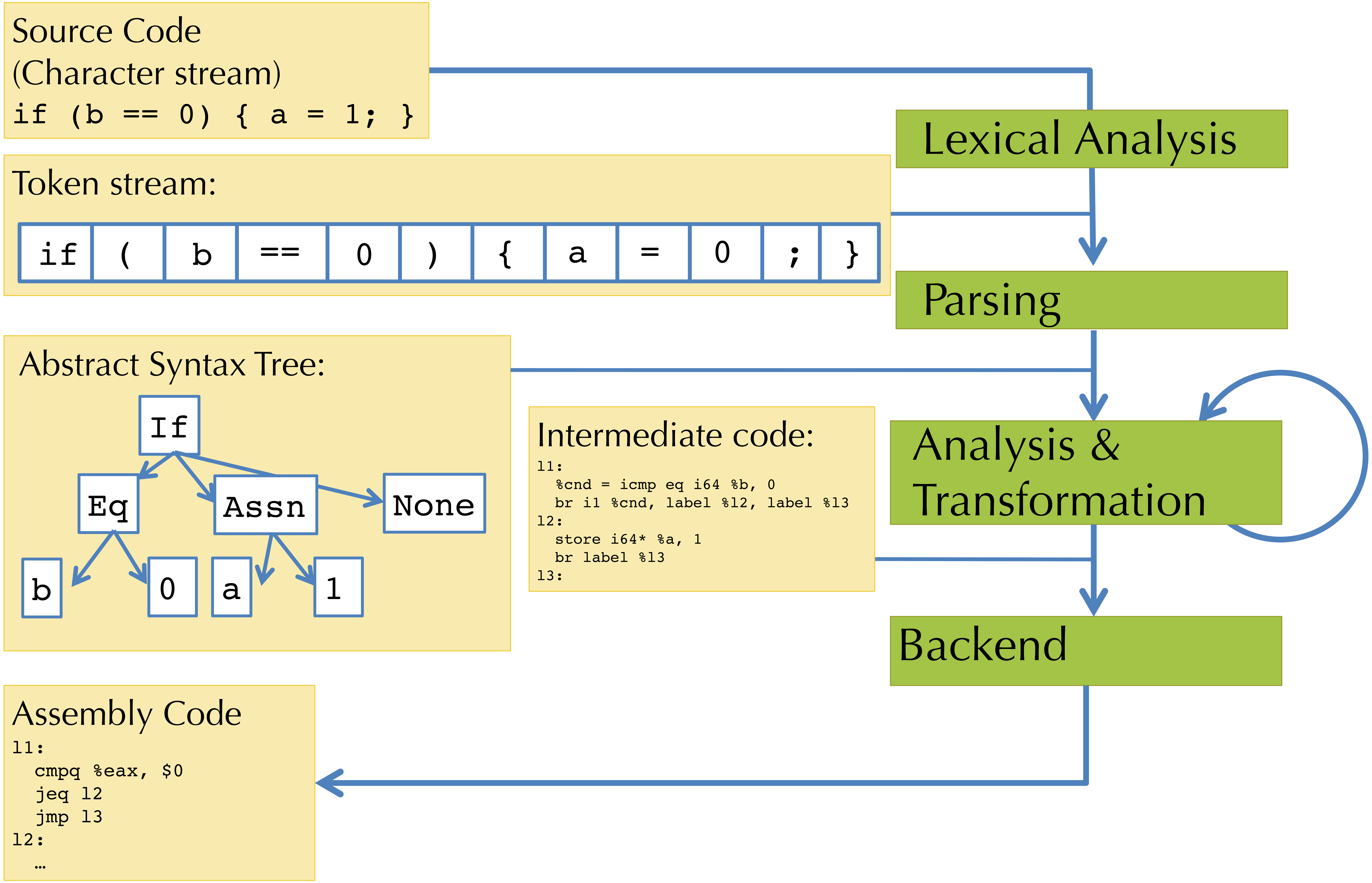
YSC4230: Programming Language Design and Implementation

Week 6: Lexing and Parsing

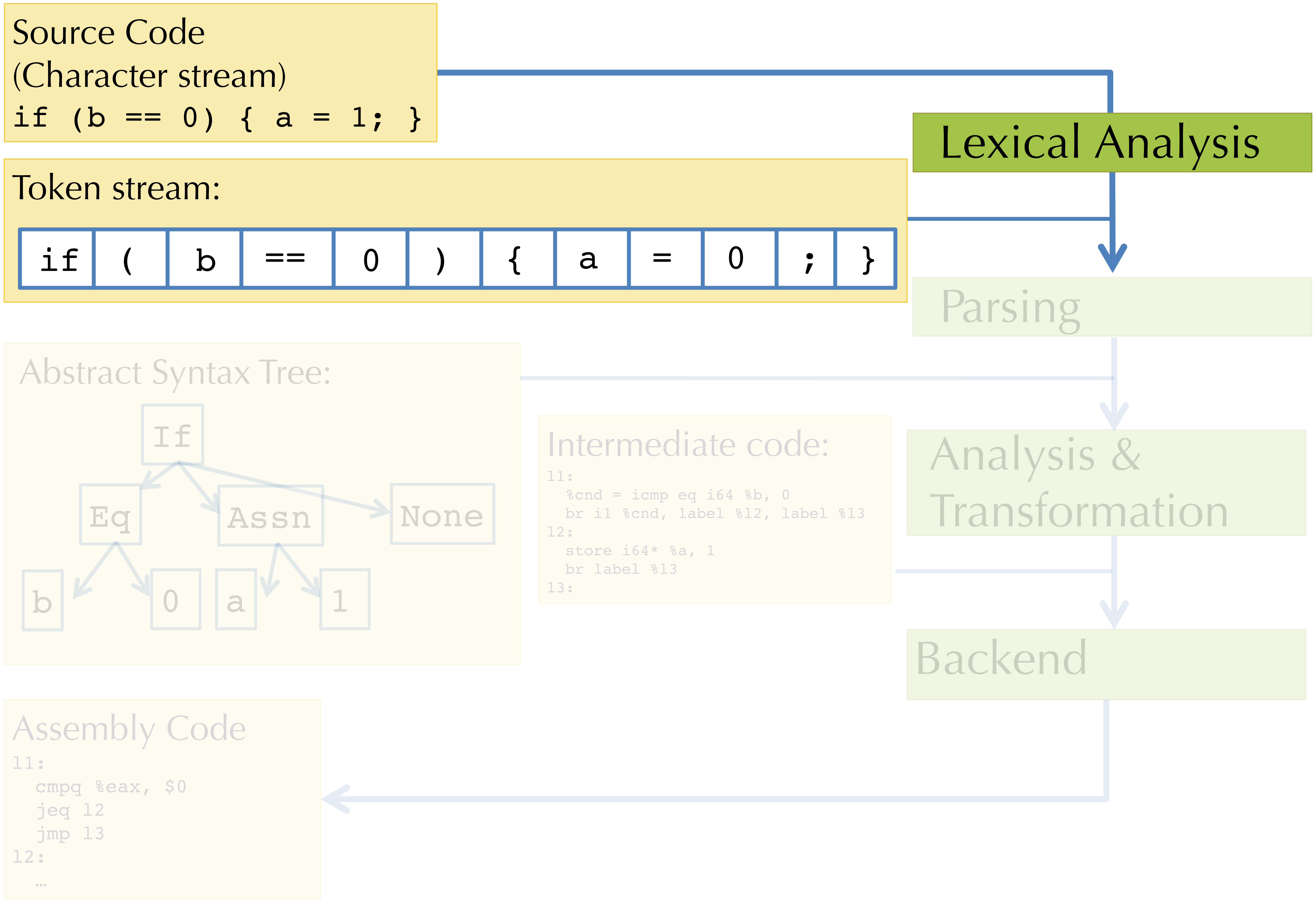
Ilya Sergey

ilya.sergey@yale-nus.edu.sg

Compilation in a Nutshell



Last Week: Lexing



Regular Expressions

- Regular expressions precisely describe sets of strings.

- A regular expression R has one of the following forms:
  - ε — Epsilon stands for the empty string
  - `‘a’` — An ordinary character stands for itself
  - $R_1 \mid R_2$ — Alternatives, stands for choice of R_1 or R_2
  - R_1R_2 — Concatenation, stands for R_1 followed by R_2
  - `R*` — Kleene star, stands for *zero or more* repetitions of `R`

- *Useful extensions:*
  - `“foo”` — Strings, equivalent to `'f''o''o'`
  - `R+` — One or more repetitions of `R`, equivalent to `RR*`
  - `R?` — Zero or one occurrences of `R`, equivalent to `(ε|R)`
  - `['a'-'z']` — One of `a` or `b` or `c` or … `z`, equivalent to `(a|b|…|z)`
  - `[^'0'-'9']` — Any character except `0` through `9`
  - `R as x` — Name the string matched by `R` as `x`

Example Regular Expressions

- Recognise the keyword "if": `"if"`
- Recognise a digit: `['0'-'9']`
- Recognise an integer literal: `'-'?['0'-'9']+`
- Recognise an identifier:
  `(['a'-'z']|['A'-'Z'])(['0'-'9']|'_'|['a'-'z']|['A'-'Z'])*`

- In practice, it's useful to be able to *name* regular expressions:

```
let lowercase = ['a'-'z']
let uppercase = ['A'-'Z']
let character = uppercase | lowercase
```

How to Match?

- Consider the input string: `ifx = 0`
  - Could lex as: `if` | `x` | `=` | `0` or as: `ifx` | `=` | `0`

- Regular expressions alone are *ambiguous*, need a rule to choose between the options above
- Most languages choose “longest match”
  - So the 2nd option above will be picked
  - Note that only the first option is “correct” for parsing purposes

- Conflicts: arise due to two tokens whose regular expressions have a shared prefix
  - Ties broken by giving some matches **higher priority**
  - Example: keywords have priority over identifiers
  - Usually specified by order the rules appear in the lex input file

Lexer Generators

- Reads a list of regular expressions: $R_1, \ldots, R_n$, one per token.
- Each token has an attached "action" A_i
  (just a piece of code to run when the regular expression is matched)

```
rule token = parse
| '-'?digit+                          { Int (Int32.of_string (lexeme lexbuf)) }
| '+'                                 { PLUS }
| 'if'                                { IF }
| character (digit|character|'_')*    { Ident (lexeme lexbuf) }
| whitespace+                         { token lexbuf }
```

token
regular expressions

actions

- Generates scanning code that:
  1. Decides whether the input is of the form $(R_1|\ldots|R_n)^*$
  2. Whenever the scanner matches a (longest) token, it runs the associated action

Implementation Strategies

- Most Tools: lex, ocamllex, flex, etc.:
  - Table-based
  - Deterministic Finite Automata (DFA)
  - Goal: Efficient, compact representation, high performance

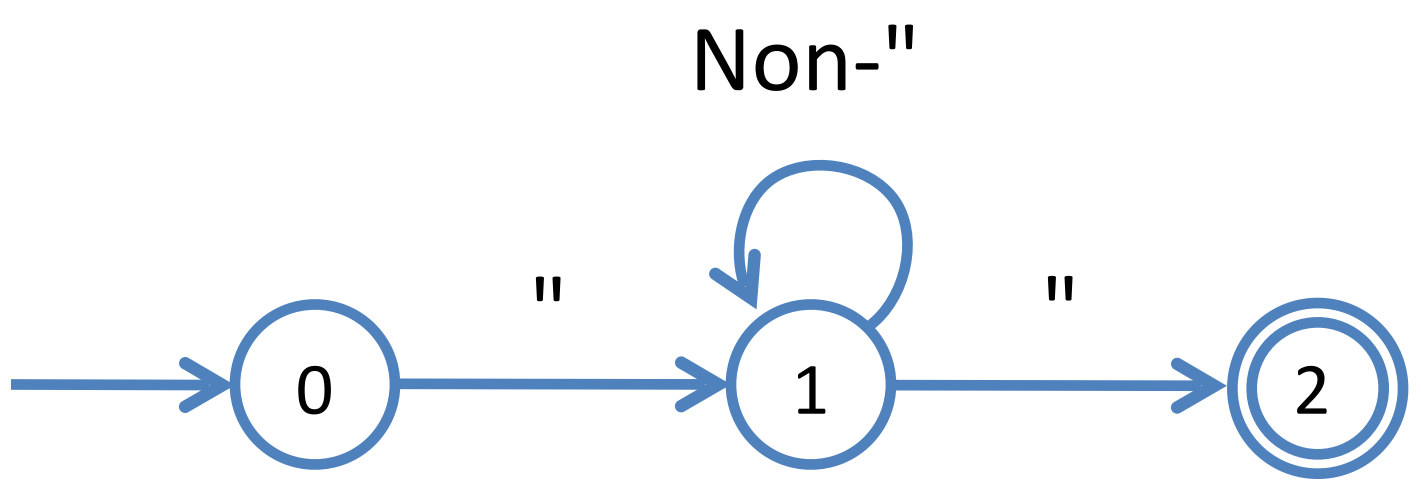
- Other approaches:
  - Brzozowski derivatives
  - Idea: directly manipulate the (abstract syntax of) the regular expression
  - Compute partial "derivatives"
    - Regular expression that is "left-over" after seeing the next character
  - Elegant, purely functional, implementation (very cool!)
  - See "Regular-expression derivatives re-examined" (2009) by Owens et al.

Finite Automata

- Consider the regular expression: ‘"’ [^ ’"’] * ’"’
- An automaton (DFA) can be represented as:
  - A transition table:

	"	Non-"
0	1	ERROR
1	2	1
2	ERROR	ERROR

  - A graph:

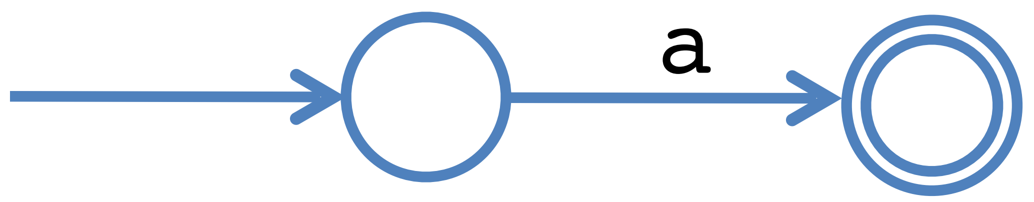


RE to Finite Automaton?

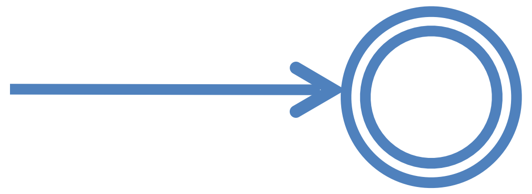
- Can we build a finite automaton for every regular expression?
  - Yes! (But the full theory is outside of the scope of this module)

- Strategy: consider every possible regular expression
  (by induction on the structure of the regular expressions):

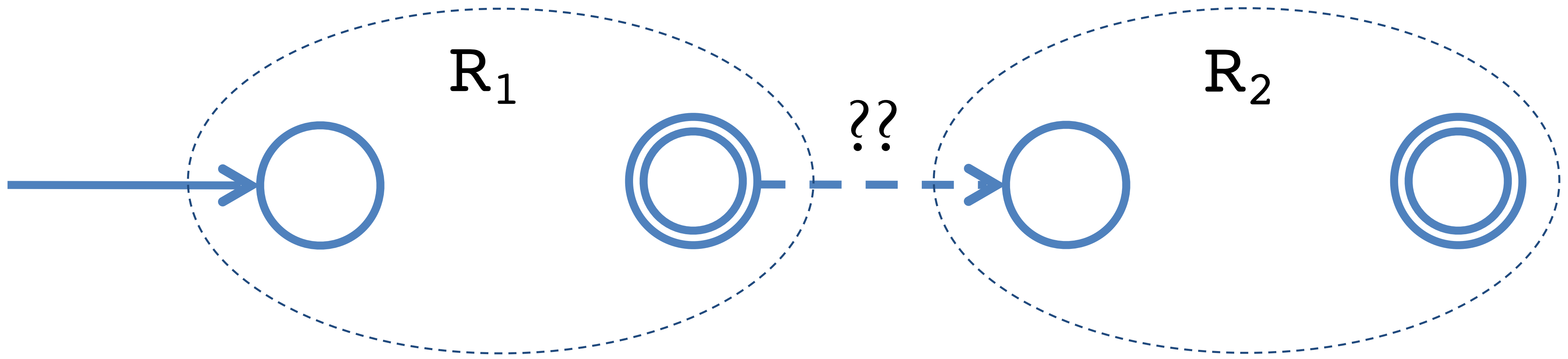
'a'



ε



R_1R_2

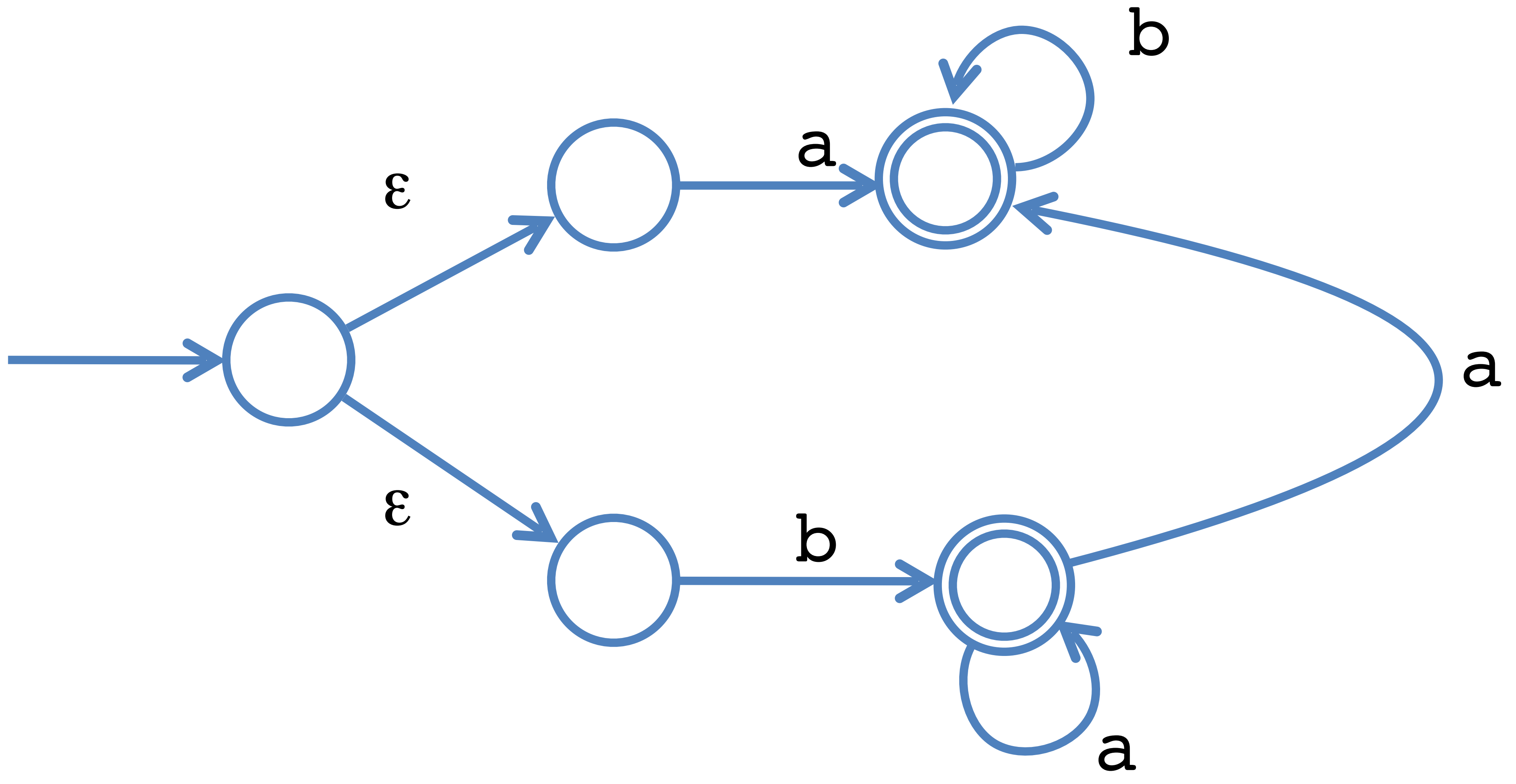


What about?

$R_1 | R_2$

Nondeterministic Finite Automata

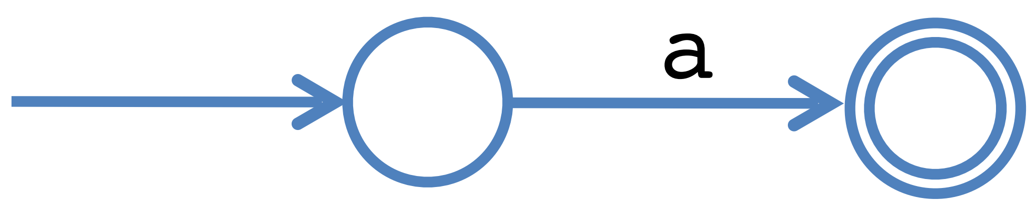
- A finite set of states, a start state, and accepting state(s)
- Transition arrows connecting states
  - Labeled by input symbols
  - Or ε (which does not consume input)
- *Nondeterministic*: two arrows leaving the same state may have the same label



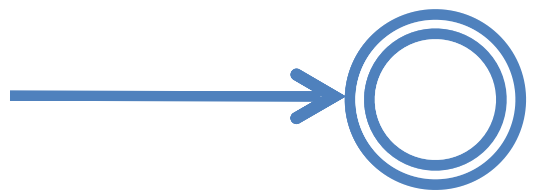
RE to NFA?

- Converting regular expressions to NFAs is easy.
- Assume each NFA has one start state, unique accept state

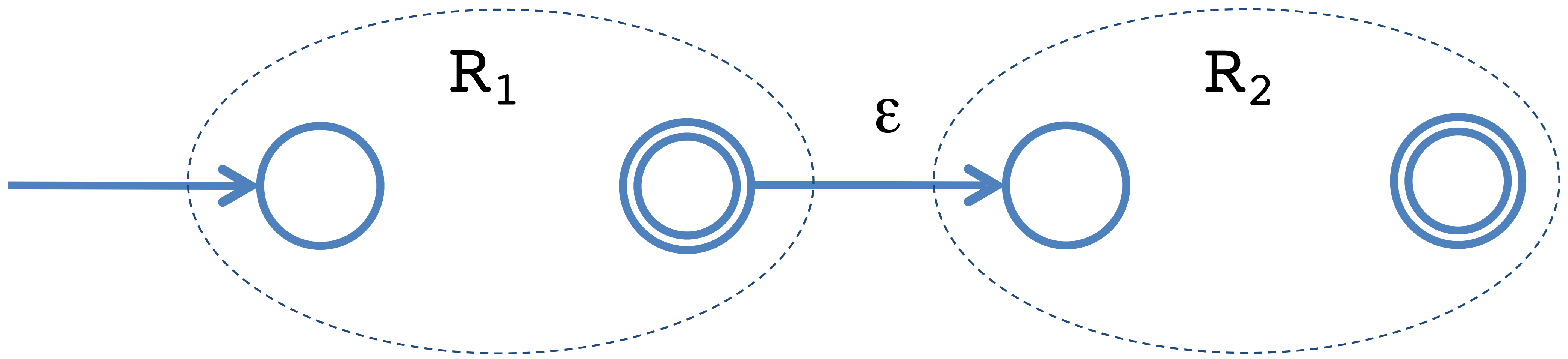
`'a'`



ε



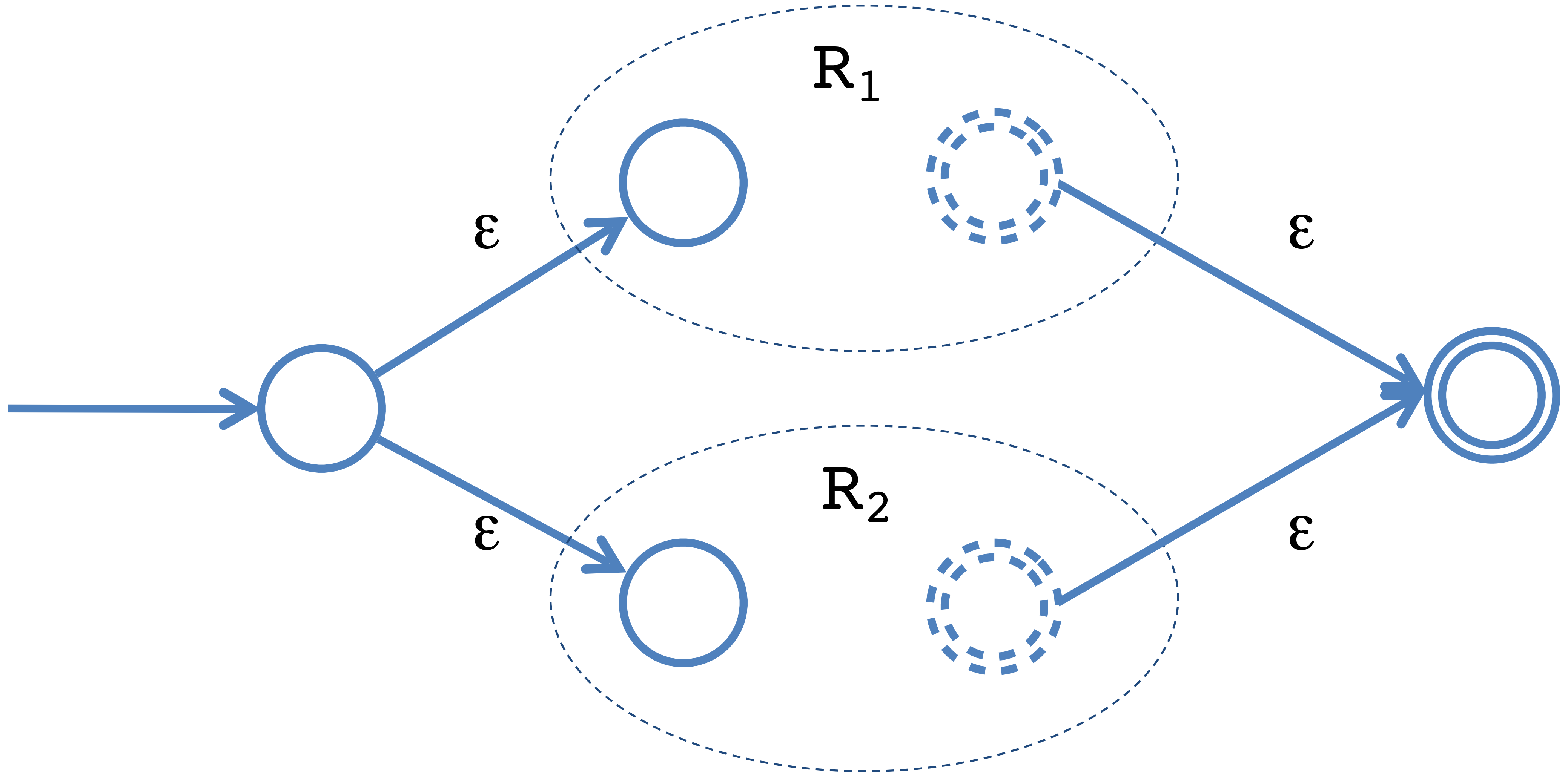
R_1R_2



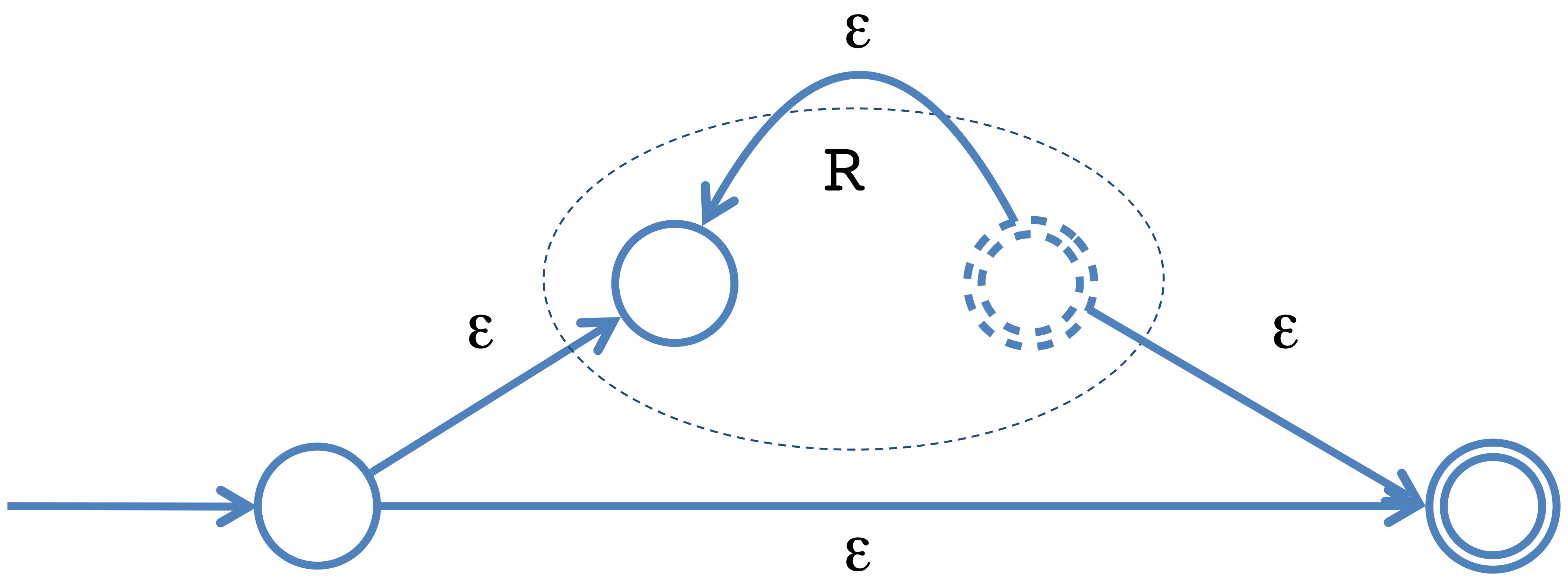
RE to NFA (cont'd)

- Alternatives and Kleene star are easy with NFAs

$R_1 | R_2$



$R*$



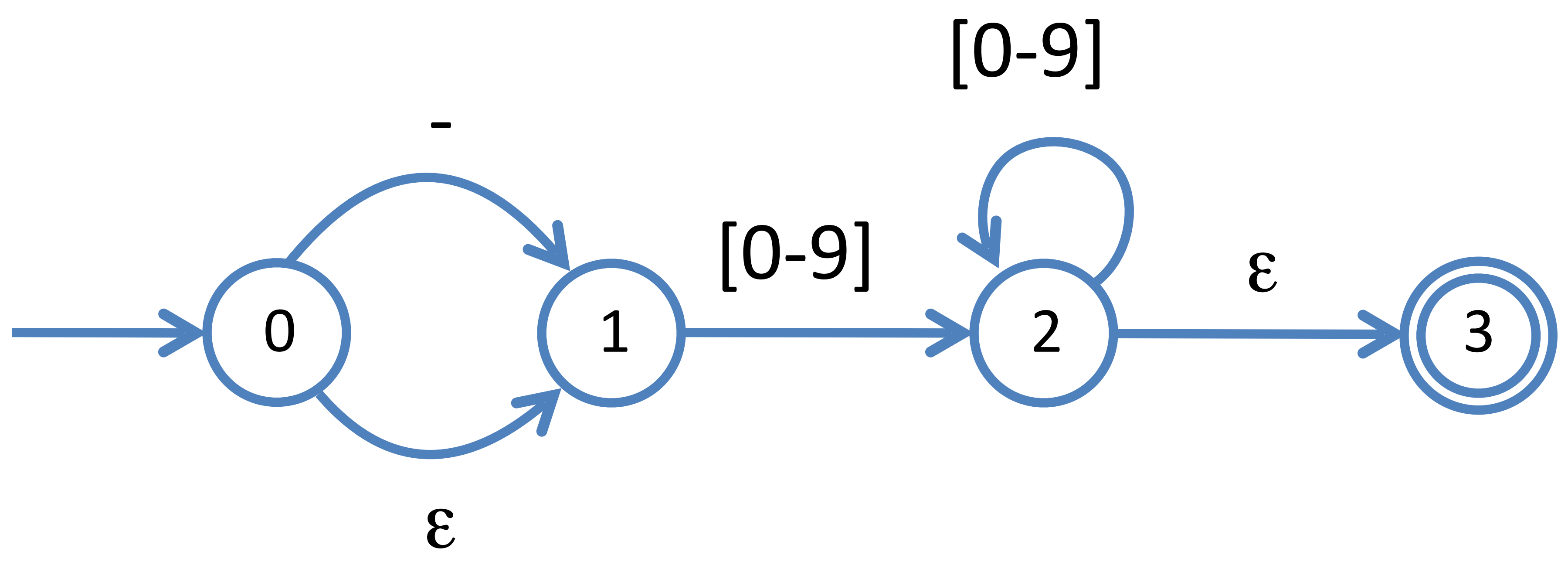
DFA versus NFA

- DFA:
  - Action of the automaton for each input is fully determined
  - Automaton accepts if the input is consumed upon reaching an accepting state
  - Obvious table-based implementation

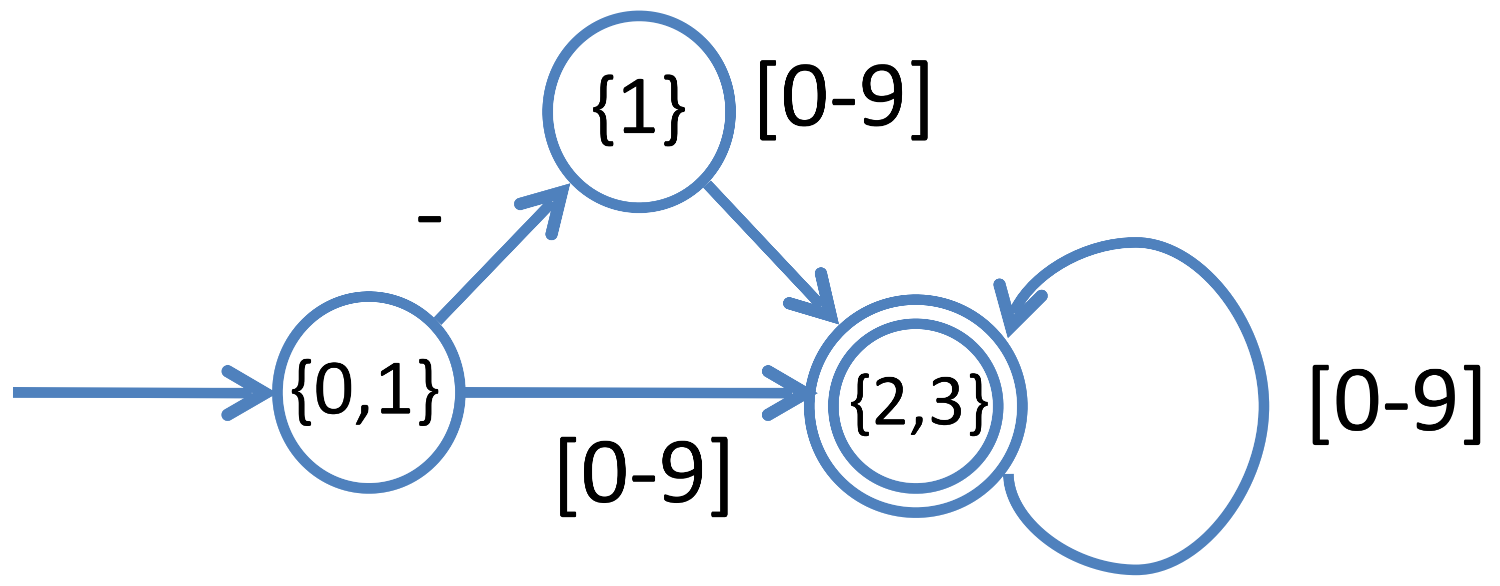
- NFA:
  - Automaton potentially *has a choice* at every step
  - Automaton accepts an input string if there *exists* a way to reach an accepting state
  - Less obvious how to implement **efficiently**

NFA to DFA conversion (Intuition)

- Idea: Run all possible executions of the NFA "in parallel"
- Keep track of a set of possible states: "finite fingers"
- Consider: `-?[0-9]+`
- NFA representation:



- DFA representation:



Summary of Lexer Generator Behavior

- Take each regular expression R_i and it's action A_i
- Compute the NFA formed by $(R_1 \mid R_2 \mid \ldots \mid R_n)$
  - Remember the actions associated with the accepting states of the R_i
- Compute the DFA for this big NFA
  - There may be multiple accept states (why?)
  - A single accept state may correspond to one or more actions (why?)
- Compute the minimal equivalent DFA
  - There is a standard algorithm due to Myhill & Nerode
- Produce the transition table
- Implement longest match:
  - Start from initial state
  - Follow transitions, remember last accept state entered (if any)
  - Accept input until no transition is possible (i.e. next state is "ERROR")
  - Perform the highest-priority action associated with the last accept state; if no accept state there is a lexing error

Lexer Generators in Practice

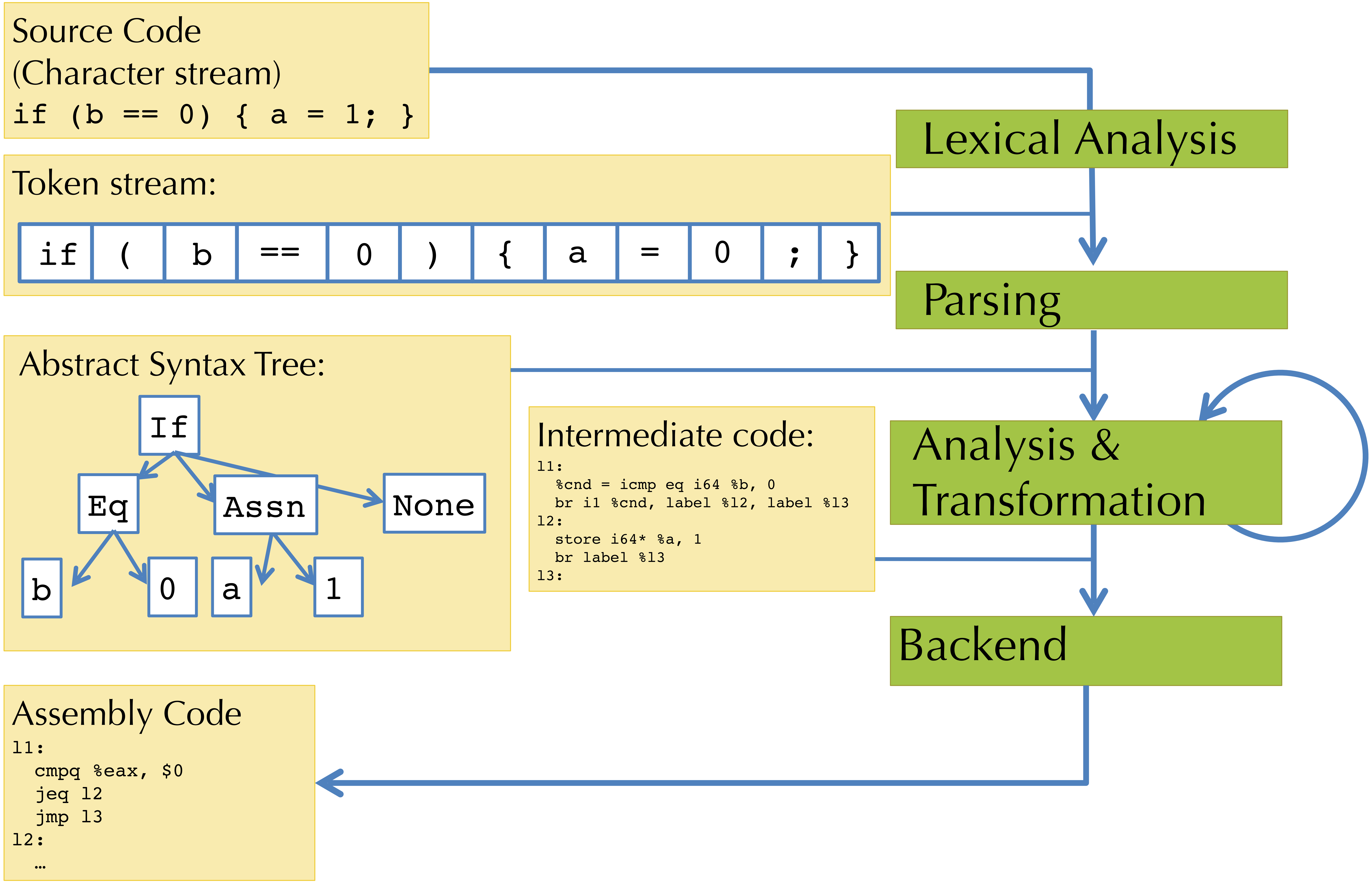
- Many existing implementations: lex, Flex, Jlex, ocamllex, …
  - For example ocamllex program
    - see lexlex.mll, olex.mll, piglatin.mll

- Error reporting:
  - Associate line number/character position with tokens
  - Use a rule to recognise '\n' and increment the line number
  - The lexer generator itself usually provides character position info.

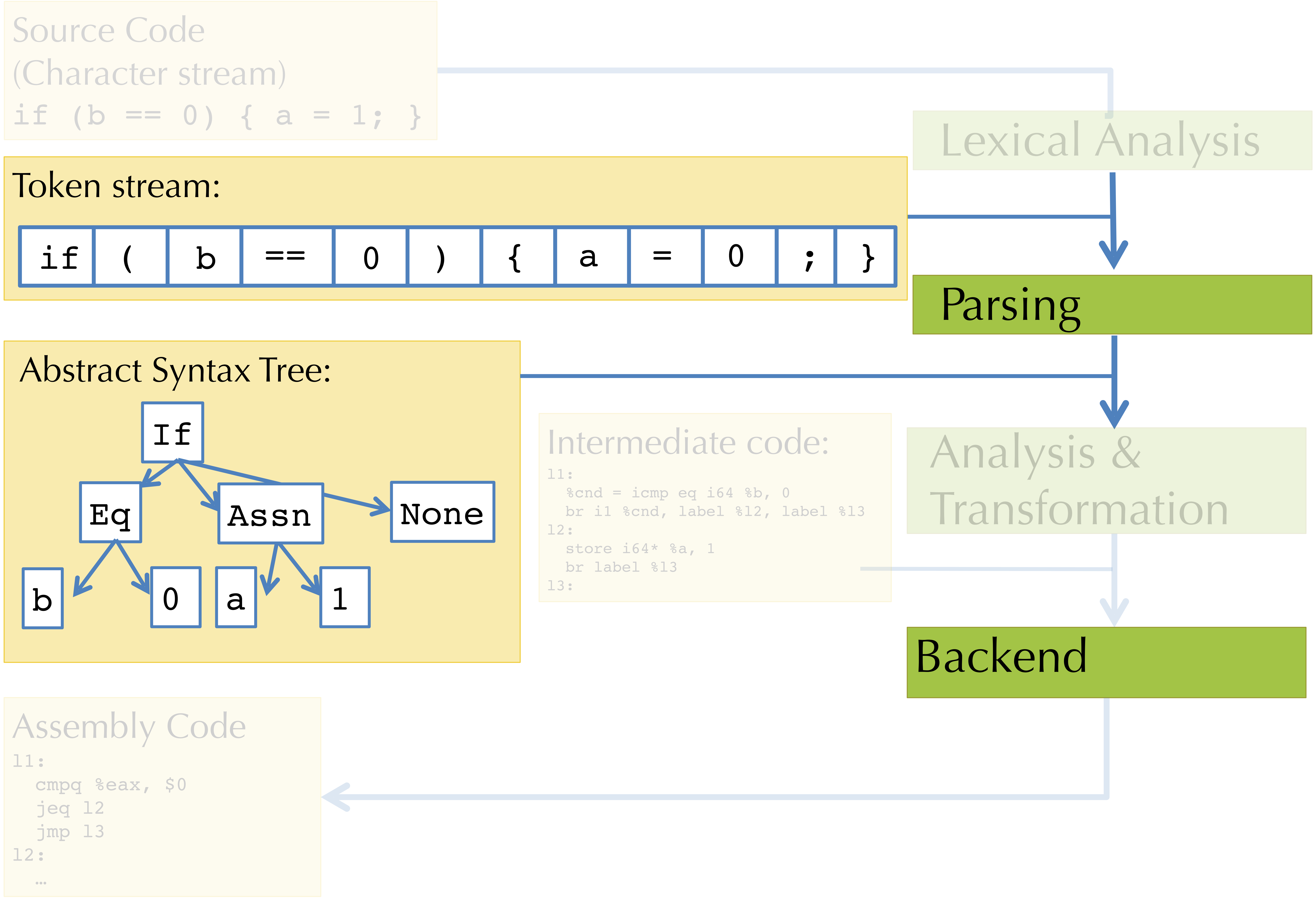
- Sometimes useful to treat comments specially
  - Nested comments: keep track of nesting depth

- Lexer generators are usually designed to work closely with parser generators…

Compilation in a Nutshell



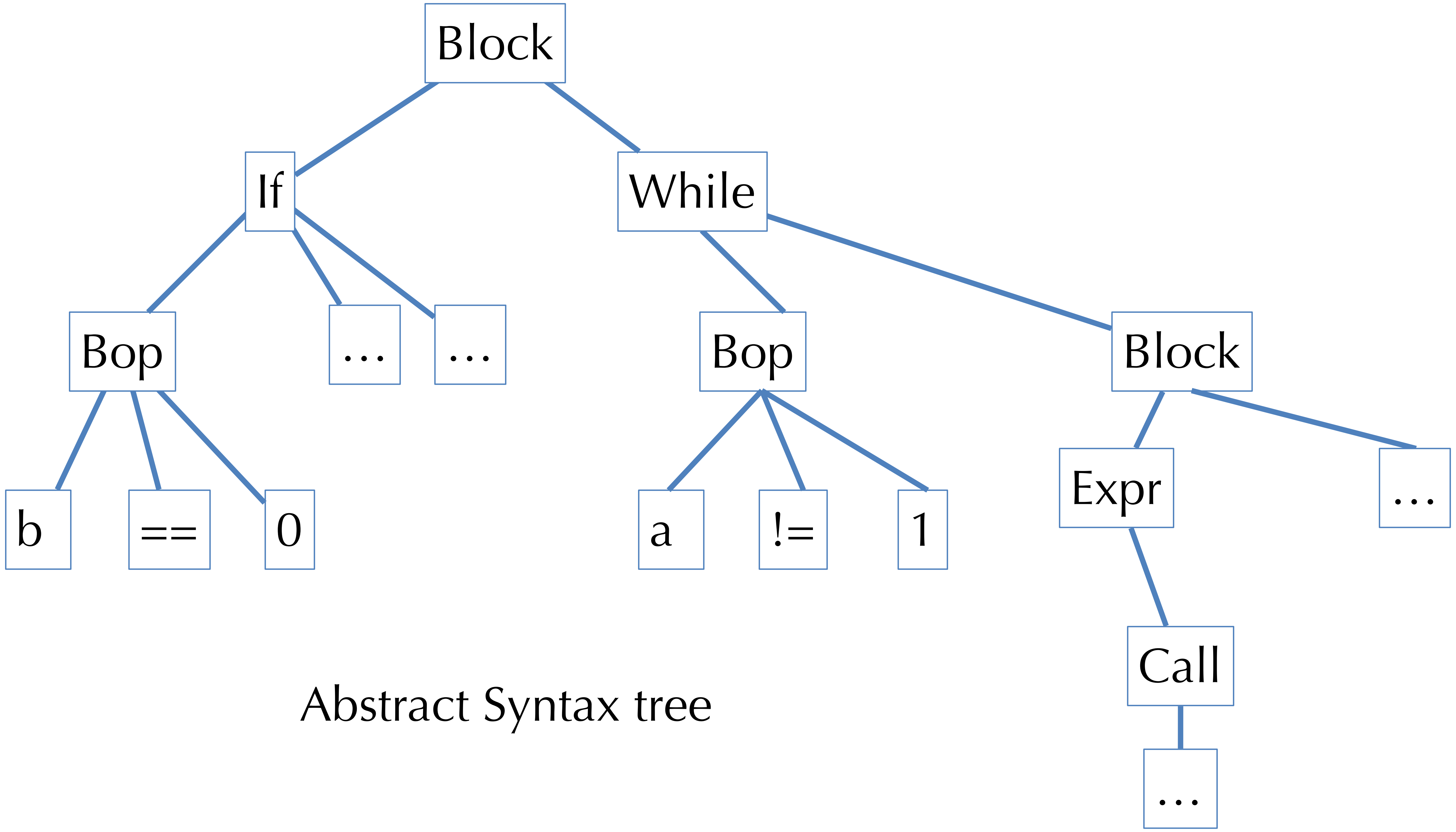
This week: Parsing



Parsing: Finding Syntactic Structure

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

Source input



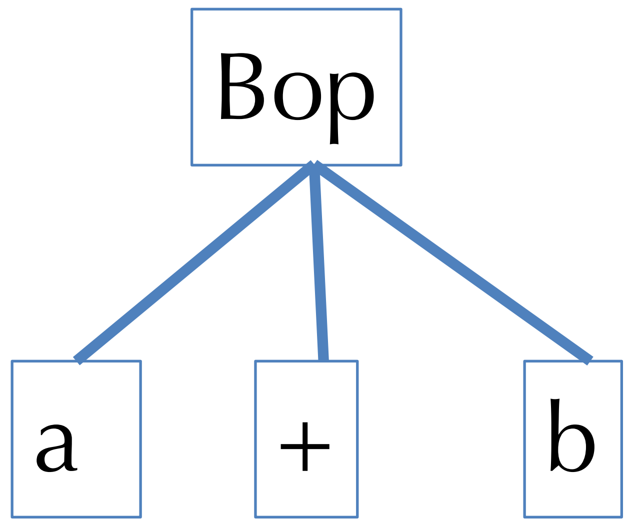
Abstract Syntax tree

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, correct initialisation, …

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
  – Why not use regular expressions on tokens to *specify programming language syntax*?

- Limits of regular expressions:
  – DFA's have only finite # of states
  – So… DFA's can't "count" (why is it a problem?)
- For example, consider the language of all strings that contain balanced parentheses – easier than most programming languages, but not regular (needs a stack to keep track of "(" and ")").

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

Context-Free Grammars

Context-Free Grammars

- 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 "$\longmapsto$") from object-language elements (e.g. "(").*

- The definition is *recursive* – S mentions itself.

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

- You can replace the "nonterminal" S by one of its definitions anywhere
- A context-free grammar ***accepts*** a string iff there is a derivation from the start symbol

* And, since we're writing this description in English, we are careful distinguish the meta-meta-language (e.g. words) from the meta-language and object-language (e.g. symbols) by using quotes.

CFGs Mathematically

- A Context-free Grammar (CFG) consists of
  - A set of *terminals* (e.g., a lexical token or 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 parenthesised sums of numbers:

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

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

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

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

4 productions

2 nonterminals: S, E

4 terminals: (,), +, number

Start symbol: S

Derivations in CFGs

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$S \longmapsto E + S \quad | \quad E$$

$$E \longmapsto \text{number} \quad | \quad (S)$$

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.

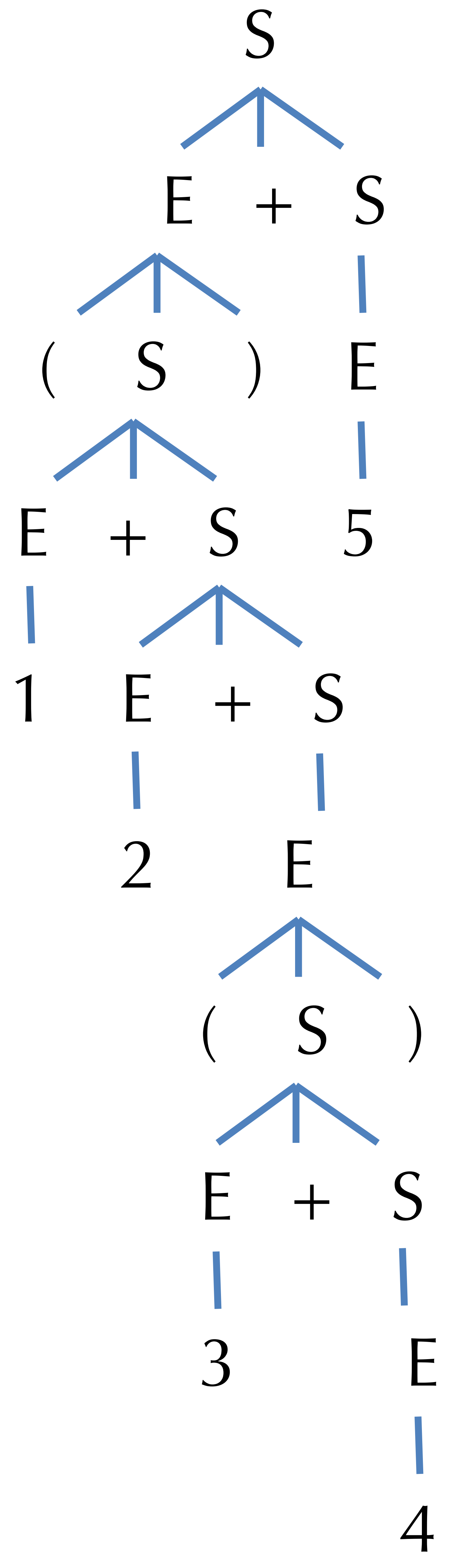
From Derivations to Parse Trees

- Tree representation of the derivation
- Leaves of the tree are *terminals*
  - In-order (DFS) traversal yields the input sequence of tokens
- Internal nodes: nonterminals
- No information about the *order* of the derivation steps

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

$S \longmapsto E + S \mid E$

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

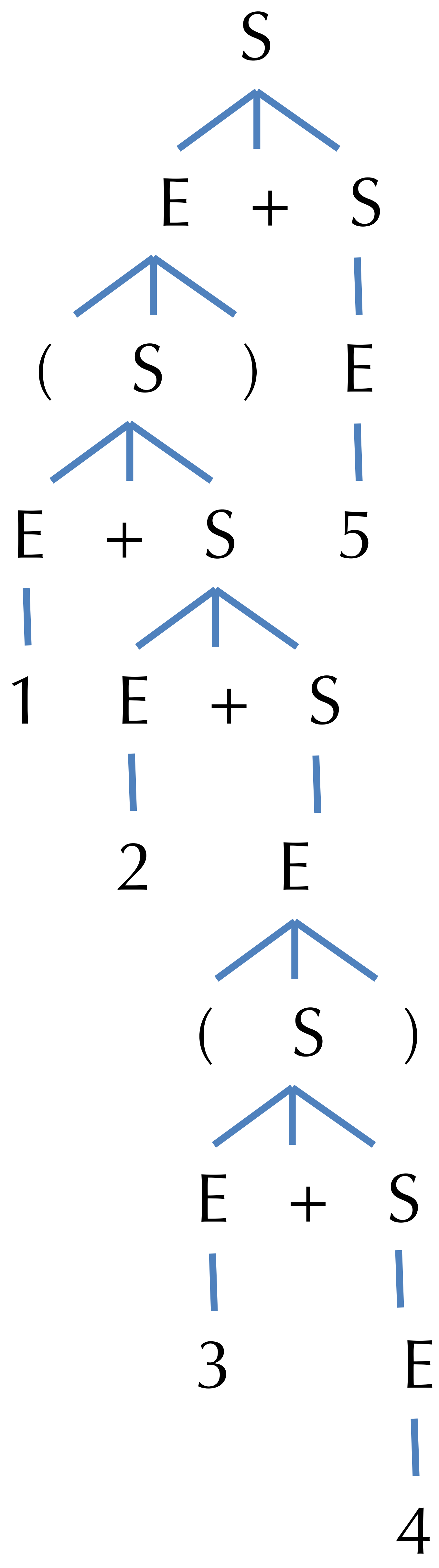


Parse Tree

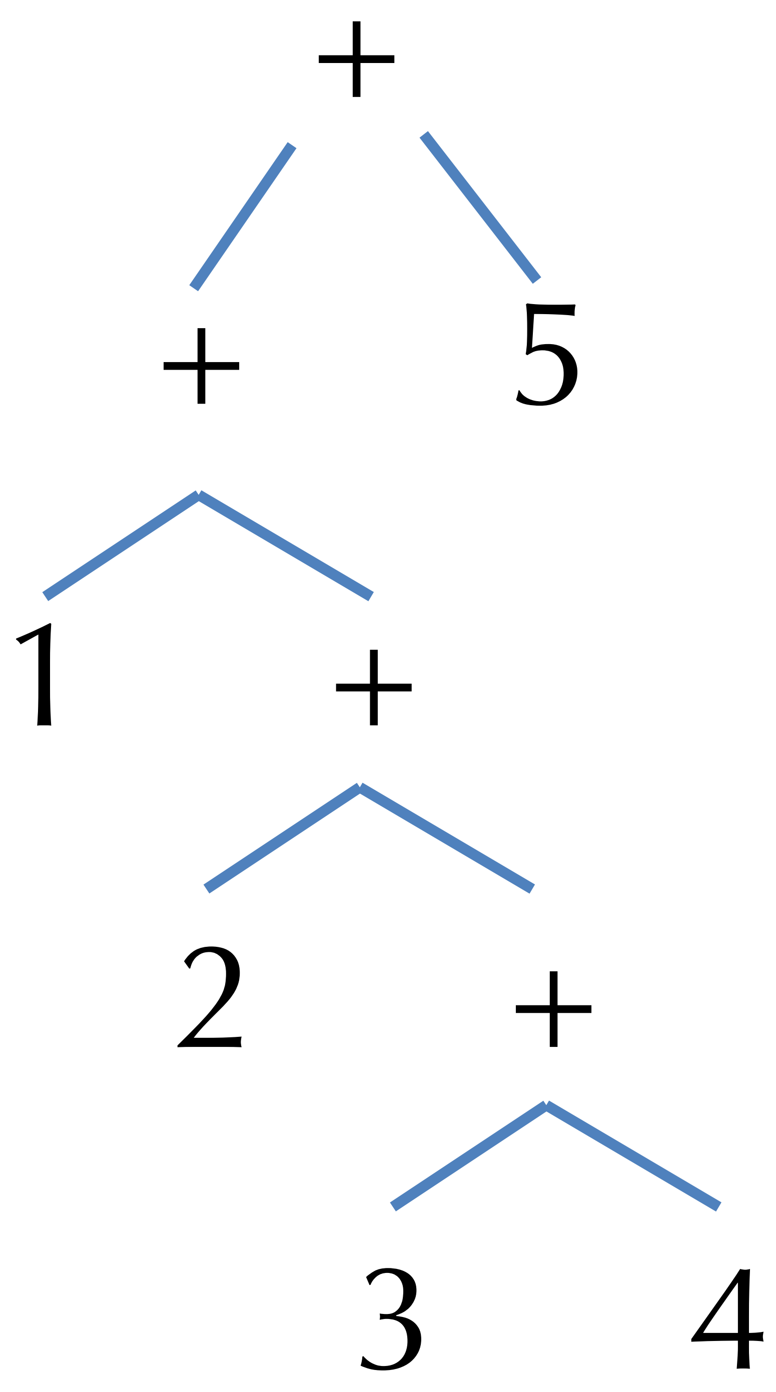
From Parse Trees to Abstract Syntax

- *Parse tree*:

"concrete syntax"



- *Abstract syntax tree* (AST):



- Hides, or *abstracts*, unneeded information.

Derivation Orders

- Productions of the grammar "fire" non-deterministically.
- They 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.

- Note that for this grammar 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

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

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

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

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

Rightmost derivation:

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

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

Loops and Termination

- Some care is needed when defining CFGs to avoid loops
- Consider:

$$S \longmapsto E$$
$$E \longmapsto S$$

- 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.
- Consider: $S \longmapsto (S)$
- This grammar is productive, but again there is no finite derivation starting from S, so the language is empty. One can easily generalise 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.

Grammars for Programming Languages

Associativity, ambiguity, and precedence

Associativity

Consider the input: $1 + 2 + 3$

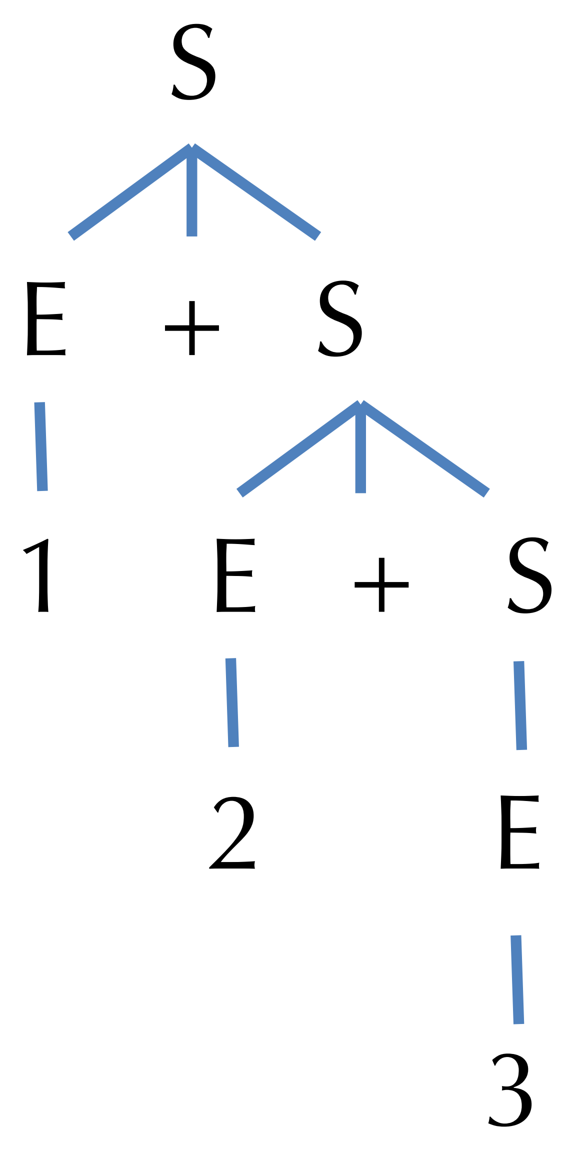
$$S \longmapsto E + S \mid E$$
$$E \longmapsto \text{number} \mid (S)$$

Leftmost derivation:

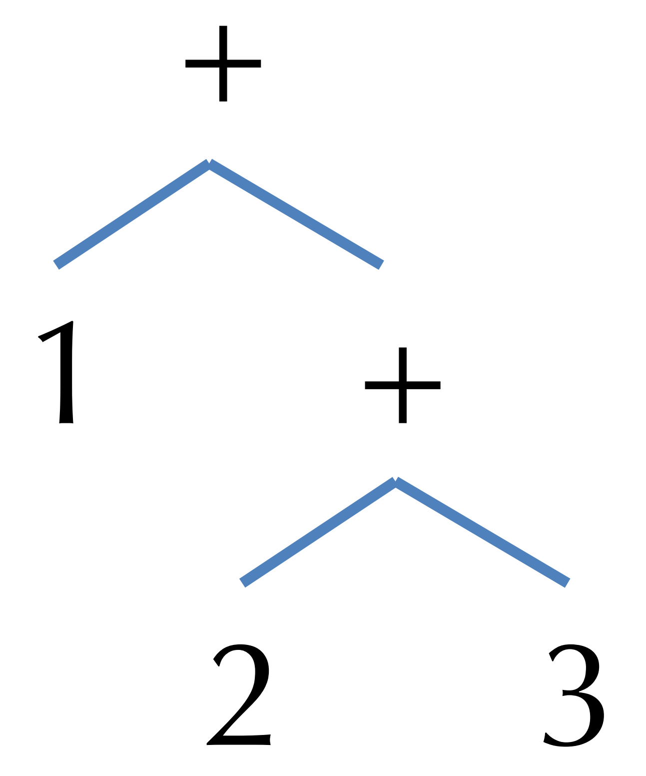
$$\underline{\mathbf{S}} \longmapsto \underline{\mathbf{E}} + S$$
$$\longmapsto 1 + \underline{\mathbf{S}}$$
$$\longmapsto 1 + \underline{\mathbf{E}} + S$$
$$\longmapsto 1 + 2 + \underline{\mathbf{S}}$$
$$\longmapsto 1 + 2 + \underline{\mathbf{E}}$$
$$\longmapsto 1 + 2 + 3$$

Rightmost derivation:

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



Parse Tree



AST

Associativity

- This grammar makes '+' *right associative*…
- The abstract syntax tree is *the same* for both

  1 + 2 + 3 and 1 + (2 + 3)
- Note that the grammar is *right recursive*…

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

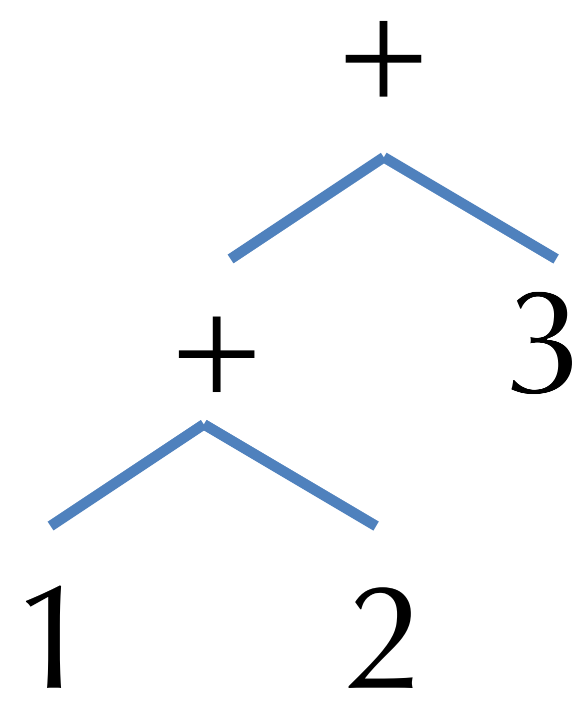
- How would you make '+' left associative?
- What are the trees for "1 + 2 + 3"?

Ambiguity

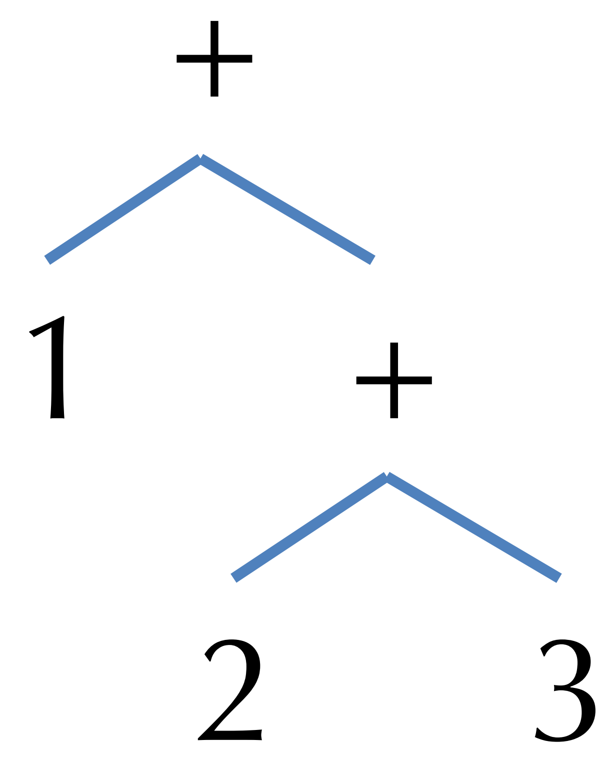
- Consider this grammar:

$$S \longmapsto \quad S + S \quad | \ (S) \ | \quad \text{number}$$

- Claim: it accepts the *same* set of strings as the previous one.
- What's the difference?
- Consider these *two* leftmost derivations for 1 + 2 + 3:
  - $\underline{\mathbf{S}} \longmapsto \underline{\mathbf{S}} + S \longmapsto 1 + \underline{\mathbf{S}} \longmapsto 1 + \underline{\mathbf{S}} + S \longmapsto 1 + 2 + \underline{\mathbf{S}} \longmapsto 1 + 2 + 3$
  - $\underline{\mathbf{S}} \longmapsto \underline{\mathbf{S}} + S \longmapsto \underline{\mathbf{S}} + S + S \longmapsto 1 + \underline{\mathbf{S}} + S \longmapsto 1 + 2 + \underline{\mathbf{S}} \longmapsto 1 + 2 + 3$
- One derivation gives left associativity, the other gives right associativity to '+'
  - Which is which?



AST 1



AST 2

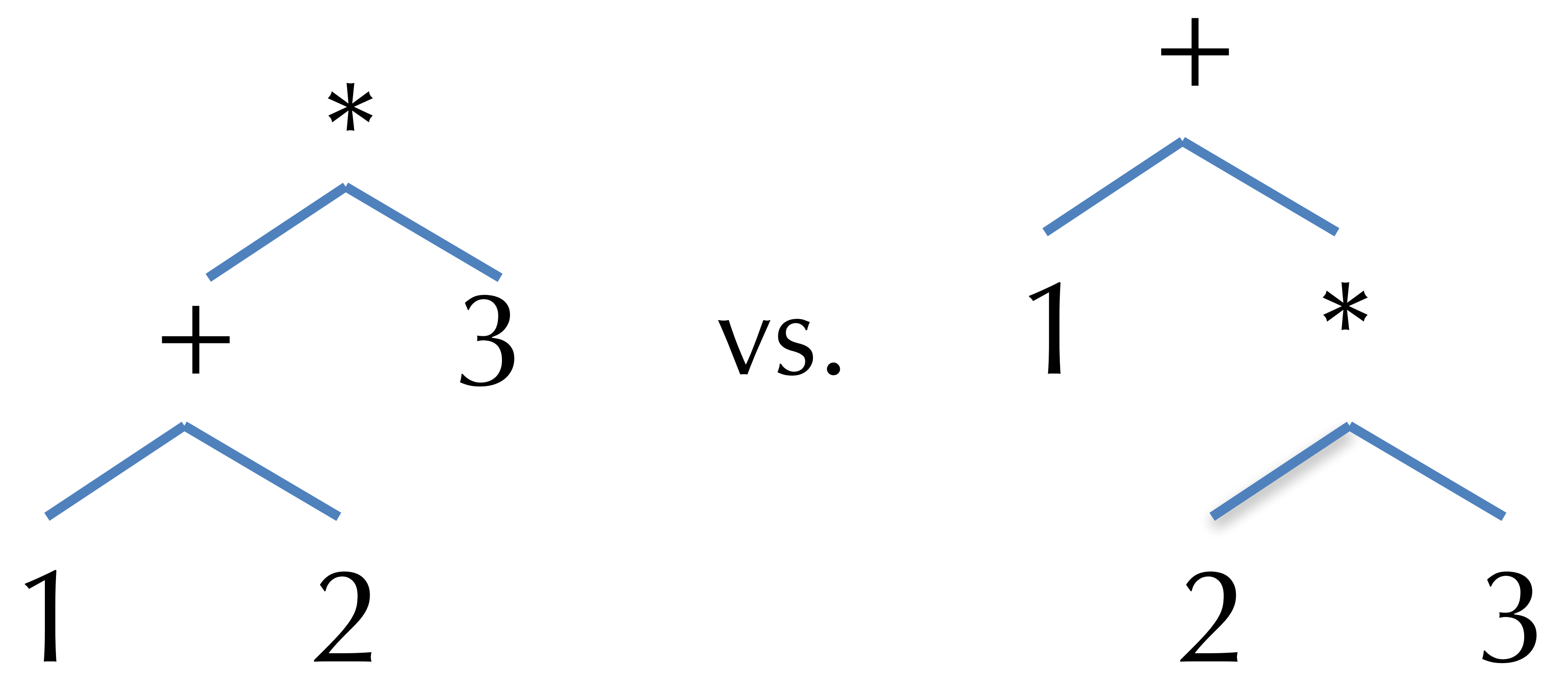
Precedence

- The ‘+’ operation is associative, so it doesn’t matter which tree we pick. Mathematically, $x + (y + z) = (x + y) + z$
  - But, some operations aren’t 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:

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

- 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
- Note:
  - S_2 corresponds to 'atomic' expressions

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

Context Free Grammars: Summary

- Context-free grammars allow concise specifications of programming 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 in this case all derivations correspond to the same abstract syntax tree.

- Still to come: how to *find* a derivation that matches the string of tokens?
  - But first, let's see some tools: menhir

Demo: Parsing for Boolean Logic

- https://github.com/ysc4230/week-06-parsing
- Definitions:
  - ast.ml
  - parser.mly
  - lexer.mll
  - range.ml
- What about precedence of binary connectives? Associativity?
- Running: main.ml