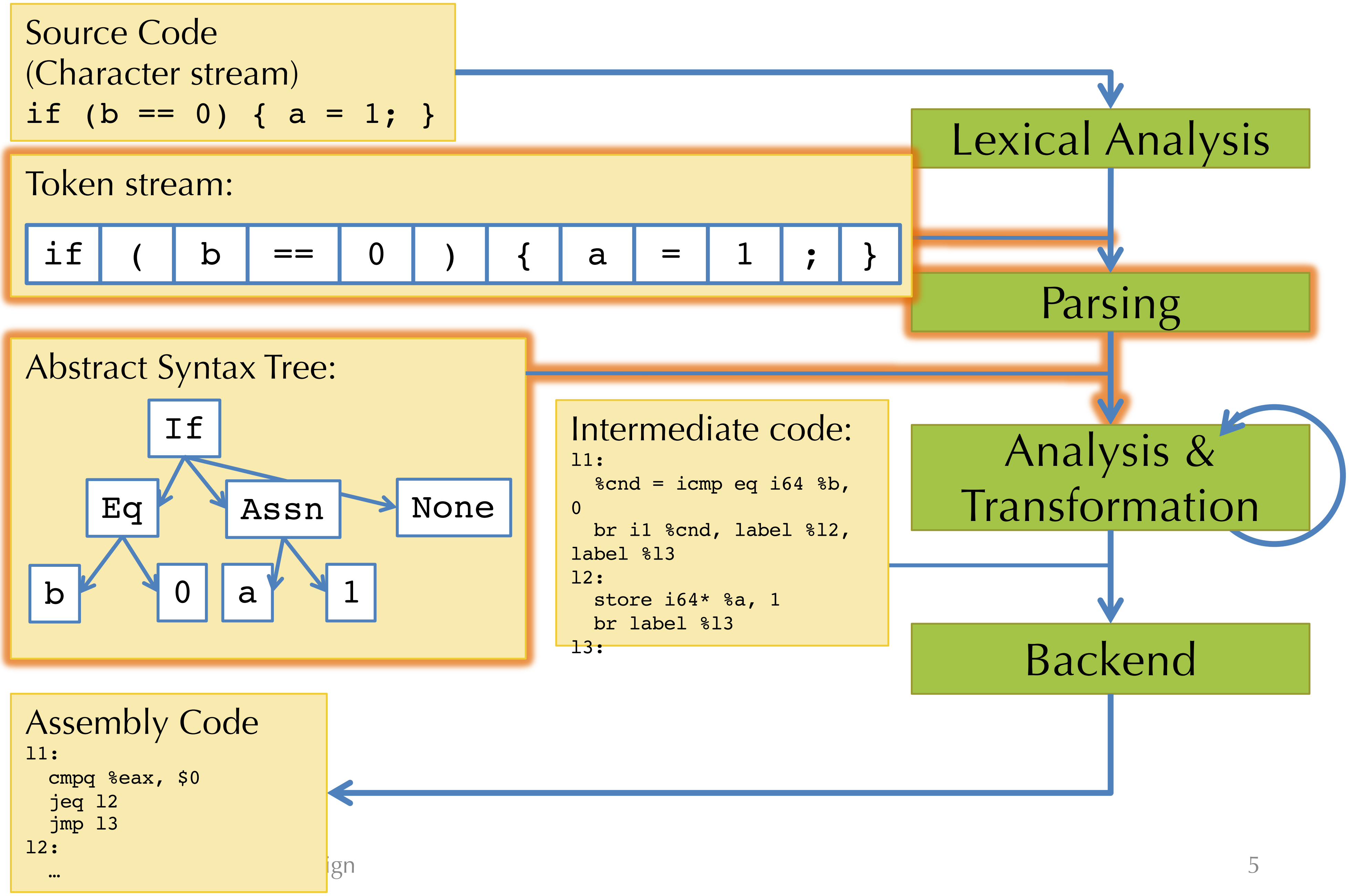
```
CMP ECX, 0
SETBZ EAX
```

Front End
(machine independent)

Middle End
(compiler dependent)

Back End
(machine dependent)

Today: Parsing



Source Code
(Character stream)

```
if (b == 0) { a = 1; }
```

Lexical Analysis

Token stream:

if	(	b	==	0	)	{	a	=	1	;	}

Parsing

Abstract Syntax Tree:

Analysis & Transformation

Intermediate code:

```
l1:
  %cnd = icmp eq i64 %b,
0
  br i1 %cnd, label %l2,
label %l3
l2:
  store i64* %a, 1
  br label %l3
l3:
```

Backend

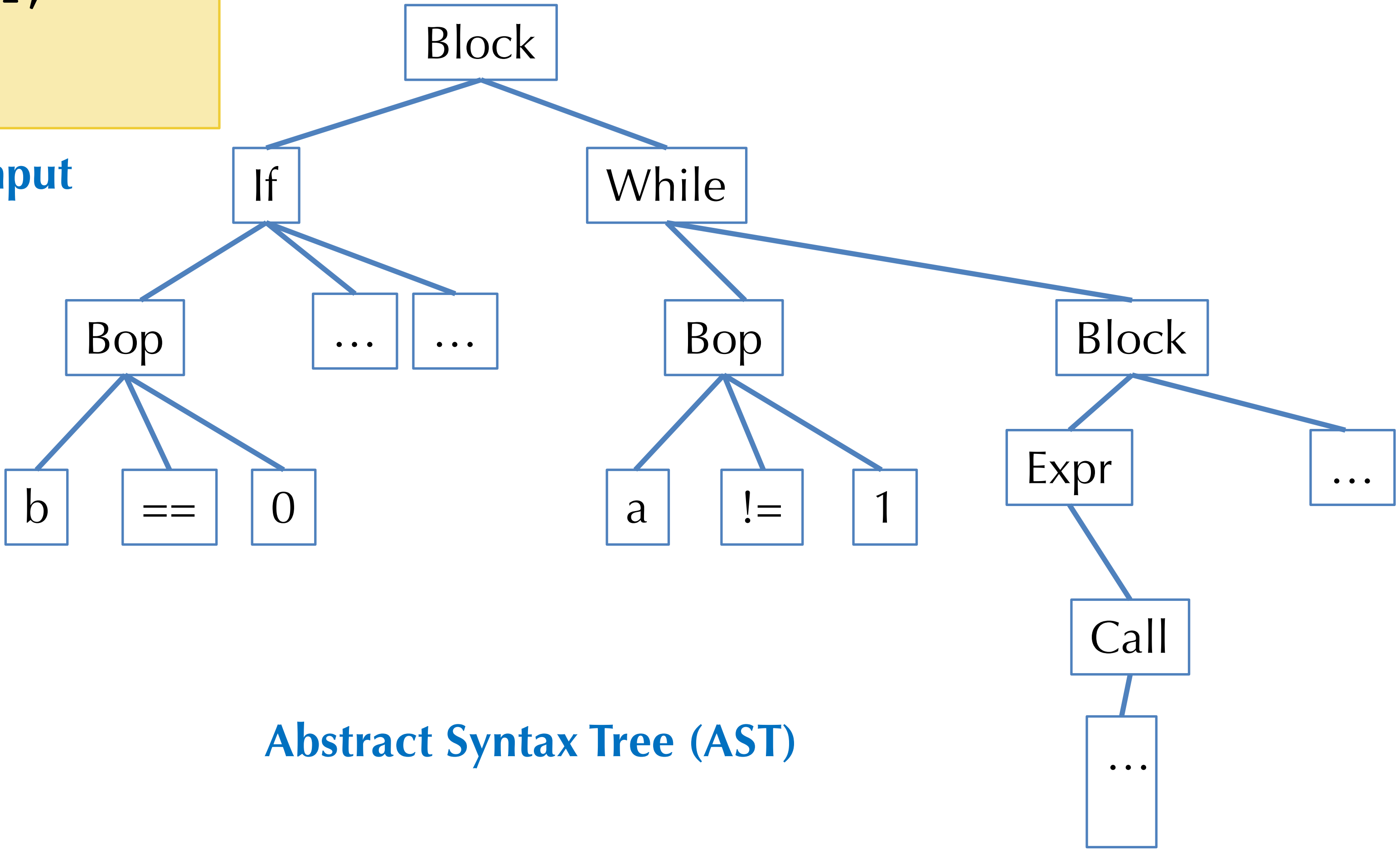
Assembly Code

```
l1:
  cmpq %eax, $0
  jeq l2
  jmp l3
l2:
  …
```

Parsing: Finding Syntactic Structure

```
{
  if (b == 0) a = b;
  while (a != 1) {
    print_int(a);
    a = a – 1;
  }
}
```

Source input



Abstract Syntax Tree (AST)

Syntactic Analysis (Parsing): Overview

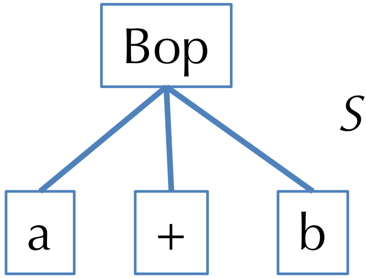
- Input: stream of tokens (generated by lexer)
- Output: abstract syntax tree

- Strategy
  - Parse the token stream to traverse the "concrete" syntax
  - During traversal, build a tree representing the "abstract" syntax

- Why abstract?

  Consider these three *different* concrete inputs

```
a + b
(a + ((b)))
((a) + (b))
```



Same abstract syntax tree

- Note: parsing doesn't check many things
  - Variable scoping, type agreement, initialization, etc.

Specifying Language Syntax

- First question:

  How to describe language syntax precisely and conveniently?

- Last time: we described tokens using regular expressions
  - Easy to implement, efficient DFA representation
  - So, why not use regular expressions over tokens to specify syntax?

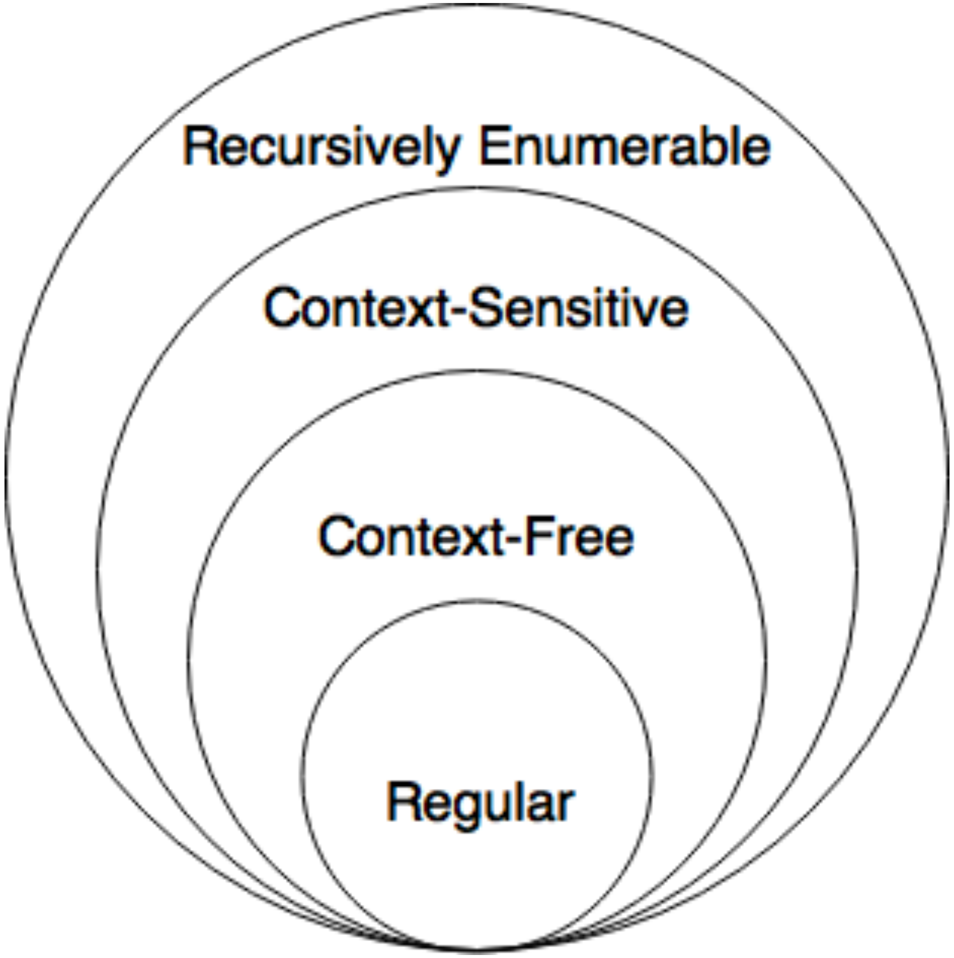
- Limits of regular expressions
  - DFA's have only finite # of states (i.e., finite memory)
  - So, DFA's can't "count"
  - E.g., consider the language of strings with balanced parentheses

$$(^k\,)^k$$

- So, we need more expressive power than DFA's

CONTEXT FREE GRAMMARS

Chomsky Hierarchy



Context-free Grammars (CFG)

- Here is a specification of the language of balanced parens

$$S \longmapsto (S)S$$

$$S \longmapsto \varepsilon$$

Note: Once again we have to take care to distinguish meta-language elements (e.g. "S" and "⟼") from object-language elements (e.g. "(").

- The definition is *recursive*: S mentions itself

- Idea: "derive" a string in the language starting from S and rewriting according to the rules
  - Example: $S \longmapsto (S)S \longmapsto ((S)S)S \longmapsto ((\varepsilon)S)S \longmapsto ((\varepsilon)S)\varepsilon \longmapsto ((\varepsilon)\varepsilon)\varepsilon = (())$

- We can replace the "nonterminal" S by its definition anywhere
- A CFG accepts a string iff there is a derivation from the start symbol

CFGs Mathematically

- A Context-free Grammar (CFG) consists of
  - A set of *terminals* (e.g., a lexical token, but how about ε?)
  - A set of *nonterminals* (e.g., S and other syntactic variables)
  - A designated nonterminal called the *start symbol*
  - A set of productions: LHS $\longmapsto$ RHS
    - LHS is a nonterminal
    - RHS is a *string* of terminals and nonterminals

- Example: The balanced parentheses language

$$S \longmapsto (S)S$$

$$S \longmapsto \varepsilon$$

- How many terminals? How many nonterminals? Productions?

Another Example: Sum Grammar

- A grammar that accepts parenthesized sums of numbers

$$\begin{array}{lllll} S & \longmapsto & E + S & \mid & E \\ E & \longmapsto & \text{number} & \mid & (\ S\ ) \end{array}$$

e.g.: (1 + 2 + (3 + 4)) + 5

- Note the vertical bar '|' is shorthand for multiple productions

$$\left.\begin{array}{l} S \longmapsto E + S \\ S \longmapsto E \\ E \longmapsto \text{number} \\ E \longmapsto (S) \end{array}\right\}$$

4 productions
2 nonterminals: S, E
4 terminals: (,), +, number
Start symbol: S

Derivations in CFGs

$S \longmapsto E + S \mid E$
$E \longmapsto \text{number} \mid (S)$

- Example: derive (1 + 2 + (3 + 4)) + 5
- $\underline{\mathbf{S}} \longmapsto \underline{\mathbf{E}} + S$

$\longmapsto (\underline{\mathbf{S}}) + S$
$\longmapsto (\underline{\mathbf{E}} + S) + S$
$\longmapsto (1 + \underline{\mathbf{S}}) + S$
$\longmapsto (1 + \underline{\mathbf{E}} + S) + S$
$\longmapsto (1 + 2 + \underline{\mathbf{S}}) + S$
$\longmapsto (1 + 2 + \underline{\mathbf{E}}) + S$
$\longmapsto (1 + 2 + (\underline{\mathbf{S}})) + S$
$\longmapsto (1 + 2 + (\underline{\mathbf{E}} + S)) + S$
$\longmapsto (1 + 2 + (3 + \underline{\mathbf{S}})) + S$
$\longmapsto (1 + 2 + (3 + \underline{\mathbf{E}})) + S$
$\longmapsto (1 + 2 + (3 + 4)) + \underline{\mathbf{S}}$
$\longmapsto (1 + 2 + (3 + 4)) + \underline{\mathbf{E}}$
$\longmapsto (1 + 2 + (3 + 4)) + 5$

For arbitrary strings α, β, γ and production rule $A \longmapsto \beta$
a single step of the derivation is

$$\alpha A \gamma \longmapsto \alpha \beta \gamma$$

(*substitute* β for an occurrence of A)

In general, there are many possible derivations for a given string

Note: underline indicates symbol being expanded

From Derivations to Parse Trees

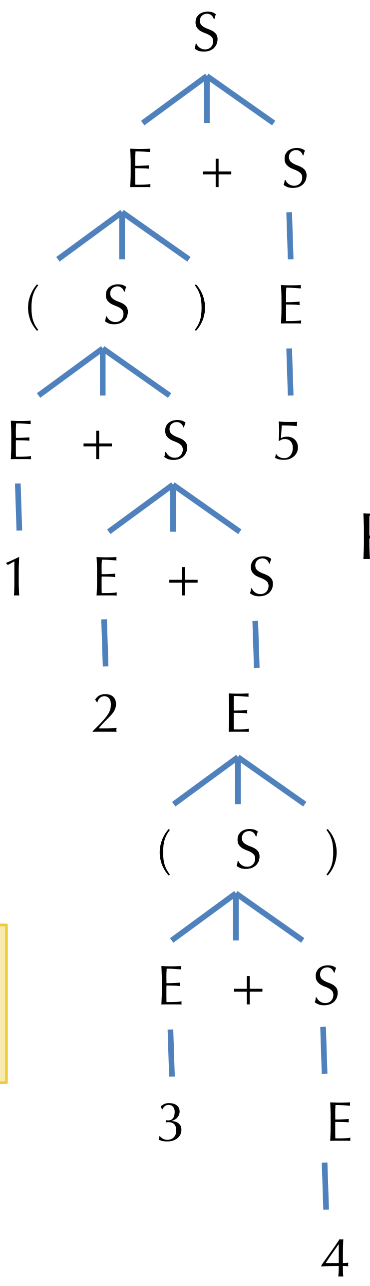
- Tree representation of a derivation
  - Leaves: terminals

  In-order traversal yields the input token sequence
  - Internal nodes: nonterminals

- No info. on the *order* of the derivation steps

(1 + 2 + (3 + 4)) + 5

S ⟼ E + S | E
E ⟼ number | (S)

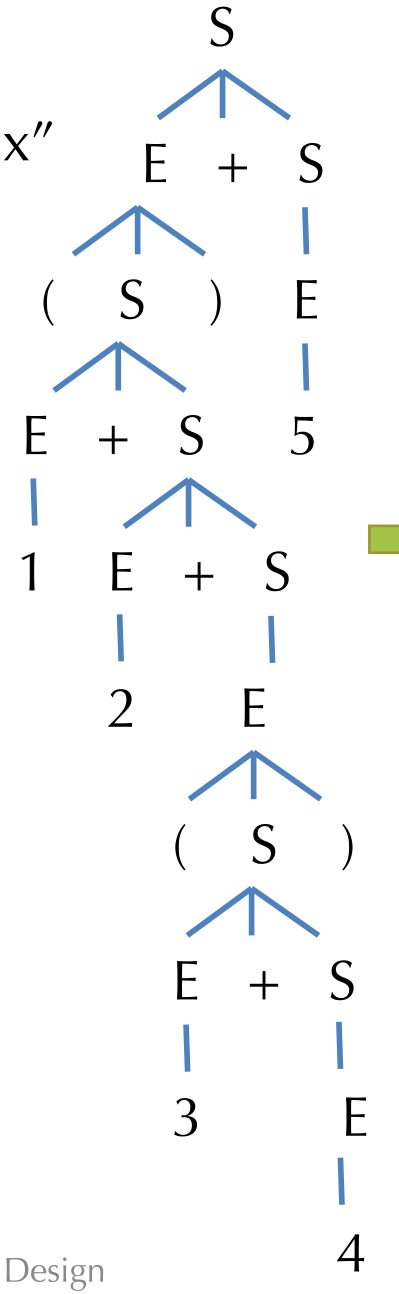


Parse Tree

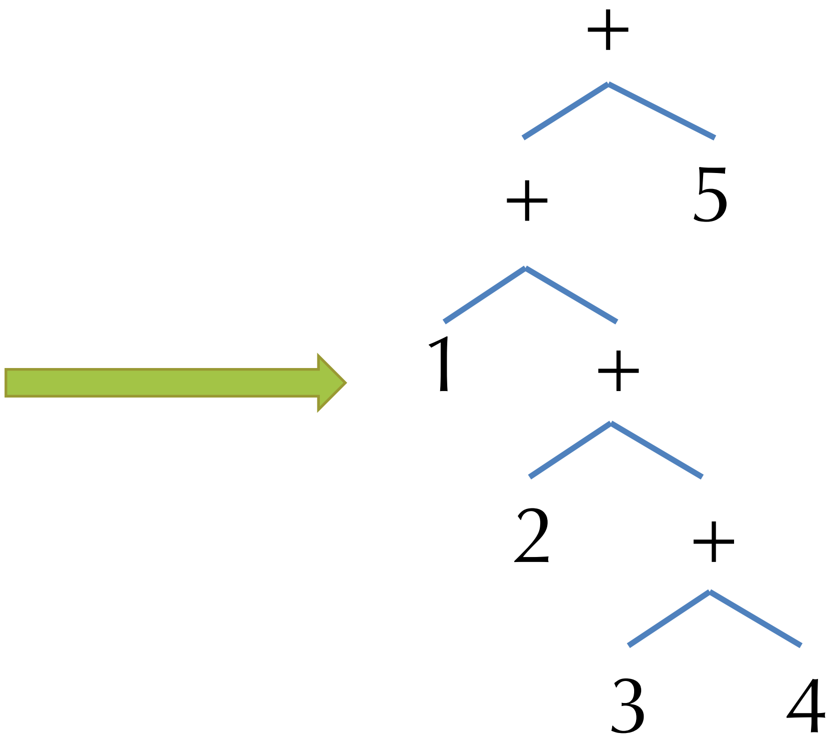
From Parse Trees to Abstract Syntax

- *Parse tree*

"concrete syntax"



- *Abstract syntax tree* (AST)



- Hides, or *abstracts*, unneeded information

Derivation Orders

- Productions of the grammar can be applied in any order
- There are two standard orders
  - *Leftmost derivation*

    Find the left-most nonterminal and apply a production to it
  - *Rightmost derivation*

    Find the right-most nonterminal and apply a production there

- Both strategies (and any other) yield the same parse tree!
  - Parse tree doesn't contain the information about what order the productions were applied

Example: Left- and rightmost derivations

- Leftmost derivation
- $\underline{\mathbf{S}} \longmapsto \underline{\mathbf{E}} + S$

$\longmapsto (\underline{\mathbf{S}}) + S$

$\longmapsto (\underline{\mathbf{E}} + S) + S$

$\longmapsto (1 + \underline{\mathbf{S}}) + S$

$\longmapsto (1 + \underline{\mathbf{E}} + S) + S$

$\longmapsto (1 + 2 + \underline{\mathbf{S}}) + S$

$\longmapsto (1 + 2 + \underline{\mathbf{E}}) + S$

$\longmapsto (1 + 2 + (\underline{\mathbf{S}})) + S$

$\longmapsto (1 + 2 + (\underline{\mathbf{E}} + S)) + S$

$\longmapsto (1 + 2 + (3 + \underline{\mathbf{S}})) + S$

$\longmapsto (1 + 2 + (3 + \underline{\mathbf{E}})) + S$

$\longmapsto (1 + 2 + (3 + 4)) + \underline{\mathbf{S}}$

$\longmapsto (1 + 2 + (3 + 4)) + \underline{\mathbf{E}}$

$\longmapsto (1 + 2 + (3 + 4)) + 5$

Rightmost derivation

$\underline{\mathbf{S}} \longmapsto E + \underline{\mathbf{S}}$

$\longmapsto E + \underline{\mathbf{E}}$

$\longmapsto \underline{\mathbf{E}} + 5$

$\longmapsto (\underline{\mathbf{S}}) + 5$

$\longmapsto (E + \underline{\mathbf{S}}) + 5$

$\longmapsto (E + E + \underline{\mathbf{S}}) + 5$

$\longmapsto (E + E + \underline{\mathbf{E}}) + 5$

$\longmapsto (E + E + (\underline{\mathbf{S}})) + 5$

$\longmapsto (E + E + (E + \underline{\mathbf{S}})) + 5$

$\longmapsto (E + E + (E + \underline{\mathbf{E}})) + 5$

$\longmapsto (E + E + (\underline{\mathbf{E}} + 4)) + 5$

$\longmapsto (E + \underline{\mathbf{E}} + (3 + 4)) + 5$

$\longmapsto (\underline{\mathbf{E}} + 2 + (3 + 4)) + 5$

$\longmapsto (1 + 2 + (3 + 4)) + 5$

$S \longmapsto E + S \mid E$

$E \longmapsto \text{number} \mid (S)$

Loops and Termination

- Some care is needed when defining CFGs …

- Consider

$$\begin{array}{lcl} S & \longmapsto & E \\ E & \longmapsto & S \end{array}$$

  - This grammar has nonterminal definitions that are “nonproductive” (i.e. they don’t mention any terminal symbols)
  - There is no finite derivation starting from S, so the language is empty

Loops and Termination

- Some care is needed when defining CFGs …
- Consider

$$S \longmapsto \quad (\; S \;)$$

  - This grammar is productive, but again there is no finite derivation starting from S, so the language is empty

Loops and Termination

- Some care is needed when defining CFGs …
- Easily generalize these examples to a “chain” of many nonterminals, which can be harder to find in a large grammar
- Upshot: be aware of “vacuously empty” CFG grammars
  - Every nonterminal should eventually rewrite to an alternative that contains only terminal symbols

Associativity, ambiguity, and precedence.

GRAMMARS FOR PROGRAMMING LANGUAGES

Associativity

Consider the input: $1 + 2 + 3$

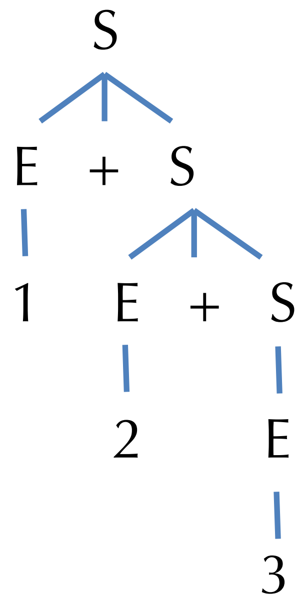
$S \mapsto E + S \mid E$
$E \mapsto \text{number} \mid (S)$

Leftmost derivation

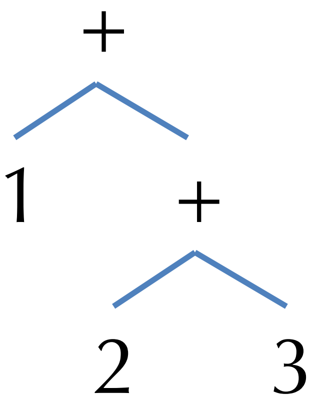
$\underline{\mathbf{S}} \mapsto \underline{\mathbf{E}} + S$
$\mapsto 1 + \underline{\mathbf{S}}$
$\mapsto 1 + \underline{\mathbf{E}} + S$
$\mapsto 1 + 2 + \underline{\mathbf{S}}$
$\mapsto 1 + 2 + \underline{\mathbf{E}}$
$\mapsto 1 + 2 + 3$

Rightmost derivation

$\underline{\mathbf{S}} \mapsto E + \underline{\mathbf{S}}$
$\mapsto E + E + \underline{\mathbf{S}}$
$\mapsto E + E + \underline{\mathbf{E}}$
$\mapsto E + \underline{\mathbf{E}} + 3$
$\mapsto \underline{\mathbf{E}} + 2 + 3$
$\mapsto 1 + 2 + 3$



Parse Tree



AST

Associativity

- This grammar makes '+' *right associative*
  - AST is the same for both 1 + 2 + 3 and 1 + (2 + 3)

- Note that the grammar is *right recursive*

$$
\begin{aligned}
S &\longmapsto E + S \mid E \\
E &\longmapsto \text{number} \mid (S)
\end{aligned}
$$

- How would you make '+' left associative?
- What are the trees for "1 + 2 + 3"?

Ambiguity

- Consider this grammar

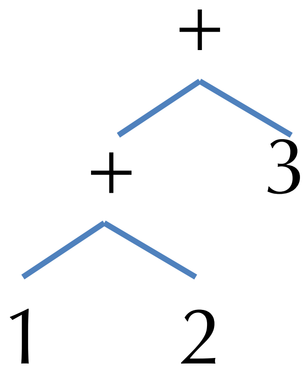
$$S \longmapsto S + S \mid (S) \mid \text{number}$$

- Claim: It accepts the *same* set of strings as the previous one
- What's the difference?
- Consider these *two* leftmost derivations

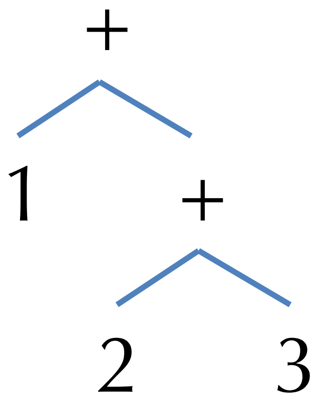
$$\underline{\mathbf{S}} \longmapsto \underline{\mathbf{S}} + S \longmapsto 1 + \underline{\mathbf{S}} \longmapsto 1 + \underline{\mathbf{S}} + S \longmapsto 1 + 2 + \underline{\mathbf{S}} \longmapsto 1 + 2 + 3$$

$$\underline{\mathbf{S}} \longmapsto \underline{\mathbf{S}} + S \longmapsto \underline{\mathbf{S}} + S + S \longmapsto 1 + \underline{\mathbf{S}} + S \longmapsto 1 + 2 + \underline{\mathbf{S}} \longmapsto 1 + 2 + 3$$

- One derivation gives left associativity, the other gives right associativity to '+'
  - Which is which?



AST 1



AST 2

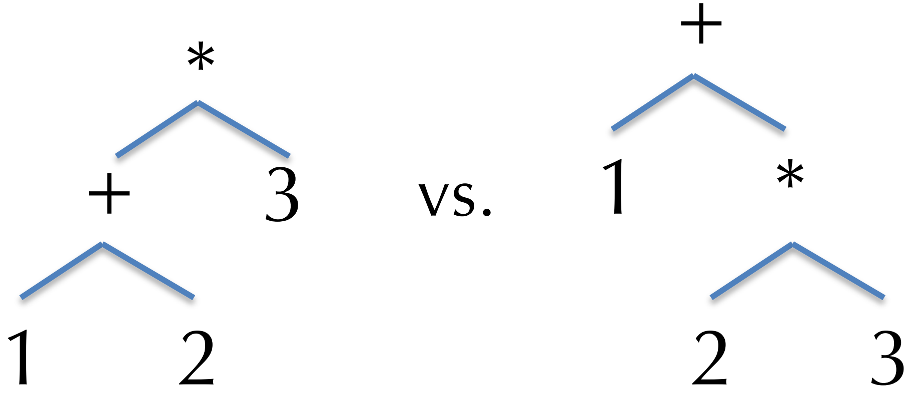
Why do we care about ambiguity?

- The '+' operation is associative, so it doesn't matter which tree we pick. Mathematically, $x + (y + z) = (x + y) + z$
  - But, some operations are non-associative. Examples?
  - Some operations are only left (or right) associative. Examples?

- Moreover, if there are multiple operations, ambiguity in the grammar leads to ambiguity in their *precedence*
- Consider

S ⟼ S + S | S * S | (S) | number

- Input: 1 + 2 * 3
  - One parse = (1 + 2) * 3 = 9
  - The other = 1 + (2 * 3) = 7



Eliminating Ambiguity

- We can often eliminate ambiguity by adding nonterminals and allowing recursion only on the left (or right)
- Higher-precedence operators go *farther* from the start symbol
- Example

$$S \longmapsto S + S \mid S * S \mid (S) \mid \text{number}$$

- To disambiguate
  - Decide (following math) to make ‘*’ higher precedence than ‘+’
  - Make ‘+’ left associative
  - Make ‘*’ right associative (fix?)
- Note: S_2 corresponds to ‘atomic’ expressions

$$\begin{array}{lcll} S_0 & \longmapsto & S_0 + S_1 & \mid S_1 \\ S_1 & \longmapsto & S_2 * S_1 & \mid S_2 \\ S_2 & \longmapsto & \text{number} & \mid (S_0) \end{array}$$

Context Free Grammars: Summary

- CFGs allow concise specifications of prog. languages
  - An unambiguous CFG specifies how to parse: convert a token stream to a (parse tree)
  - Ambiguity can (often) be removed by encoding precedence and associativity in the grammar

- Even with an unambiguous CFG, there may be more than one derivation
  - Though all derivations correspond to the same abstract syntax tree

- Still to come: finding a derivation
  - But first: menhir

parser.mly, lexer.mll, range.ml, ast.ml, main.ml

DEMO: BOOLEAN LOGIC