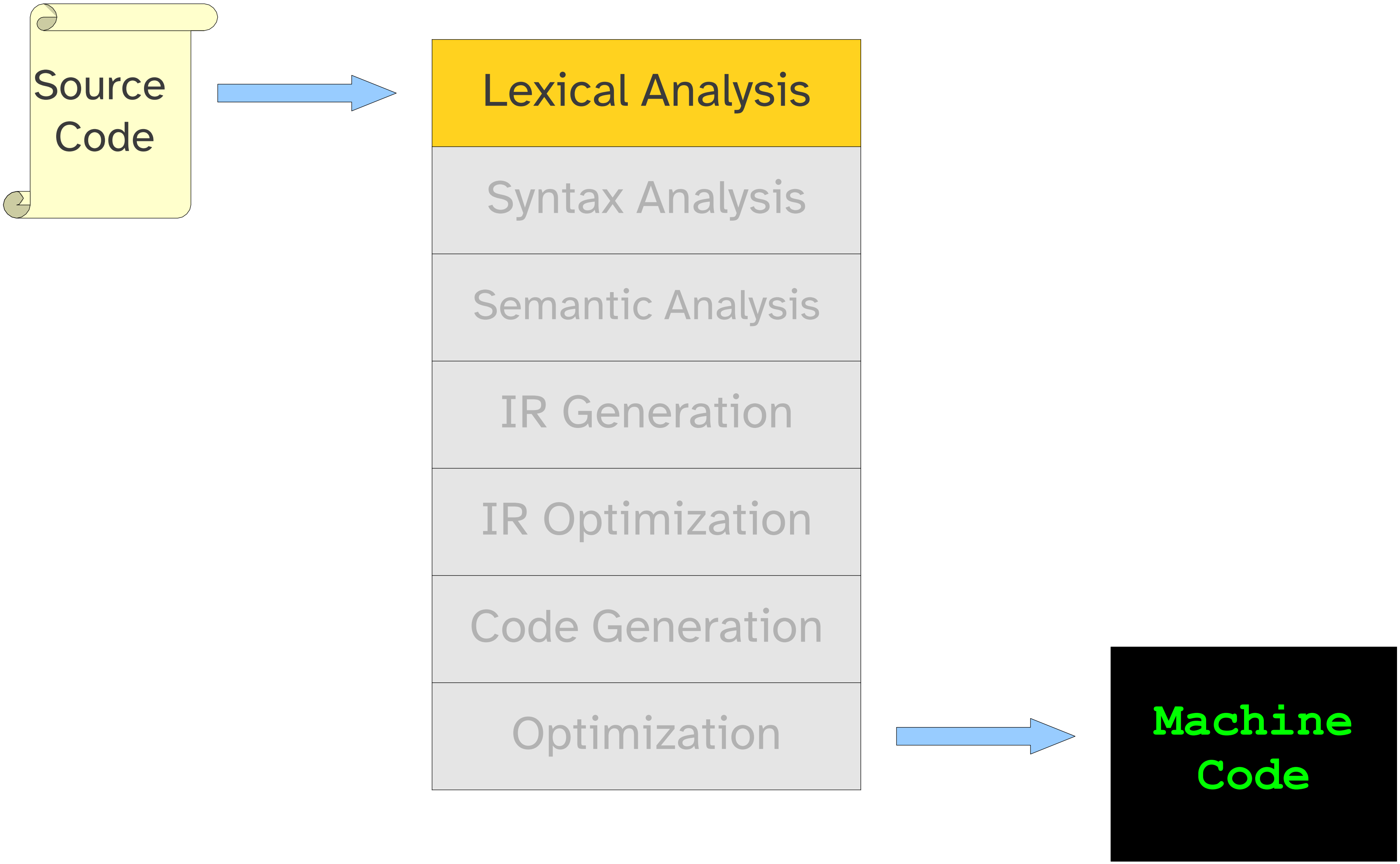
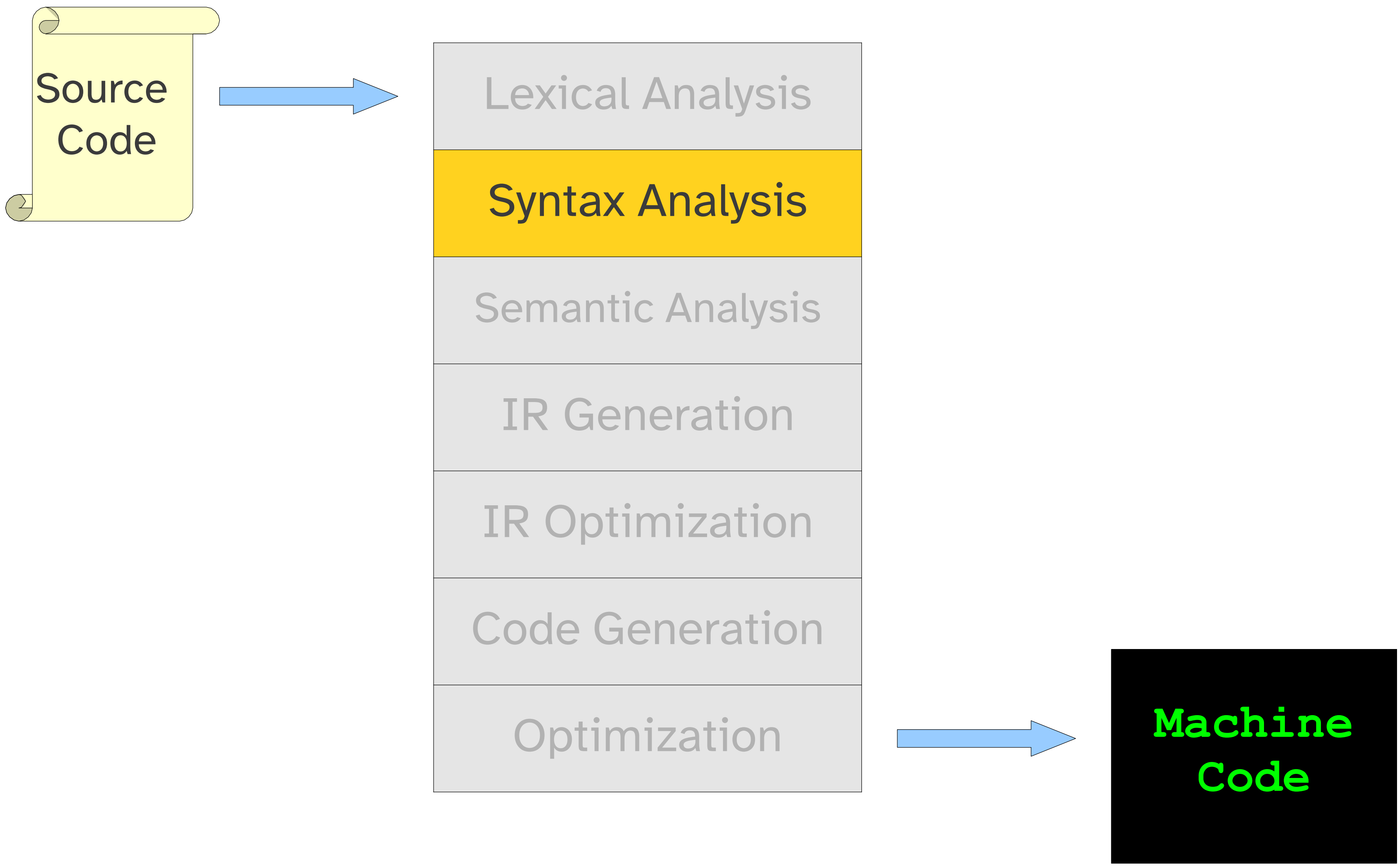
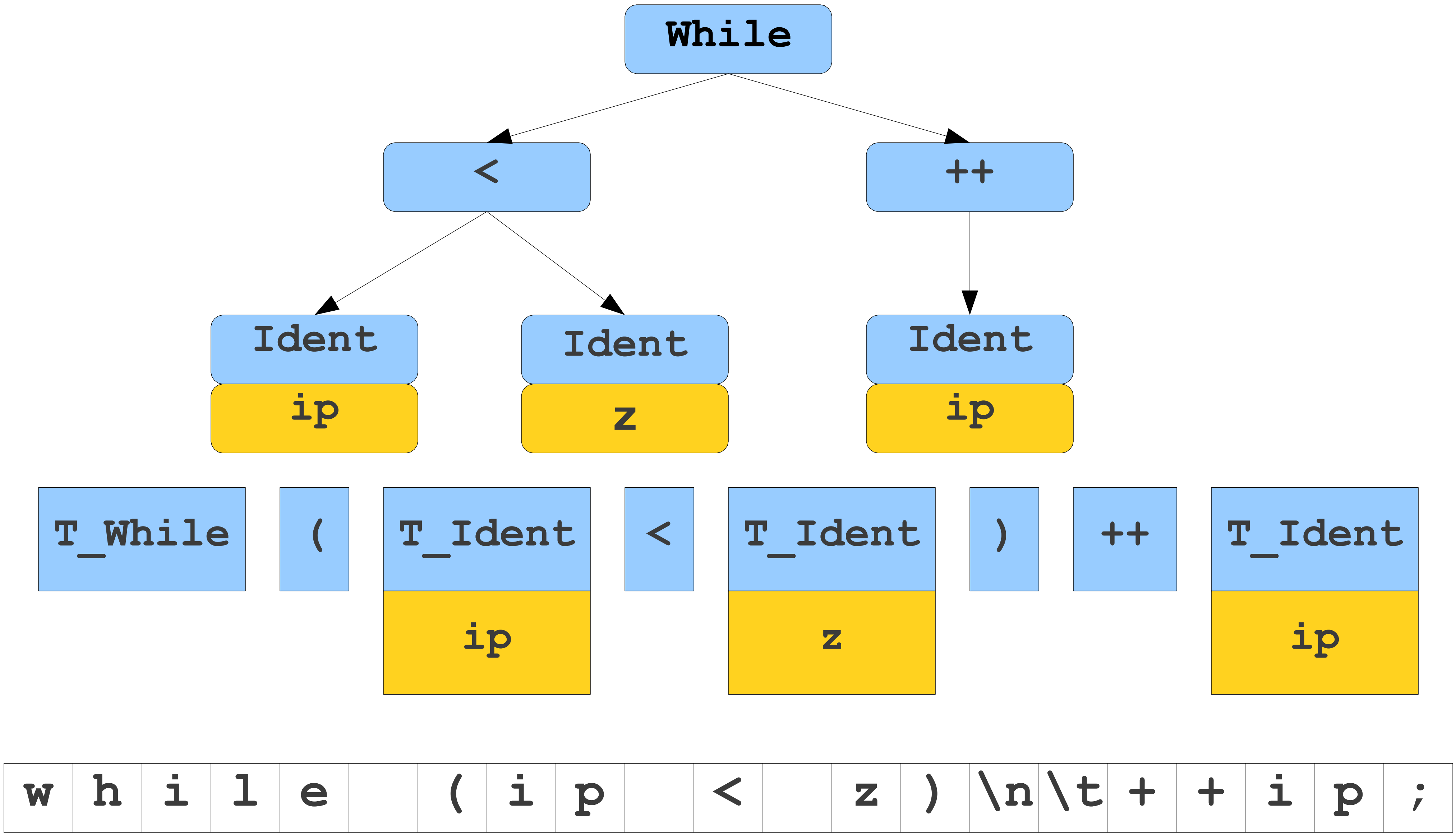
Syntax Analysis

Where We Were

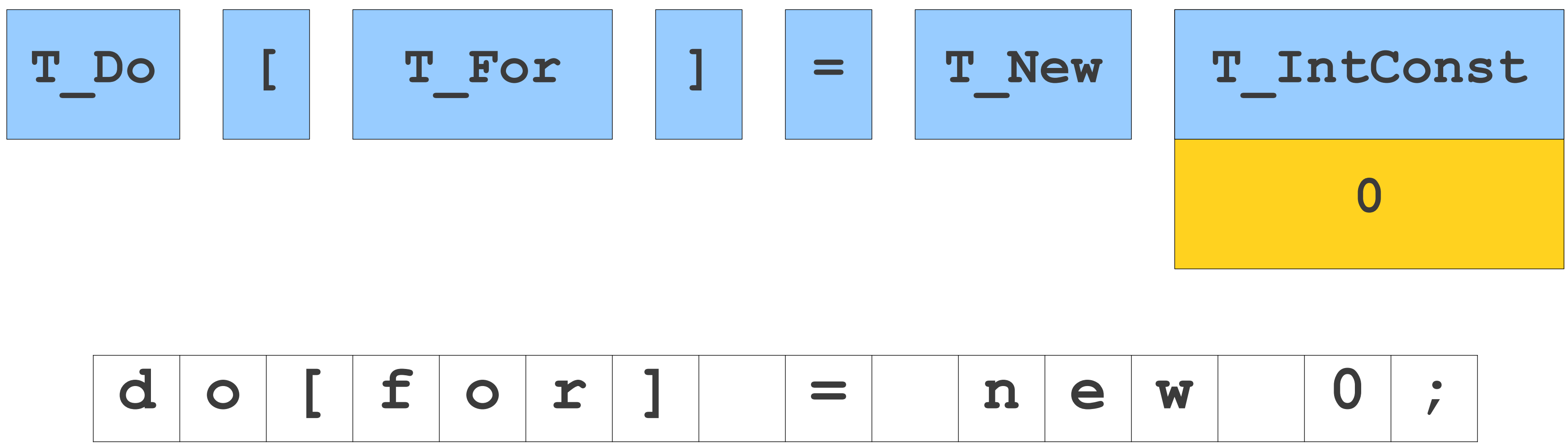


Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code


While
<
++
Ident
ip
Ident
z
Ident
ip
T_While
(
T_Ident
ip
<
T_Ident
z
)
++
T_Ident
ip
w h i l e (i p < z) \n \t + + i p ;

```
while (ip < z)
  ++ip;
```



do[for] = new 0;

Outline

- Today: Formalisms for syntax analysis
  - Context-Free Grammars
  - Derivations
  - Concrete and Abstract Syntax Trees
  - Ambiguity
- Later: Parsing algorithms
  - Top-Down Parsing
  - Bottom-Up Parsing

Context-Free Grammars

- A **context-free grammar** (or **CFG**) is a formalism for defining languages.
- Can define the **context-free languages**, a strict superset of the the regular languages.
- CFGs are best explained by example...

Arithmetic Expressions

- Suppose we want to describe all legal arithmetic expressions using addition, subtraction, multiplication, and division.
- Here is one possible CFG:

$$\mathbf{E} \rightarrow \texttt{int}$$

$$\mathbf{E} \rightarrow \mathbf{E}\ \mathbf{Op}\ \mathbf{E}$$

$$\mathbf{E} \rightarrow \texttt{(}\mathbf{E}\texttt{)}$$

$$\mathbf{Op} \rightarrow \texttt{+}$$

$$\mathbf{Op} \rightarrow \texttt{-}$$

$$\mathbf{Op} \rightarrow \texttt{*}$$

$$\mathbf{Op} \rightarrow \texttt{/}$$

Context-Free Grammars (CFGs)

A CFG is a collection of four objects:

- A set of **nonterminal symbols** (or **variables**),
- A set of **terminal symbols**,
- A set of **production rules** saying how each nonterminal can be converted by a string of terminals and nonterminals, and
- A **start symbol** that begins the derivation.

$$\mathbf{E} \rightarrow \texttt{int}$$
$$\mathbf{E} \rightarrow \mathbf{E}\ \mathbf{Op}\ \mathbf{E}$$
$$\mathbf{E} \rightarrow \texttt{(}\mathbf{E}\texttt{)}$$
$$\mathbf{Op} \rightarrow \texttt{+}$$
$$\mathbf{Op} \rightarrow \texttt{-}$$
$$\mathbf{Op} \rightarrow \texttt{*}$$
$$\mathbf{Op} \rightarrow \texttt{/}$$

CFGs for Programming Languages

```
BLOCK   →  STMT
        |  { STMTS }

STMTS   →  ε
        |  STMT STMTS

STMT    →  EXPR;
        |  if (EXPR) BLOCK
        |  while (EXPR) BLOCK
        |  do BLOCK while (EXPR);
        |  BLOCK
        |  …

EXPR    →  identifier
        |  constant
        |  EXPR + EXPR
        |  EXPR – EXPR
        |  EXPR * EXPR
        |  …
```

Some CFG Notation

- Capital letters at the beginning of the alphabet will represent nonterminals.
  - i.e., **A**, **B**, **C**, **D**
- Lowercase letters at the end of the alphabet will represent terminals.
  - i.e., **`t`**, **`u`**, **`v`**, **`w`**
- Lowercase Greek letters will represent arbitrary strings of terminals and nonterminals.
  - i.e., α, γ, ω

Examples

- We might write an arbitrary production as

$$\mathbf{A} \to \omega$$

- We might write a string of a nonterminal followed by a terminal as

$$\mathbf{A}\mathtt{t}$$

- We might write an arbitrary production containing a nonterminal followed by a terminal as

$$\mathbf{B} \to \alpha\mathbf{A}\mathtt{t}\omega$$

Derivations

```
  E
⇒ E Op E
⇒ E Op (E)
⇒ E Op (E Op E)
⇒ E * (E Op E)
⇒ int * (E Op E)
⇒ int * (int Op E)
⇒ int * (int Op int)
⇒ int * (int + int)
```

- This sequence of steps is called a **derivation**.
- A string $\alpha \mathbf{A} \omega$ **yields** string $\alpha\gamma\omega$ iff $\mathbf{A} \rightarrow \gamma$ is a production.
- If α yields β, we write $\alpha \Rightarrow \beta$.
- We say that α **derives** β iff there is a sequence of strings where

$$\alpha \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \beta$$

- If α derives β, we write $\alpha \Rightarrow^* \beta$.

Derivations

- A **leftmost derivation** is a derivation in which each step expands the leftmost nonterminal.
- A **rightmost derivation** is a derivation in which each step expands the rightmost nonterminal.
- These will be of great importance when we talk about parsing later.

Derivations Examples

E	E
⇒ E Op E	⇒ E Op E
⇒ `int` Op E	⇒ E Op `(`E`)`
⇒ `int *` E	⇒ E Op `(`E Op E`)`
⇒ `int * (`E`)`	⇒ E Op `(`E Op `int)`
⇒ `int * (`E Op E`)`	⇒ E Op `(`E `+ int)`
⇒ `int * (int` Op E`)`	⇒ E Op `(int + int)`
⇒ `int * (int +` E`)`	⇒ E `* (int + int)`
⇒ `int * (int + int)`	⇒ `int * (int + int)`

Derivations Revisited

- A derivation encodes two pieces of information:
    - What productions were applied produce the resulting string from the start symbol?
    - In what order were they applied?
- Multiple derivations might use the same productions, but apply them in a different order.

Parse Trees

- A **parse tree** is a tree encoding the steps in a derivation.
- Internal nodes represent nonterminal symbols used in the production.
- Inorder walk of the leaves contains the generated string.
- Encodes what productions are used, not the order in which those productions are applied.

Parse Trees

E

⇒ E Op E

⇒ int Op E

⇒ int * E

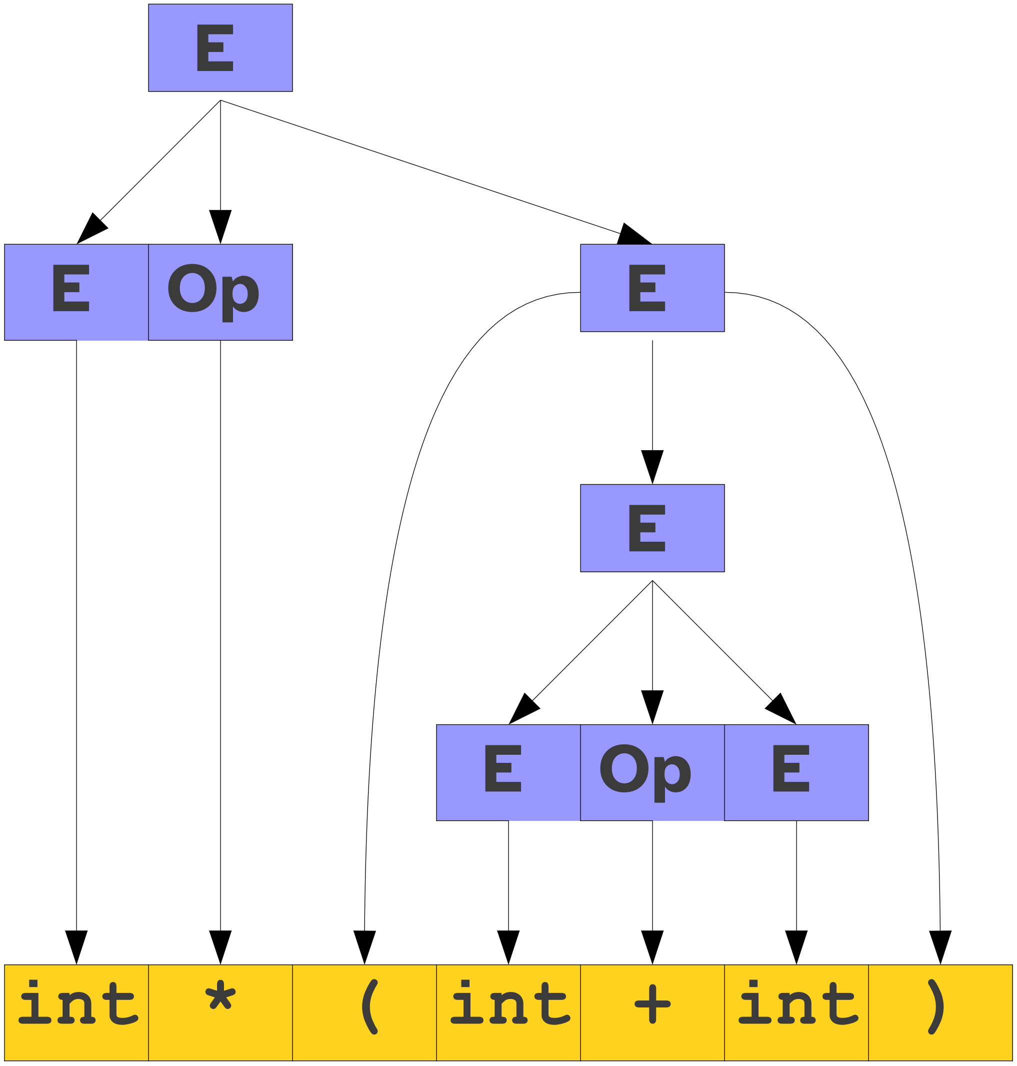
⇒ int * (E)

⇒ int * (E Op E)

⇒ int * (int Op E)

⇒ int * (int + E)

⇒ int * (int + int)

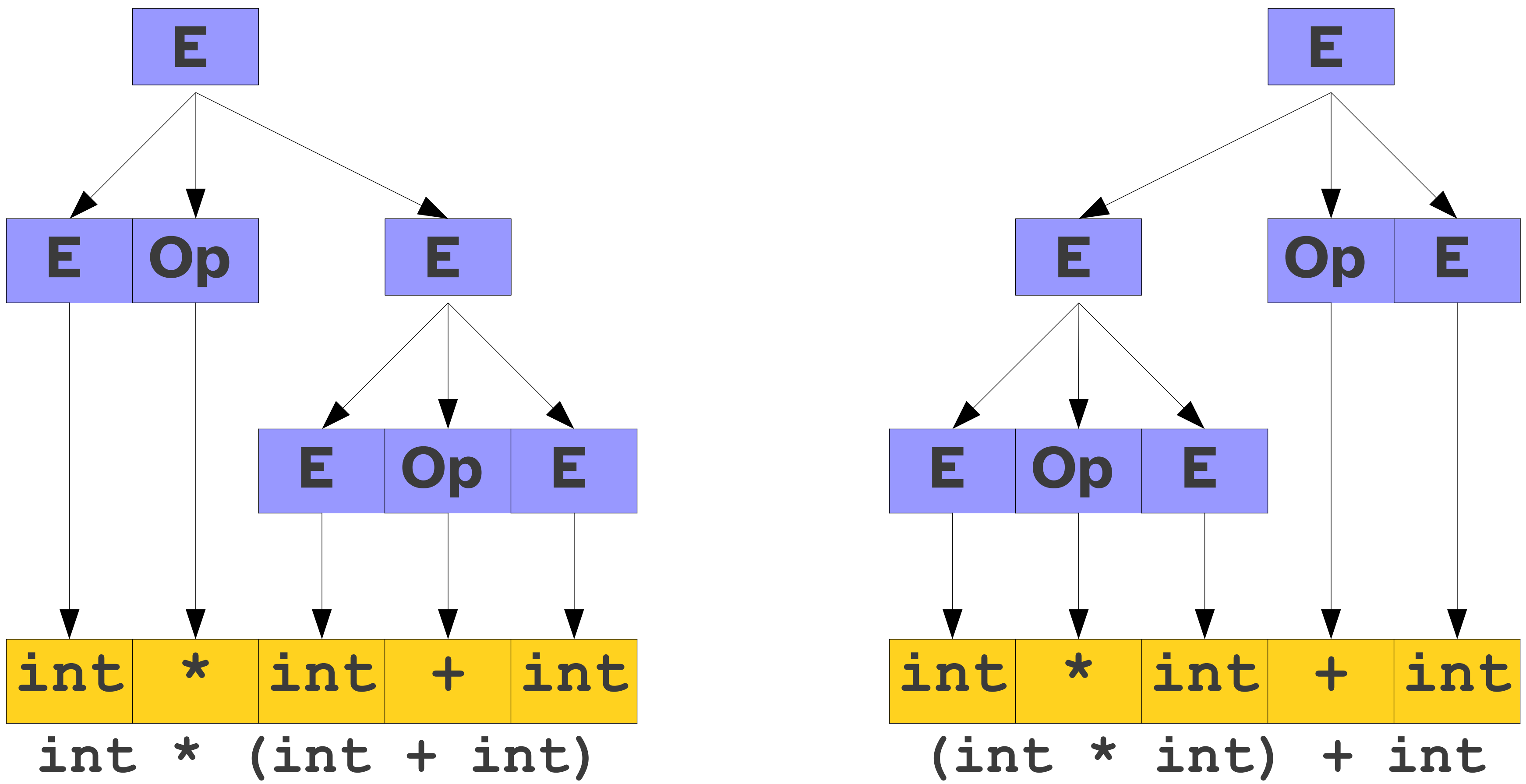


The Goal of Parsing

- Goal of syntax analysis: Recover the **structure** described by a series of tokens.
- If language is described as a CFG, goal is to recover a parse tree for the the input string.

Challenges in Parsing

A Serious Problem



Ambiguity

- A CFG is said to be **ambiguous** if there is at least one string with two or more parse trees.
  - Using the same leftmost/rightmost derivation, one string can have at least 2 ways to be derived.
- There is no algorithm for converting an arbitrary ambiguous grammar into an unambiguous one.
- There is no algorithm for detecting whether an arbitrary grammar is ambiguous.

Example: With string `int * (int + int)`

E → `int` |
E Op E |
`(`E`)`

Op → `+` | `-` |
`*` | `/`

E

Example: With string `int * (int + int)`

$E \rightarrow$ `int` |

E Op E |

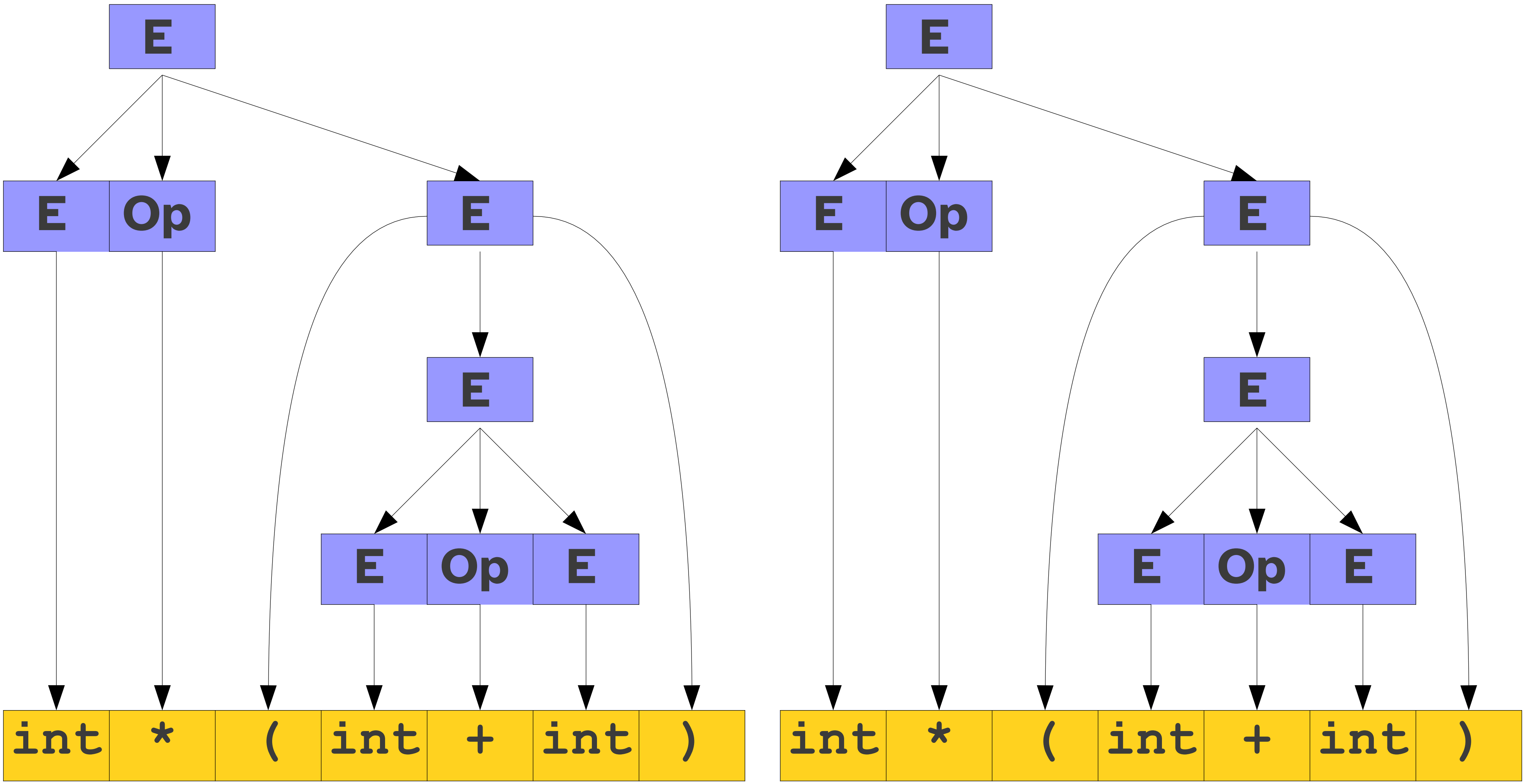
`(`E`)`

Op $\rightarrow$ `+` | `-` |

`*` | `/`

E

This is ambiguous



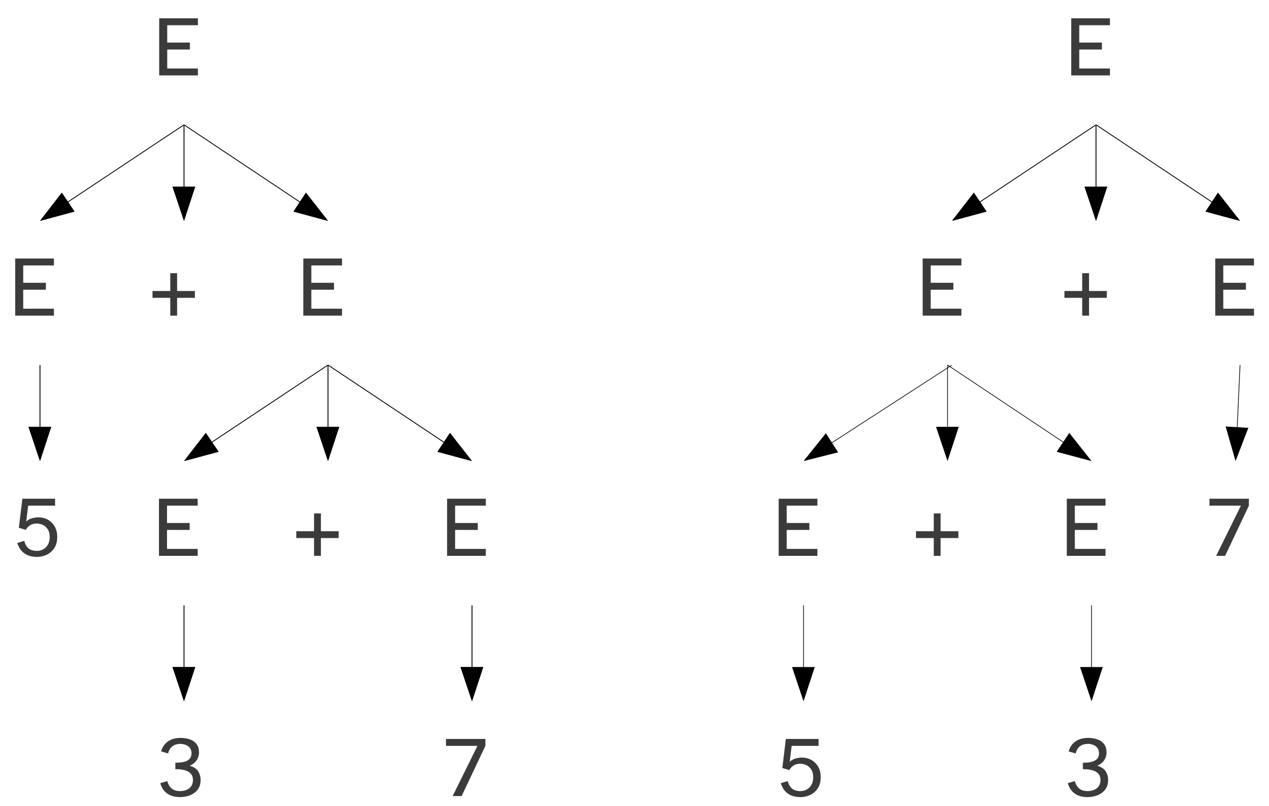
Because 1 strings created by 2 ways

Wrong! The parse trees are exactly the same!

Is Ambiguity a Problem?

- Depends on **semantics**.

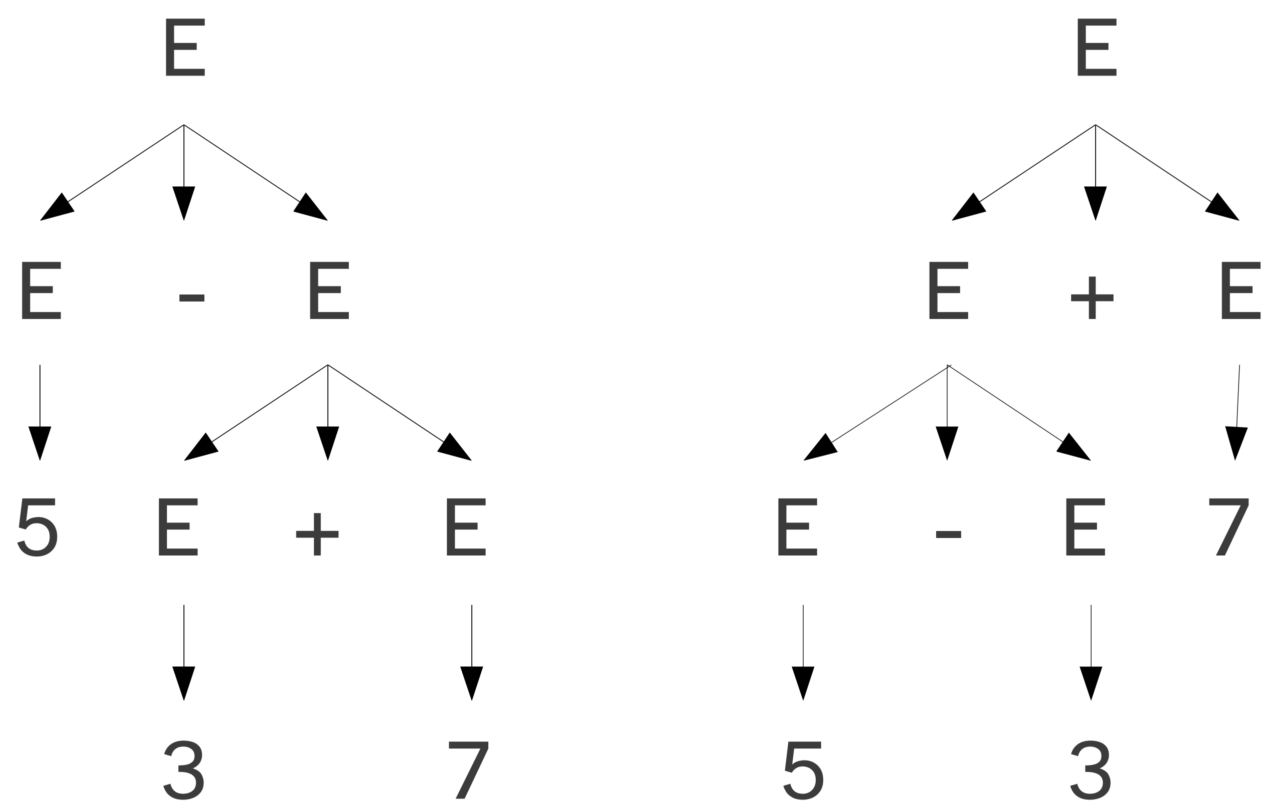
$$\mathbf{E} \rightarrow \texttt{int} \mid \mathbf{E} + \mathbf{E}$$



Is Ambiguity a Problem?

- Depends on **semantics**.

$$E \rightarrow \texttt{int} \mid E + E \mid E - E$$



Drawbacks of ambiguous grammars

- Ambiguous semantics
- Parsing complexity
- May affect other phases
- Solutions
  - Allow only non-ambiguous grammars
  - Transform grammar into non-ambiguous
  - Handle as part of parsing method
    - Using special form of “precedence”

Resolving Ambiguity

- If a grammar can be made unambiguous at all, it is usually made unambiguous through **layering**.
- Have exactly one way to build each piece of the string.
- Have exactly one way of combining those pieces back together.

CFG for RegEx

- Recall: A regular expression can be
  - Any letter
  - ε
  - The concatenation of regular expressions.
  - The union of regular expressions.
  - The Kleene closure of a regular expression.
  - A parenthesized regular expression.

CFG for RegEx

- This gives us the following CFG:

  $\mathbf{R} \rightarrow$ `a` | `b` | `c` | …

  $\mathbf{R} \rightarrow$ "`ε`"

  $\mathbf{R} \rightarrow \mathbf{RR}$

  $\mathbf{R} \rightarrow \mathbf{R}$ "`|`" $\mathbf{R}$

  $\mathbf{R} \rightarrow \mathbf{R}$`*`

  $\mathbf{R} \rightarrow$ `(`$\mathbf{R}$`)`

An Ambiguous Grammar

R → a | b | c | ...

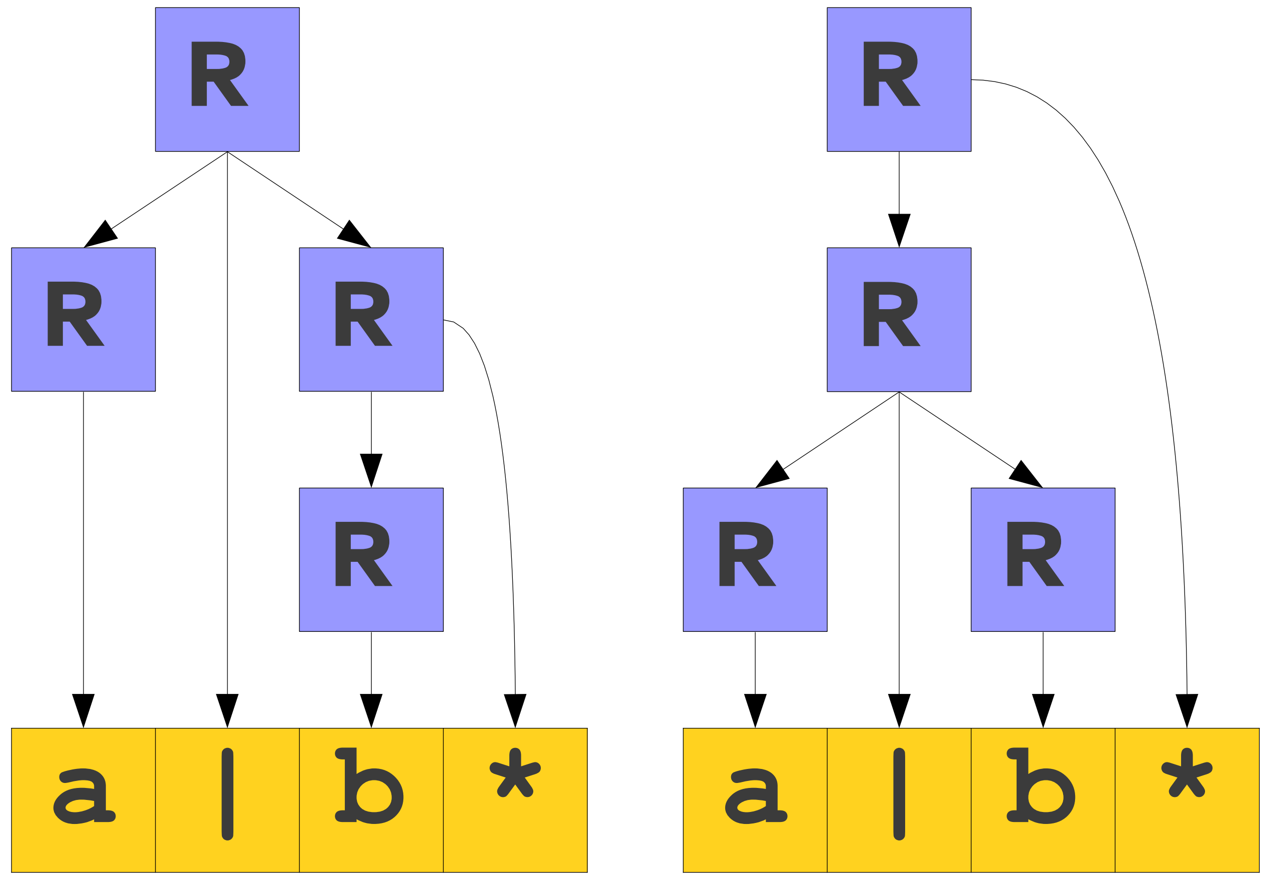
R → “ε”

R → RR

R → R “|” R

R → R*

R → (R)



a | (b*)   (a | b)*

Resolving Ambiguity

- We can try to resolve the ambiguity via layering:

R → `a` | `b` | `c` | …

R → "`ε`"

R → **RR**

R → **R** "`|`" **R**

R → **R**`*`

R → `(`**R**`)`

Resolving Ambiguity

- We can try to resolve the ambiguity via layering:

$R \rightarrow$ a | b | c | …

$R \rightarrow$ "ε"

$R \rightarrow RR$

$R \rightarrow R$ "|" R

$R \rightarrow R$*

$R \rightarrow$ (R)

$R \rightarrow S$ | R "|" S

$S \rightarrow T$ | ST

$T \rightarrow U$ | T*

$U \rightarrow$ a | b | c | …

$U \rightarrow$ "ε"

$U \rightarrow$ (R)

Why is this unambiguous?

$R \longrightarrow S \mid R$ "|" S

$S \longrightarrow T \mid ST$

$T \longrightarrow U \mid T$*

$U \longrightarrow$ a | b | …

$U \longrightarrow$ "ε"

$U \longrightarrow$ (R)

Infix Expression Example

Given grammar G:

E → **E** "+" **E** |
E "-" **E** |
"-" **E** |
E "*" **E** |
E "/" **E** |
E "^" **E** |
"(" **E** ")" |
var

G is ambiguous

What about G1? (Wirth notation)

E → **P** { **B** **P** }
P → **var** | "(" **E** ")" | **U** **P**
B → "+" | "-" | "*" | "/" | "^"
U → "-"

Also, is L(G1) = L(G)?

Another solution: G2 grammar

E $\longrightarrow$ **T** { ("+" | "-") **T** }

T $\longrightarrow$ **F** { ("*" | "/") **F** }

F $\longrightarrow$ **P** ["^" **F**]

P $\longrightarrow$ **var** | "(" **E** ")" | "-" **T**

- Is G2 ambiguous?
- Is L(G2)=L(G1)?
- Is L(G2)=L(G)?

What can we tell about G?

$E \rightarrow T$

$E \rightarrow E + T$

$T \rightarrow F$

$T \rightarrow T * F$

$F \rightarrow \mathbf{var}$

$F \rightarrow (E)$

- Unambiguous
- () has highest precedence
- * has precedence over +
- * and +: left associative: operations to be done from left to right
- e.g., n+n+n, n+n*n

If G is changed to G’

$E \to T$

$E \to T + E$

$T \to F$

$T \to T * F$

$F \to \mathbf{var}$

$F \to (E)$

- Unambiguous
- * has precedence over +
- * is left associative
  - left recursive
- + is right associative
  - right recursive

Note:

- while L(G) is equivalent to L(G’),
- G is **not** equivalent to G’: + is now right associative

The Structure of a Parse Tree

R → **S** | **R** "|" **S**

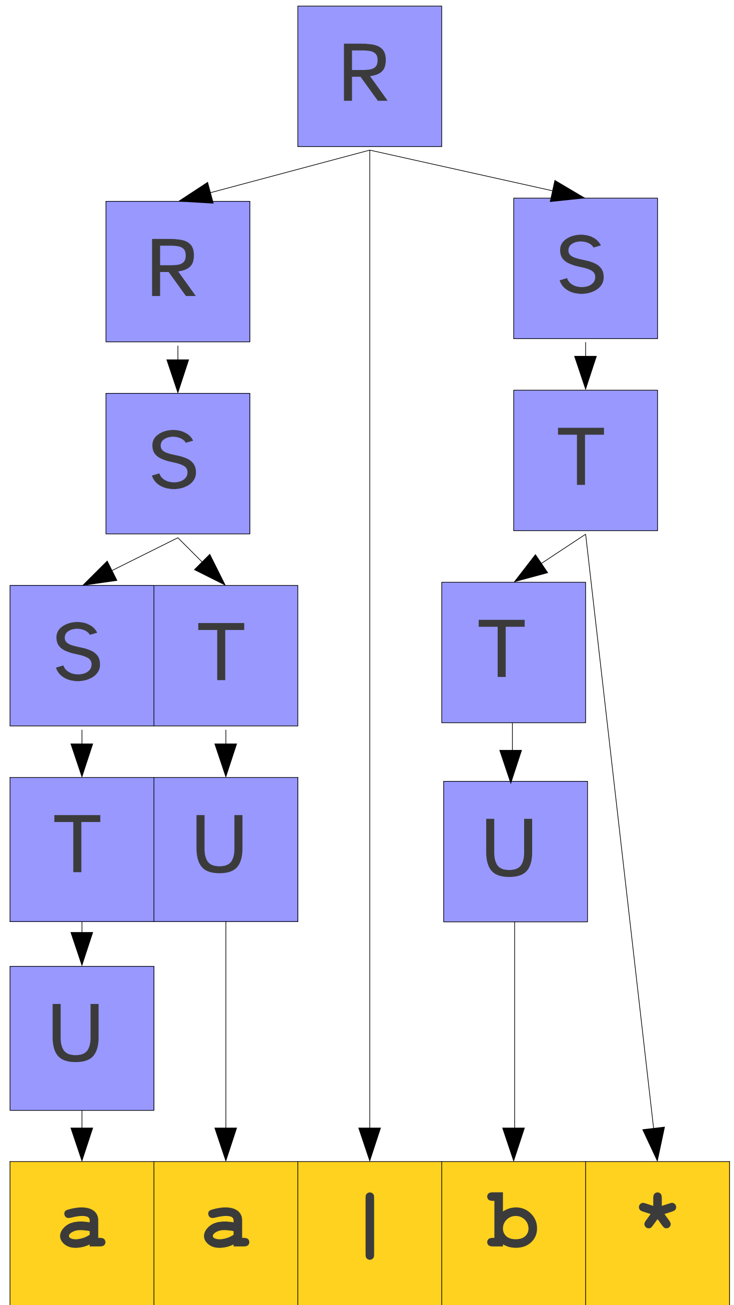
S → **T** | **ST**

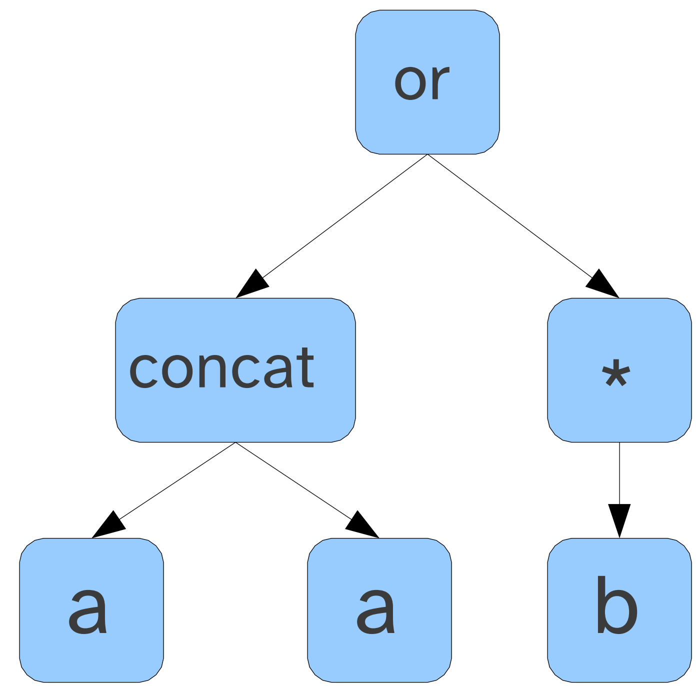
T → **U** | **T***

U → **a** | **b** | …

U → "**ε**"

U → **(R)**





R → S | R "|" S

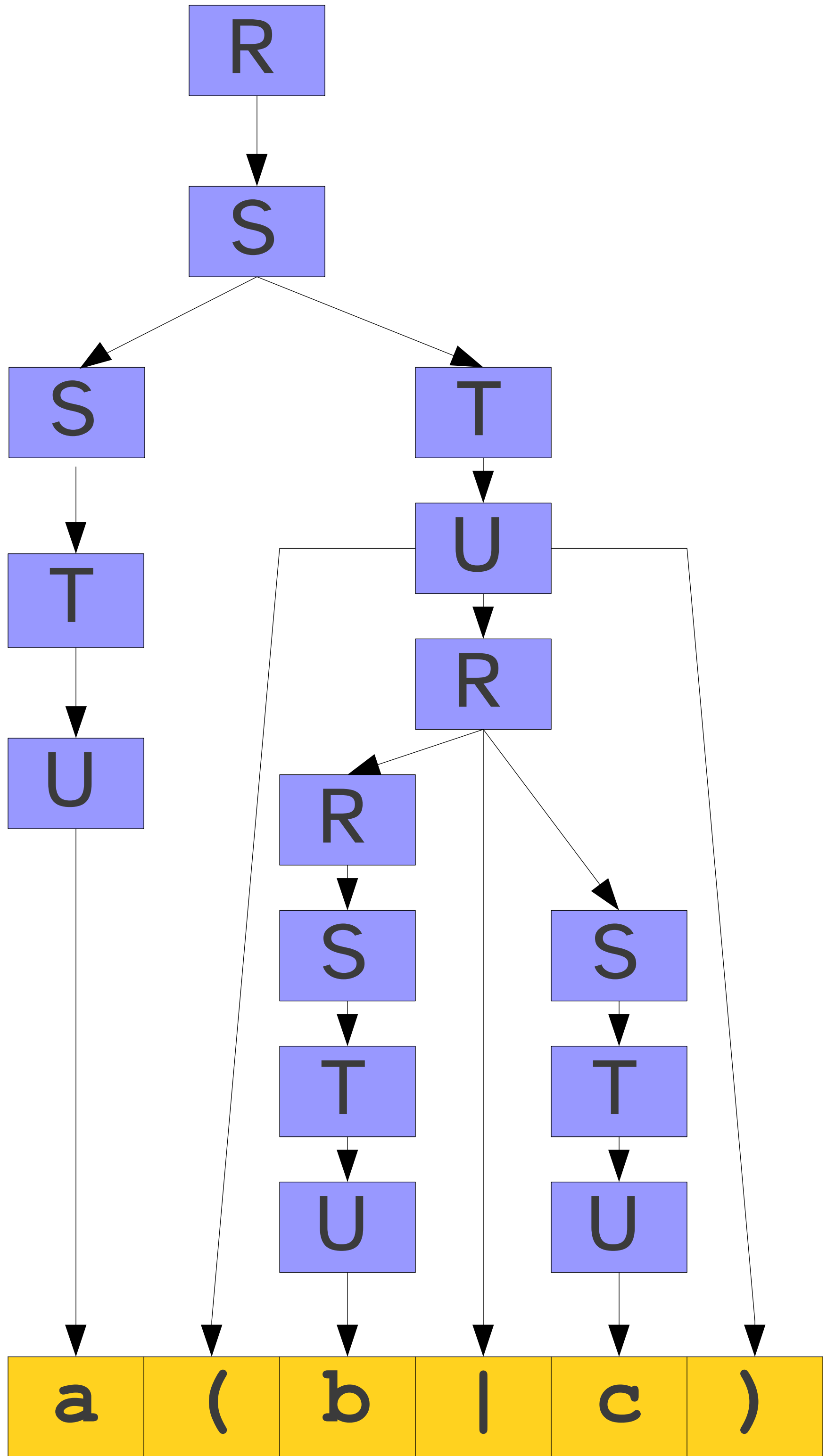
S → T | ST

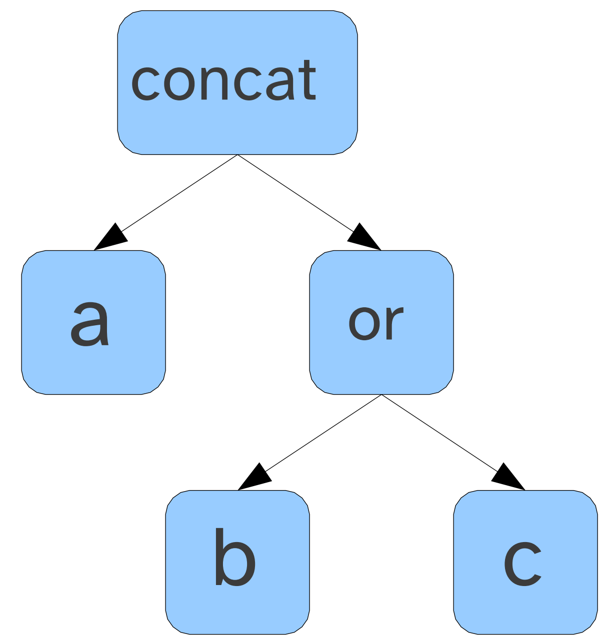
T → U | T*

U → a | b | …

U → "ε"

U → (R)





Abstract Syntax Trees (ASTs)

- A parse tree is a **concrete syntax tree**; it shows exactly how the text was derived.
- A more useful structure is an **abstract syntax tree**, which retains only the essential structure of the input.

How to build an AST?

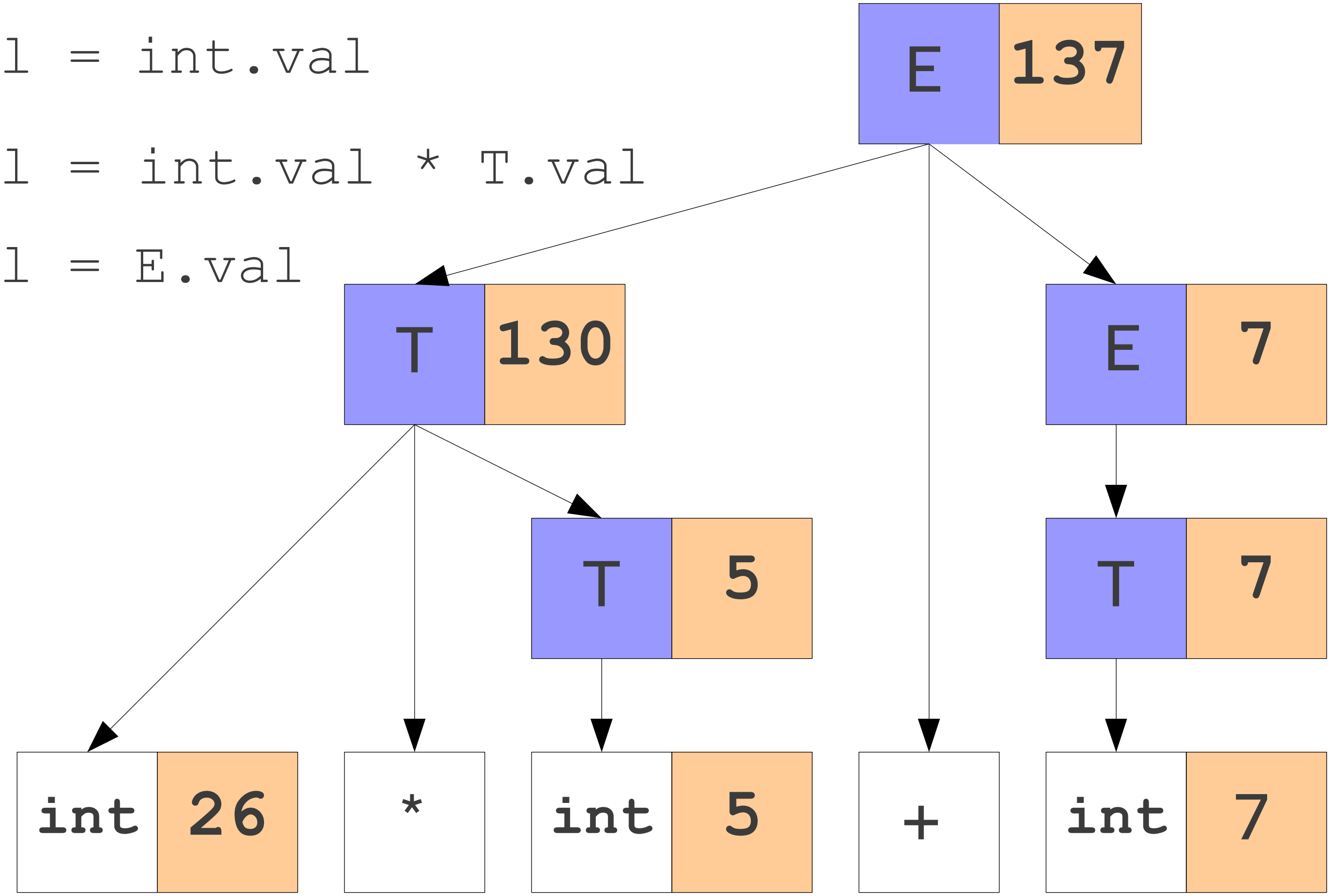
- Typically done through **semantic actions**.
- Associate a piece of code to execute with each production.
- As the input is parsed, execute this code to build the AST.

Semantic Actions to Build ASTs

$R \to S$	`R.ast = S.ast;`
$R \to R$ "|" S	R_1`.ast = new Or(`R_2`.ast, S.ast);`
$S \to T$	`S.ast = T.ast;`
$S \to ST$	S_1`.ast = new Concat(`S_2`.ast, T.ast);`
$T \to U$	`T.ast = U.ast;`
$T \to T$*	T_1`.ast = new Star(`T_2`.ast);`
$U \to$ a	`U.ast = new SingleChar('a');`
$U \to$ "ε"	`U.ast = new Epsilon();`
$U \to$ (R)	`U.ast = R.ast;`

Another example: Semantic Actions

$E \rightarrow T + E$	$E_1.val = T.val + E_2.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow int$	$T.val = int.val$
$T \rightarrow int * T$	$T.val = int.val * T.val$
$T \rightarrow (E)$	$T.val = E.val$



Summary

- Syntax analysis (**parsing**) extracts the structure from the tokens produced by the scanner.
- Languages are usually specified by **context-free grammars** (**CFG**s).
- A **parse tree** shows how a string can be **derived** from a grammar.
- A grammar is **ambiguous** if it can derive the same string multiple ways.
  - There is no algorithm for eliminating ambiguity; it must be done by hand.
- **Abstract syntax trees** (**AST**s) contain an abstract representation of a program's syntax.
- **Semantic actions** associated with productions can be used to build ASTs.

Next Time

- **Top-Down Parsing**
  - Parsing as a Search
  - Backtracking Parsers
  - Predictive Parsers LL(1)