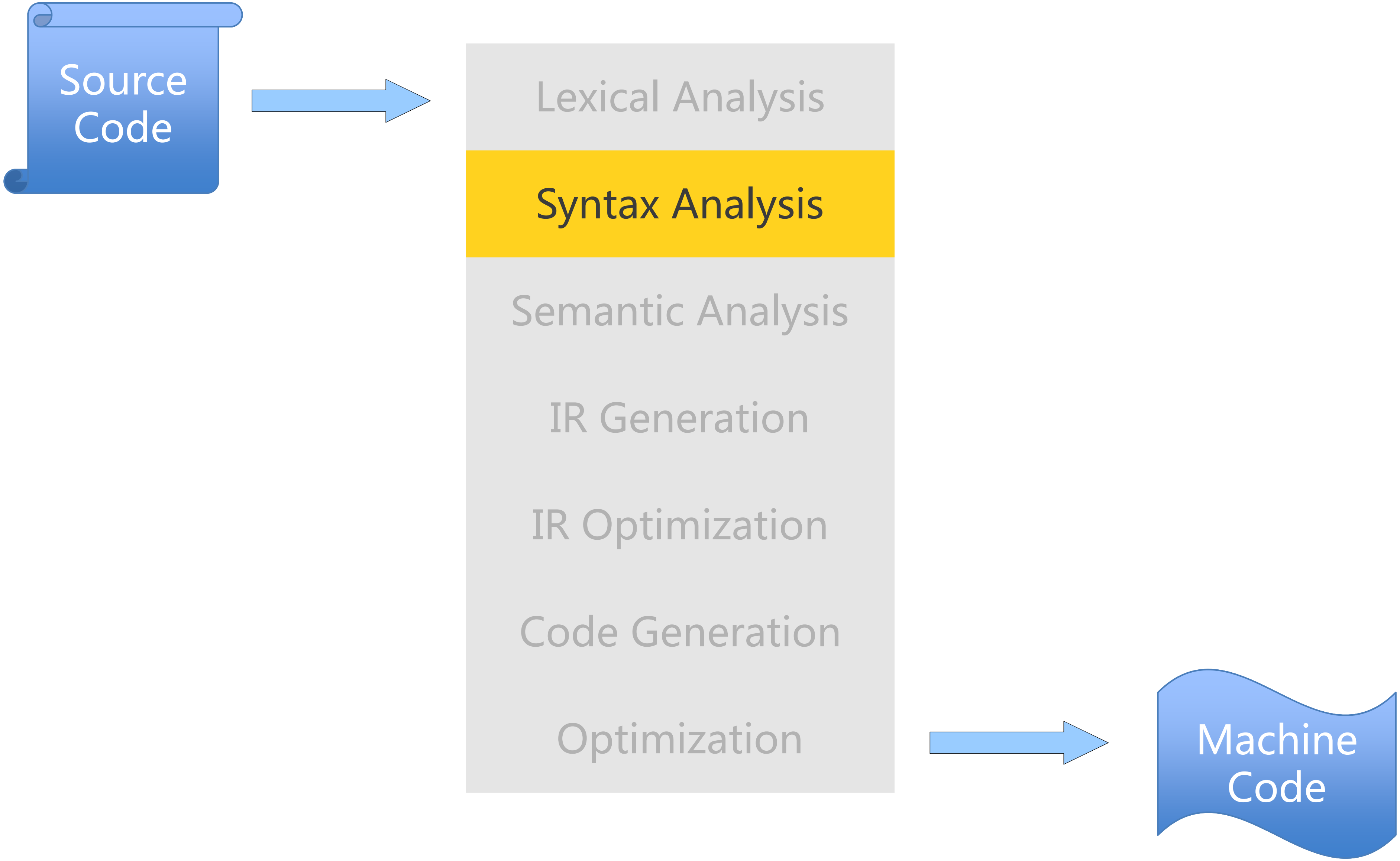
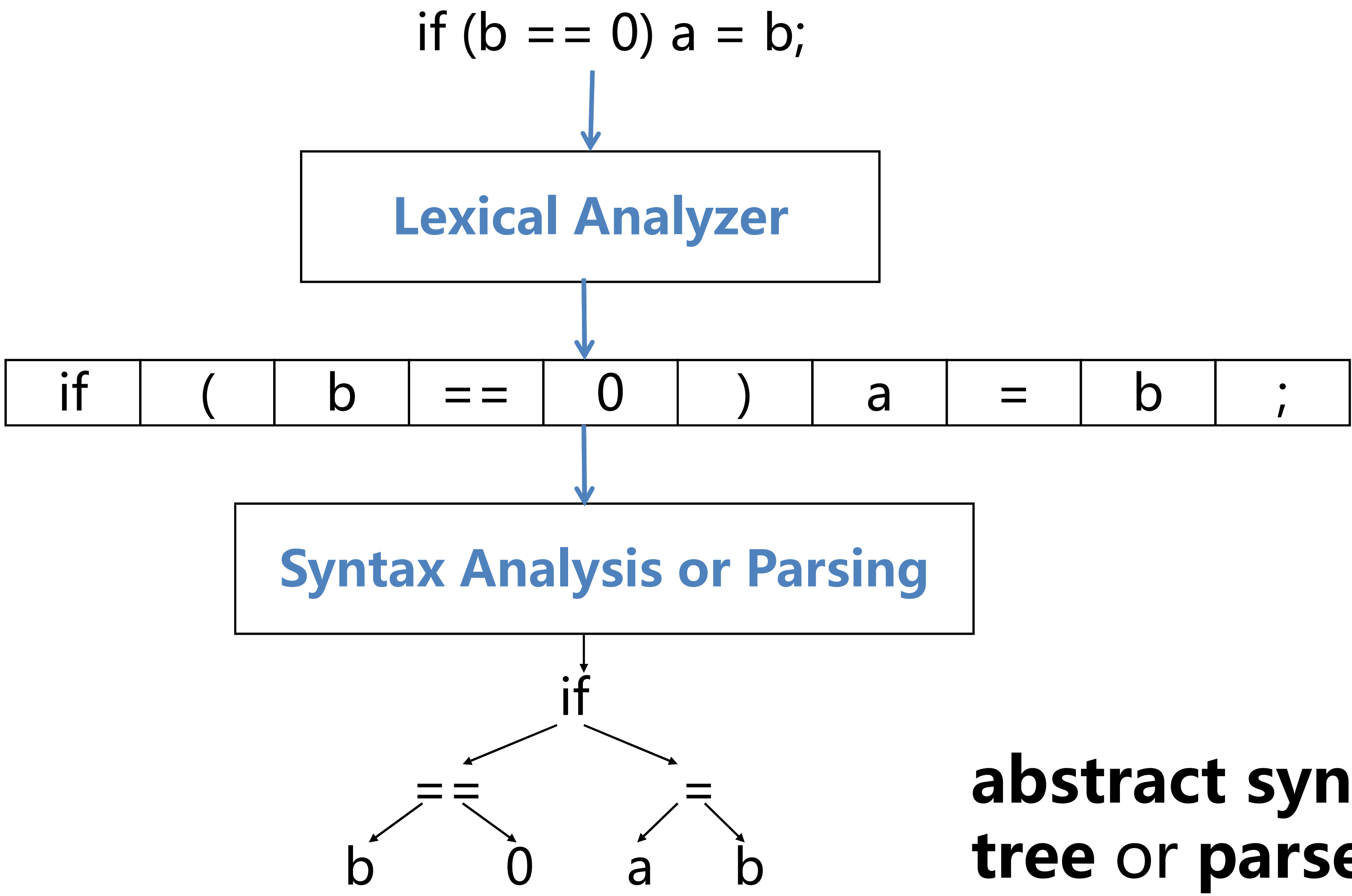
Compilers and Interpreters

Syntax Analysis

Where are we ?


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

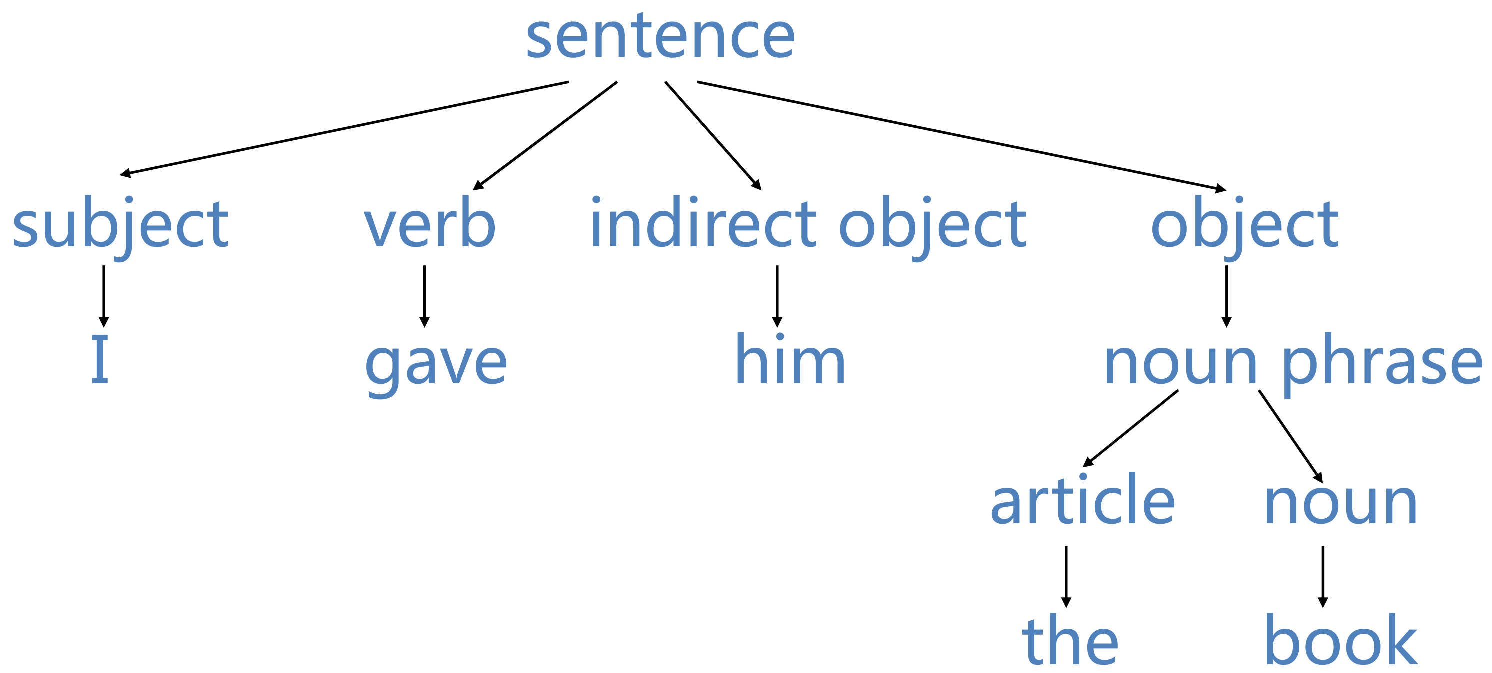
Where is Syntax Analysis Performed?



abstract syntax tree or **parse tree**

Parsing Analogy

- Syntax analysis for natural languages
  - Recognize whether a sentence is grammatically correct
  - Identify the function of each word

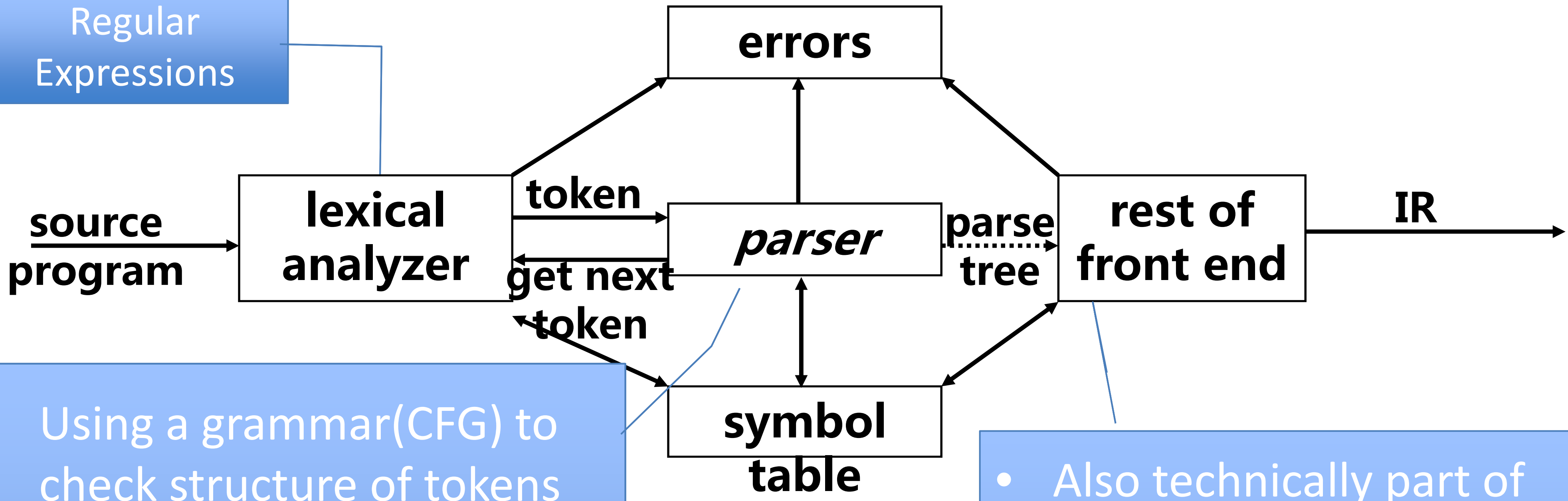


"I gave him the book"

Parsing During Compilation

- Parser works on a stream of tokens.
- The smallest item is a token.

Regular Expressions



- Using a grammar(CFG) to check structure of tokens
- Produce a parse tree
- Recognize correct syntax
- Syntactic errors and recovery
- Reprot errors

- Also technically part of parsing
- Includes augmenting info on tokens in source, type checking, semantic analysis

Error Processing

- Detecting errors
- Finding position at which they occur
- Clear / accurate presentation
- **Recover (pass over) to continue and find later errors**

Syntax Analysis Overview

- Goal – Determine if the input token stream **satisfies syntax** of the program
- What do we need to do this?
  - An expressive way to describe the syntax
  - A mechanism that determines if the input token stream satisfies the syntax description
- For lexical analysis
  - Regular expressions describe tokens
  - Finite automata = mechanisms to generate tokens from input stream

Keywords

- Formalisms for syntax analysis.
  - Context-Free Grammars Derivations
  - Concrete and Abstract Syntax Trees
  - Ambiguity

- Parsing algorithms
  - Top-down Parsing
  - Bottom-up Parsing

Formal Languages

- An alphabet is a set Σ of symbols that act as letters.
- A language over Σ is a set of strings made from symbols in Σ.
- When scanning, the alphabet was ASCII or Unicode characters. We produced tokens.
- When parsing, the alphabet is the set of tokens produced by the scanner.

The Limits of Regular Languages

- When scanning, we used **regular expressions** to define each token.
- Unfortunately, regular expressions are (usually) too weak to define programming languages.
  - Cannot define a regular expression matching all expressions with properly balanced parentheses.
  - Cannot define a regular expression matching all functions with properly nested block structure (blocks, expressions, statements)

We need a more powerful formalism.

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.

Context-Free Grammars

- Inherently **recursive** structures of a programming language are defined by a context-free grammar.
- In a context-free grammar, we have:
  – **A finite set of terminals (in our case, this will be the set of tokens)**
  – **A finite set of non-terminals (syntactic-variables)**
  – **A finite set of productions rules in the following form**
    - **$A \rightarrow \alpha$ where A is a non-terminal and α is a string of terminals and non-terminals (including the empty string)**
  – **A start symbol (one of the non-terminal symbol)**

Example Grammar

$$expr \rightarrow expr\ \ op\ \ expr$$
$$expr \rightarrow (\ expr\ )$$
$$expr \rightarrow -\ expr$$
$$expr \rightarrow id$$
$$op \rightarrow +$$
$$op \rightarrow -$$
$$op \rightarrow *$$
$$op \rightarrow /$$

Black : Nonterminal

Blue : Terminal

expr* : *Start Symbol

8 Production rules

Terminology (cont.)

- L(G) is *the language* of G (the language generated by G) which is a set of sentences.
- A ***sentence*** of L(G) is a string of terminal symbols of G.
- If S is the start symbol of G then

  ω is a sentence of L(G) if $S \stackrel{+}{\Rightarrow} \omega$ where ω is a string of terminals of G.
- A language that can be generated by a grammar is said to be a **context-free language.**
- If G is a context-free grammar, L(G) is a *context-free language*.
- Two grammars are *equivalent* if they produce the same language.
- $S \Rightarrow^* \alpha$
  - If α contains non-terminals, it is called as a ***sentential form*** of G.
  - If α does not contain non-terminals, it is called as a ***sentence*** of G.

EX. $E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+id$

How does this relate to Languages?

Let G be a CFG with start symbol S. Then $S \stackrel{+}{\Rightarrow} W$ (where W has no non-terminals) represents the language generated by G, denoted L(G). So **$W \in L(G) \Leftrightarrow S \Rightarrow^{+} W$**, W is a sentence of G.

When $S \stackrel{*}{\Rightarrow} \alpha$ (and α may have NTs) it is called a **sentential form of G.**

EXAMPLE: ***id * id*** **is a sentence**

Here's the derivation:

$exp \Rightarrow exp \text{ op } exp \Rightarrow exp * exp \Rightarrow \text{id} * exp \Rightarrow \text{id} * \text{id}$

Sentential forms

Sentence

$exp \Rightarrow^{*} id * id$

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

$$A \to \omega$$

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

$$A\mathtt{t}$$

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

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

Derivations

- The central idea here is that a production is treated as a **rewriting rule** in which the non-terminal on the left is replaced by the string on the right side of the production.
- $E \Rightarrow E+E$ E+E derives from E
  - we can replace E by E+E
  - to able to do this, we have to have a production rule $E \rightarrow E+E$ in our grammar.

- $E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+id$
- A sequence of replacements of non-terminal symbols is called a **derivation** of id+id from E.
- In general a derivation step is
- $\alpha_1 \Rightarrow \alpha_2 \Rightarrow ... \Rightarrow \alpha_n$ (α_n derives from α_1 or α_1 derives α_n)

Grammar Concepts

A step in a derivation is zero or one action that replaces a NT with the RHS of a production rule.

EXAMPLE: $E \Rightarrow -E$ (the $\Rightarrow$ means "derives" in one step) using the production rule: $E \rightarrow -E$

EXAMPLE: $E \Rightarrow E \text{ op } E \Rightarrow E * E \Rightarrow E * (E)$

DEFINITION: $\Rightarrow$ derives in one step

$\overset{+}{\Rightarrow}$ derives in $\geq$ one step

$\overset{*}{\Rightarrow}$ derives in $\geq$ zero steps

EXAMPLES: $\alpha A \beta \Rightarrow \alpha \gamma \beta$ if $A \rightarrow \gamma$ is a production rule

$\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$ then $\alpha_1 \Rightarrow^* \alpha_n$;

$\alpha \Rightarrow^* \alpha$ for all α

If $\alpha \Rightarrow \beta$ and $\beta \rightarrow \gamma$ then $\alpha \Rightarrow^* \gamma$

A Notational Shorthand

$$
\begin{aligned}
expr &\rightarrow expr \;\; op \;\; expr \\
expr &\rightarrow (\; expr \;) \\
expr &\rightarrow -\; expr \\
expr &\rightarrow \mathbf{id} \\
op &\rightarrow + \\
op &\rightarrow - \\
op &\rightarrow * \\
op &\rightarrow /
\end{aligned}
$$

$$
\begin{aligned}
expr &\rightarrow expr \;\; op \;\; expr \\
&\quad \mid (\; expr \;) \\
&\quad \mid -\; expr \\
&\quad \mid \mathbf{id} \\
op &\rightarrow + \mid - \mid * \mid /
\end{aligned}
$$

Black : Nonterminal

Blue : Terminal

expr **:** ***Start Symbol***

CFG for Programming Language

program	→	**stmt-sequence**
stmt-sequence	→	**stmt-sequence ; statement**
	|	**statement**
Statement	→	**if-stmt**
	|	**repeat-stmt**
	|	**assign-stmt**
	|	**read-stmt**
	|	**write-stmt**
if-stmt	→	**if exp then stmt-sequence end**
	|	**if exp then stmt-sequence else stmt-sequence end**

CFG vs RE

- The syntax for regular expressions does not carry over to CFGs.
- Cannot use *, |, or parentheses.

$$S \rightarrow \boldsymbol{a^*b}$$

CFG vs RE

- The syntax for regular expressions does not carry over to CFGs.
- Cannot use *, |, or parentheses.

$$S \rightarrow \boldsymbol{A}\boldsymbol{b}$$

CFG vs RE

- The syntax for regular expressions does not carry over to CFGs.
- Cannot use *, |, or parentheses.

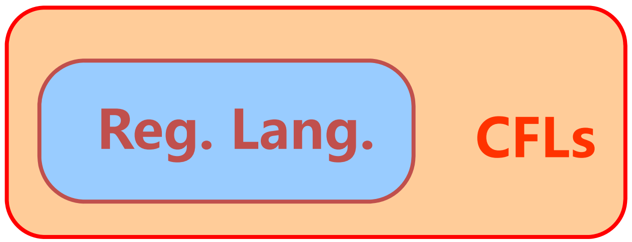
$S \rightarrow a^*b$ $\quad$ $S \rightarrow Ab$

$\qquad\qquad\quad A \rightarrow Aa \mid \varepsilon$

How about a(b|c)*, a*bc* ?

CFG vs RE

- Regular Expressions
  - → Basis of lexical analysis
  - → Represent regular languages
- Context Free Grammars
  - → Basis of parsing
  - → Represent language constructs
  - → Characterize context free languages



Other Derivation Concepts

- At each derivation step, we can choose any of the non-terminal in the sentential form of G for the replacement.
- If we always choose the left-most non-terminal in each derivation step, this derivation is called as **left-most derivation**.
- If we always choose the right-most non-terminal in each derivation step, this derivation is called as **right-most derivation**.

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$$

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+id) \Rightarrow -(id+id)$$

Left-Most and Right-Most Derivations

- Left-Most Derivation
- $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$

- Right-Most Derivation (called *canonical derivation*
- $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+id) \Rightarrow -(id+id)$

- We will see that the ***top-down parsers*** try to find the ***left-most derivation*** of the given source program.
- We will see that the ***bottom-up parsers*** try to find the ***right-most derivation*** of the given source program in the reverse order.

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.

Derivation exercise 1

Productions:

assign_stmt → ***id := expr ;***

expr* → *expr op term

expr* → *term

term* → *id

term* → *real

term* → *integer

***op* → +**

***op* → -**

Let's derive:

id := id + real – integer ;

Please use left-most derivation or right-most derivation.

id := id + real – integer ;

Left-most derivation:

assign_stmt

$\Rightarrow$ *id := expr ;*

$\Rightarrow$ *id := expr op term ;*

$\Rightarrow$ *id := expr op term op term ;*

$\Rightarrow$ *id := term op term op term ;*

$\Rightarrow$ *id := id op term op term;*

$\Rightarrow$ *id := id + term op term ;*

$\Rightarrow$ *id := id + real op term ;*

$\Rightarrow$ *id := id + real - term ;*

$\Rightarrow$ *id := id + real - integer;*

Using production:

assign_stmt $\rightarrow$ *id := expr ;*

expr $\rightarrow$ *expr op term*

expr $\rightarrow$ *expr op term*

expr $\rightarrow$ *term*

term $\rightarrow$ *id*

op $\rightarrow$ *+*

term $\rightarrow$ *real*

op $\rightarrow$ *-*

term $\rightarrow$ *integer*

Parse Trees

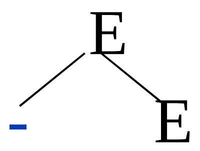
- A parse tree is a tree encoding the steps in a derivation.
- Internal nodes represent nonterminal symbols used in the production.
- Inorder walk of the leaves contains the generated string.
- Encodes what productions are used, not the order in which those productions are applied.

Parse Tree

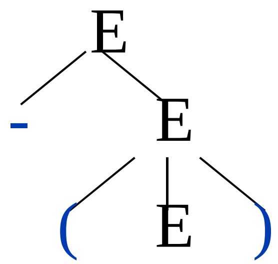
- Inner nodes of a parse tree are non-terminal symbols.
- The leaves of a parse tree are terminal symbols.
- A parse tree can be seen as a graphical representation of a derivation.

EX. $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$

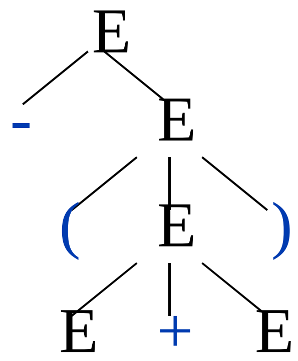
$E \Rightarrow -E$



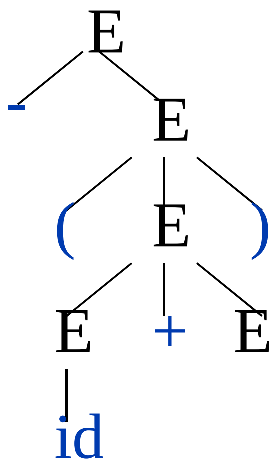
$\Rightarrow -(E)$



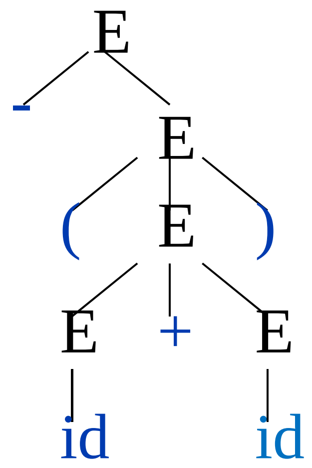
$\Rightarrow -(E+E)$



$\Rightarrow -(id+E)$



$\Rightarrow -(id+id)$



Examples of LM / RM Derivations

$$E \rightarrow E \; op \; E \mid (\, E \,) \mid -E \mid id$$

$$op \rightarrow + \mid - \mid * \mid /$$

A leftmost derivation of : **id + id * id**

A rightmost derivation of **:** **id + id * id**

$E \rightarrow E \; op \; E \mid (E) \mid -E \mid id$

$op \rightarrow + \mid - \mid * \mid /$

$E \Rightarrow E \; op \; E$

$\Rightarrow id \; op \; E$

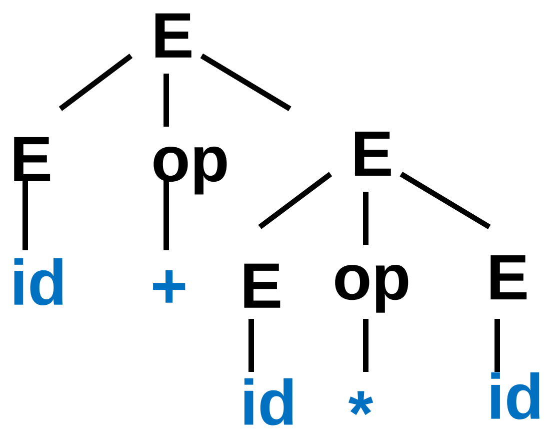
$\Rightarrow id + E$

$\Rightarrow id + E \; op \; E$

$\Rightarrow id + id \; op \; E$

$\Rightarrow id + id * E$

$\Rightarrow id + id * id$



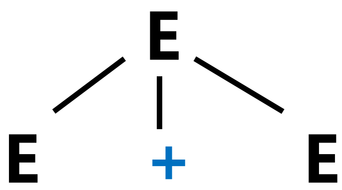
Parse Trees and Derivations

Consider the expression grammar:

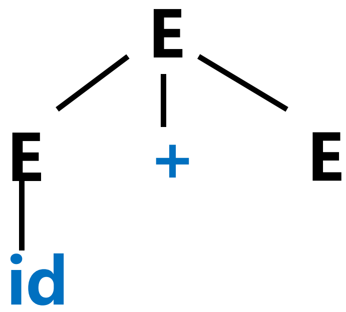
$$E \rightarrow E+E \mid E*E \mid (E) \mid -E \mid id$$

Leftmost derivations of **id + id * id**

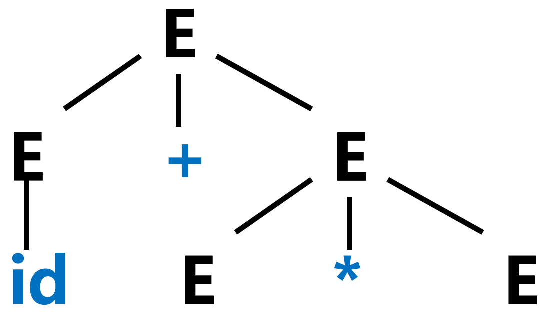
$E \Rightarrow E + E$



$E + E \Rightarrow id + E$

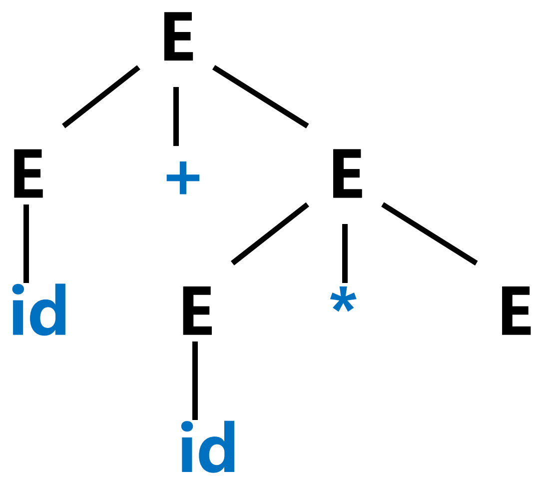


$id + E \Rightarrow id + E * E$

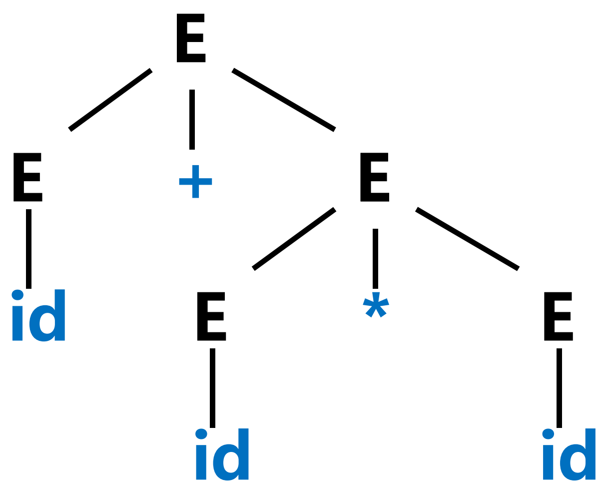


Parse Trees and Derivations (cont.)

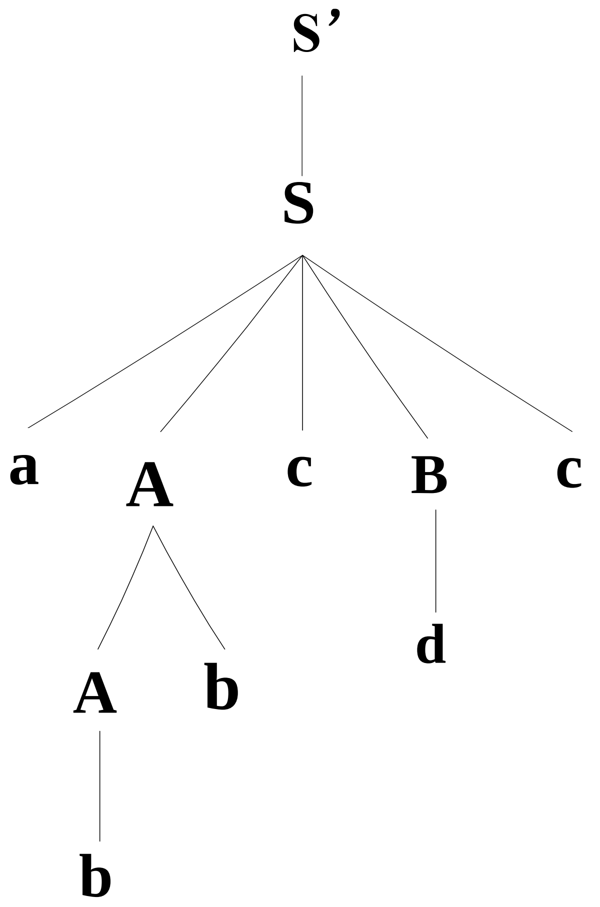
$id + E * E \Rightarrow id + id * E$



$id + id * E \Rightarrow id + id * id$



$S' \rightarrow S$
$S \rightarrow aAcBe$
$A \rightarrow b$
$A \rightarrow Ab$
$B \rightarrow d$



$$a\underline{b}bcde \Leftarrow a\underline{Ab}cde \Leftarrow aAc\underline{d}e \Leftarrow \underline{aAcBe} \Leftarrow S \Leftarrow S'$$

Goal of syntax analysis: Recover the **structure (a parse tree)** described by a series of tokens.

Alternative Parse Tree & Derivation

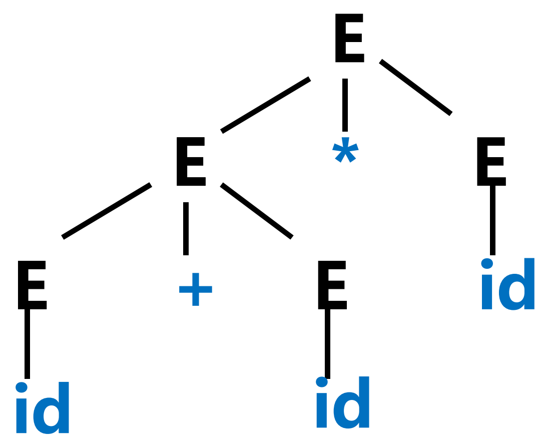
$E \Rightarrow E * E$

$\Rightarrow E + E * E$

$\Rightarrow id + E * E$

$\Rightarrow id + id * E$

$\Rightarrow id + id * id$



WHAT’S THE ISSUE HERE ?

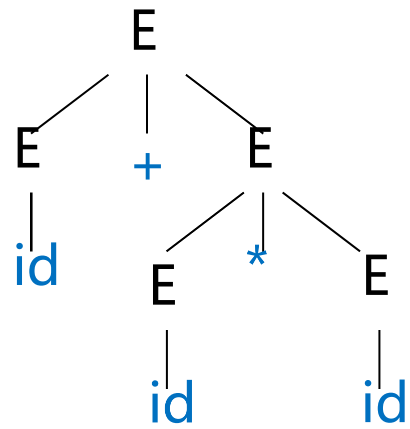
Two distinct leftmost derivations!

Challenges in Parsing

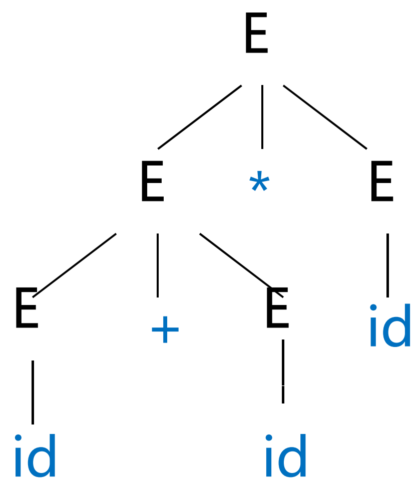
Ambiguity

- A grammar produces more than one parse tree for a sentence is called as an ***ambiguous*** grammar.

$$E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+E*E \Rightarrow id+id*E \Rightarrow id+id*id$$



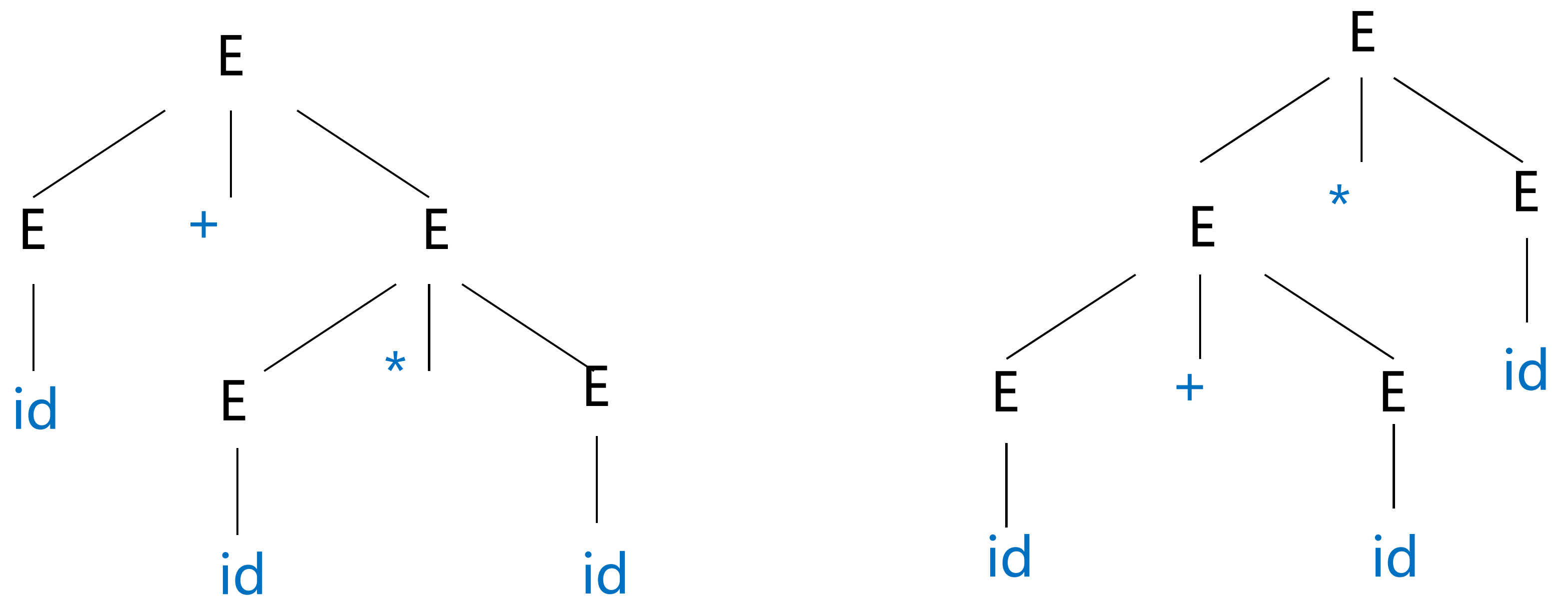
$$E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow id+E*E \Rightarrow id+id*E \Rightarrow id+id*id$$



two parse trees for id+id*id.

Is Ambiguity a Problem?

Depends on semantics.



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?

Ambiguity (cont.)

- For the most parsers, the grammar must be unambiguous.
- **unambiguous grammar**
- → unique selection of the parse tree for a sentence

- We should eliminate the ambiguity in the grammar during the design phase of the compiler.
- An ambiguous grammar should be written to eliminate the ambiguity.
- We have to prefer one of the parse trees of a sentence (generated by an ambiguous grammar) to disambiguate that grammar to restrict to this choice by "throw away" undesirable parse trees.

Ambiguity – Operator Precedence

- Ambiguous grammars (because of ambiguous operators) can be disambiguated according to the **precedence** and **associativity** rules.

- $E \rightarrow E+E \mid E*E \mid E\text{^}E \mid id \mid (E)$
- disambiguate the grammar
- precedence: ^ (right to left)
- * (left to right)
- \+ (left to right)

$$
\begin{aligned}
E &\rightarrow E+T \mid T \\
T &\rightarrow T*F \mid F \\
F &\rightarrow G\text{^}F \mid G \\
G &\rightarrow id \mid (E)
\end{aligned}
$$

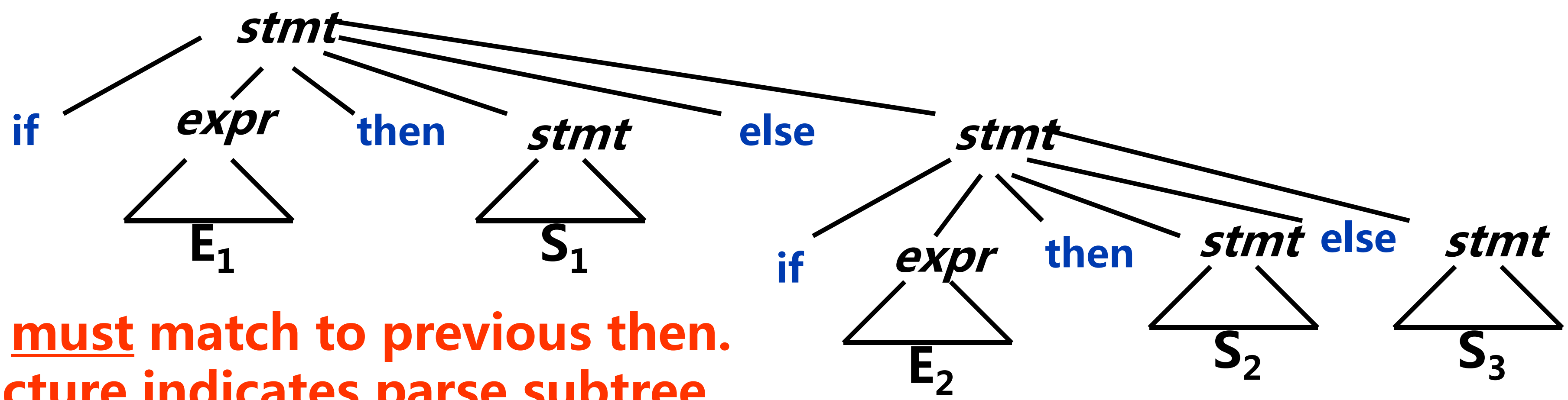
Eliminating Ambiguity

Consider the following grammar segment:

$$stmt \rightarrow \textbf{if } expr \textbf{ then } stmt$$
$$| \ \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt$$
$$| \ \textbf{other} \ \text{(any other statement)}$$

What' s problem here ?

Let' s consider a simple parse tree:



Else <u>must</u> match to previous then. Structure indicates parse subtree for expression.

Example : What Happens with this string?

if E_1 then if E_2 then S_1 else S_2

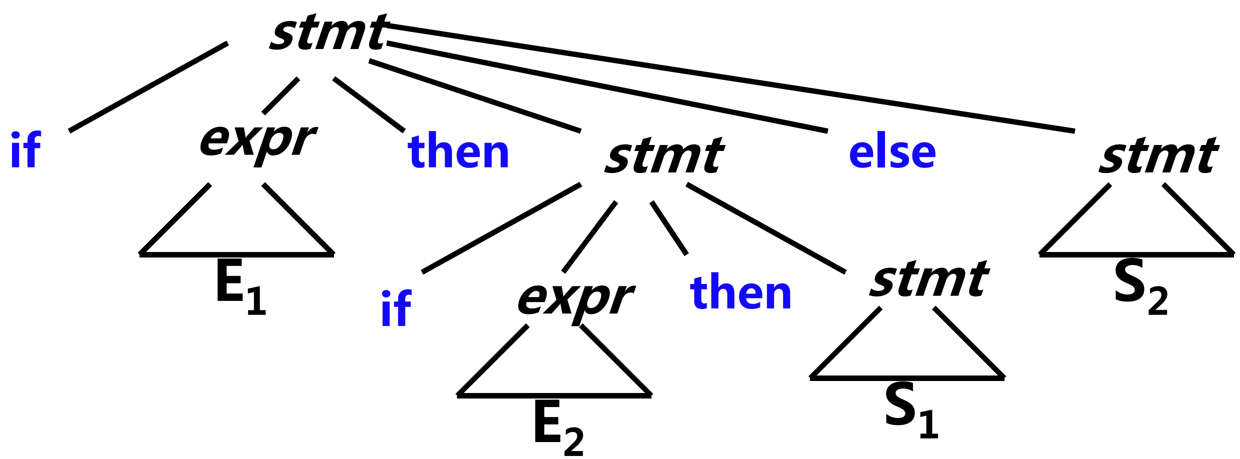
How is this parsed ?

if E_1 then
    if E_2 then
        S_1
else
        S_2
	vs.	if E_1 then
		    if E_2 then
		        S_1
		    else
		        S_2

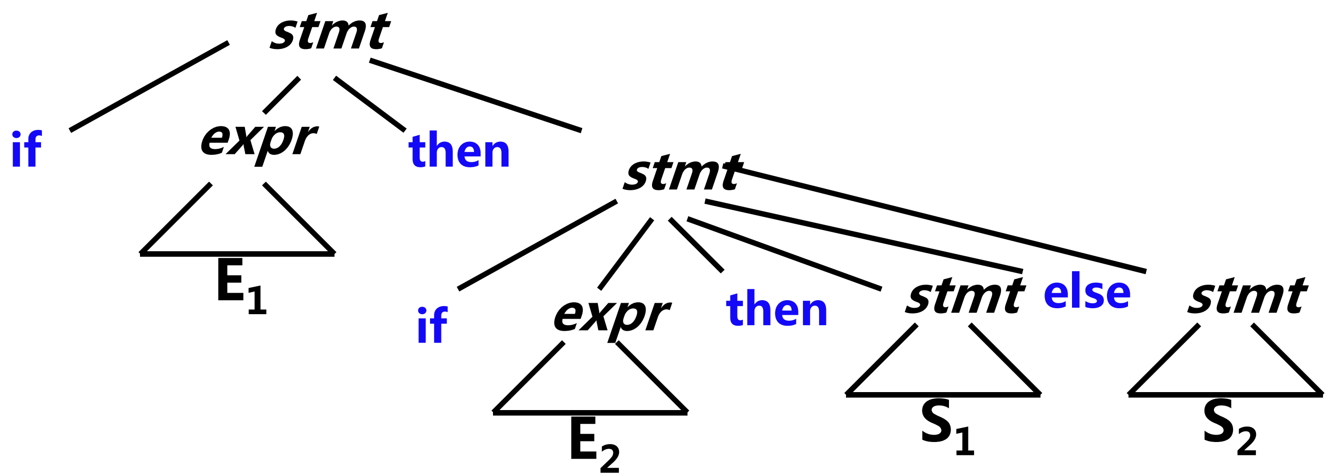
What’ s the issue here ?

Parse Trees for Example

Form 1:



Form 2:



What' s the issue here ?

two parse trees for an ambiguous sentence.

Ambiguity (cont.)

- We prefer the second parse tree (else matches with the closest if).
- So, we have to disambiguate our grammar to reflect this choice.

- The unambiguous grammar will be:

$$
\begin{aligned}
\text{stmt} &\rightarrow \text{matchedstmt} \\
&\mid \text{unmatchedstmt} \\
\text{matchedstmt} &\rightarrow \textbf{if}\ \text{expr}\ \textbf{then}\ \text{matchedstmt}\ \textbf{else}\ \text{matchedstmt} \\
&\mid \text{otherstmts} \\
\text{unmatchedstmt} &\rightarrow \textbf{if}\ \text{expr}\ \textbf{then}\ \text{stmt} \\
&\mid \textbf{if}\ \text{expr}\ \textbf{then}\ \text{matchedstmt}\ \textbf{else}\ \text{unmatchedstmt}
\end{aligned}
$$

The general rule is "match each **else** with the closest previous unmatched **then**."

Ambiguity (cont.)

- A CFG is said to be ambiguous if there is at least one string with two or more parse trees.
- Note that ambiguity is a property of *grammars*, not *languages*.
- There is no algorithm for converting an arbitrary ambiguous grammar into an unambiguous one.

- Some languages are inherently ambiguous, meaning that no unambiguous grammar exists for them.
- There is no algorithm for detecting whether an arbitrary grammar is ambiguous.

Example: Balanced Parentheses

- Consider the language of all strings of balanced parentheses.
- Examples:

ε

`()` `(()())`

`((()))(())()`

- Here is one possible grammar for balanced parentheses:

$$P \rightarrow \varepsilon \mid PP \mid (P)$$

Precedence Declarations

- In the world of pure CFGs, we can often resolve ambiguities through precedence declarations.
  - e.g. multiplication has higher precedence than addition, but lower precedence than exponentiation.
- Allows for unambiguous parsing of ambiguous grammars.

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.
  - Exact order of code execution depends on the parsing method used.
  - This is called a syntax-directed translation.

Simple Semantic Actions

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

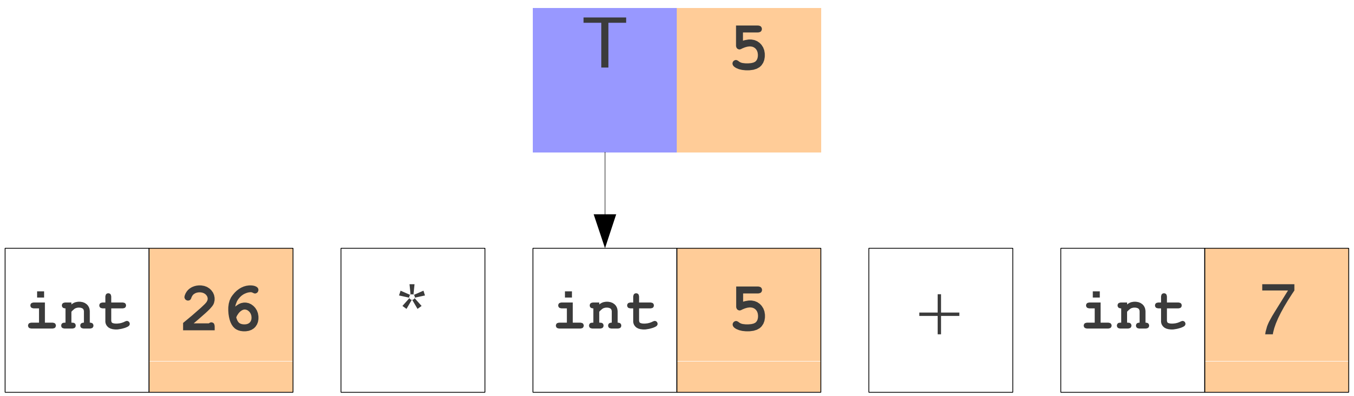
Simple Semantic Actions

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

int	26	*	int	5	+	int	7

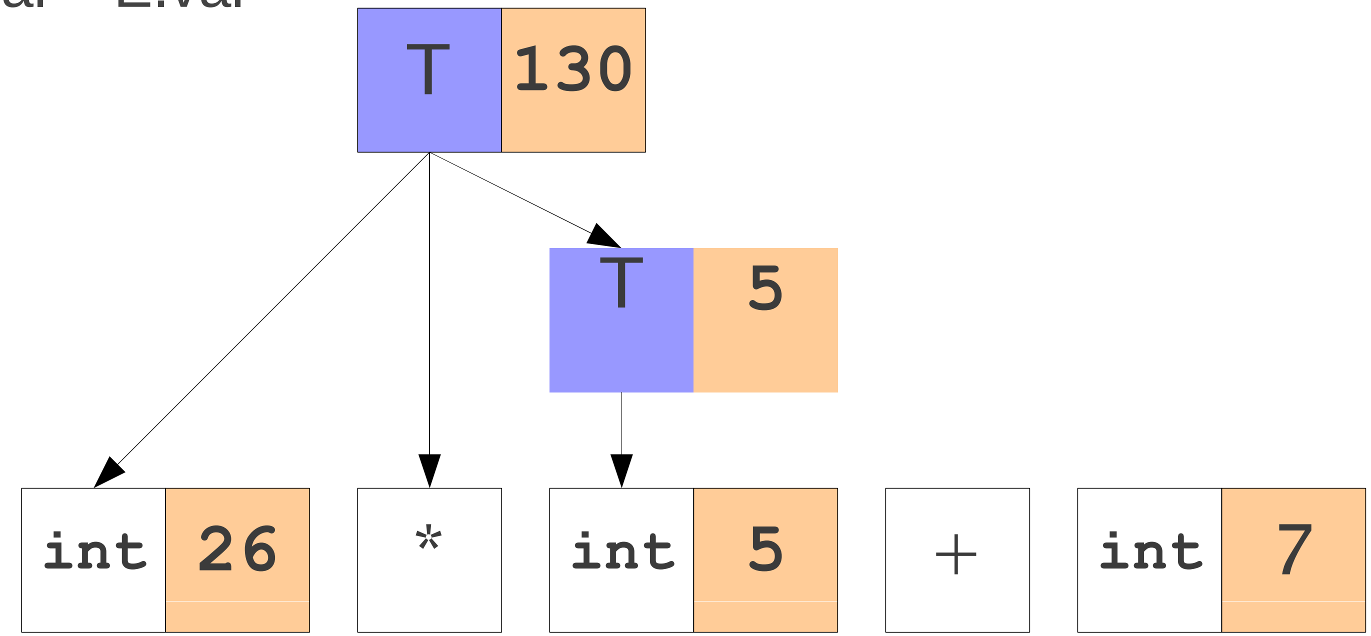
Simple Semantic Actions

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



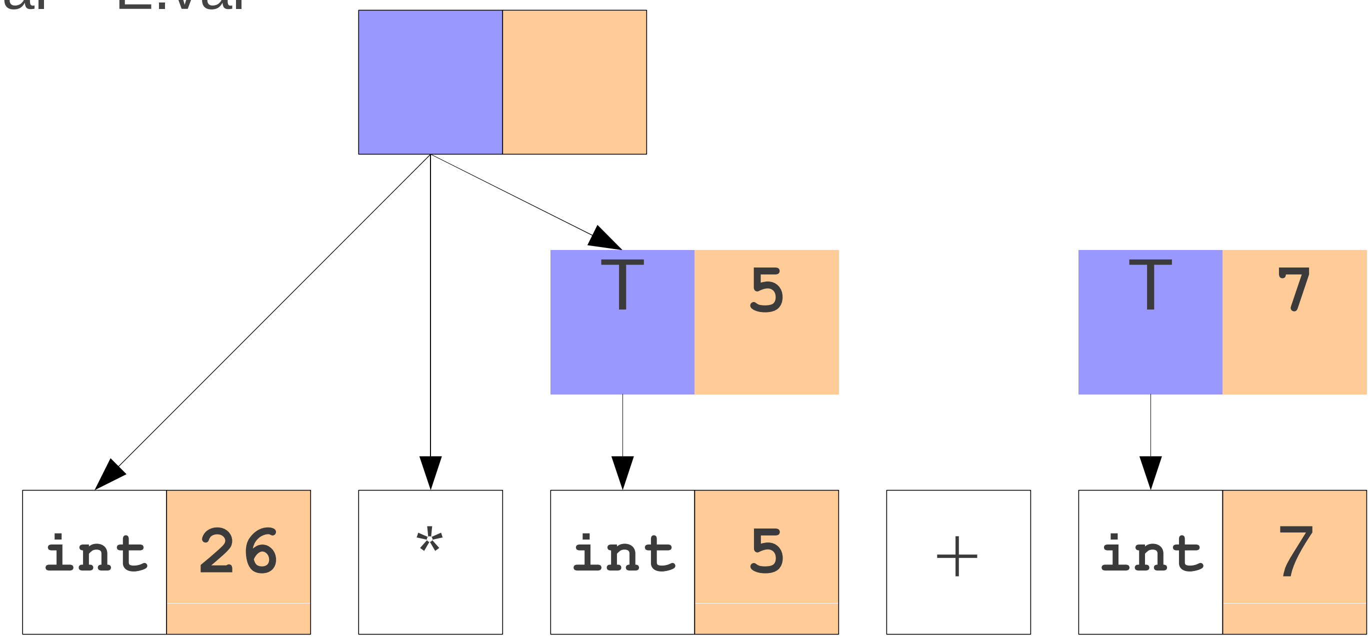
Simple Semantic Actions

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



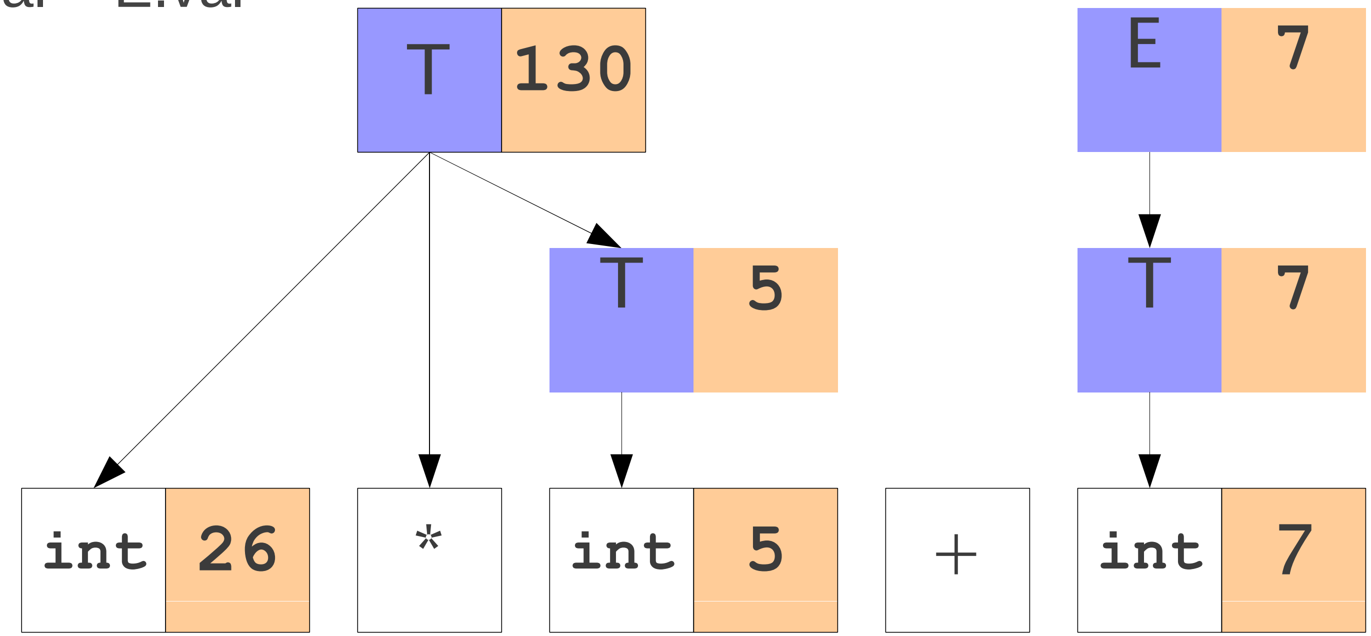
Simple Semantic Actions

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



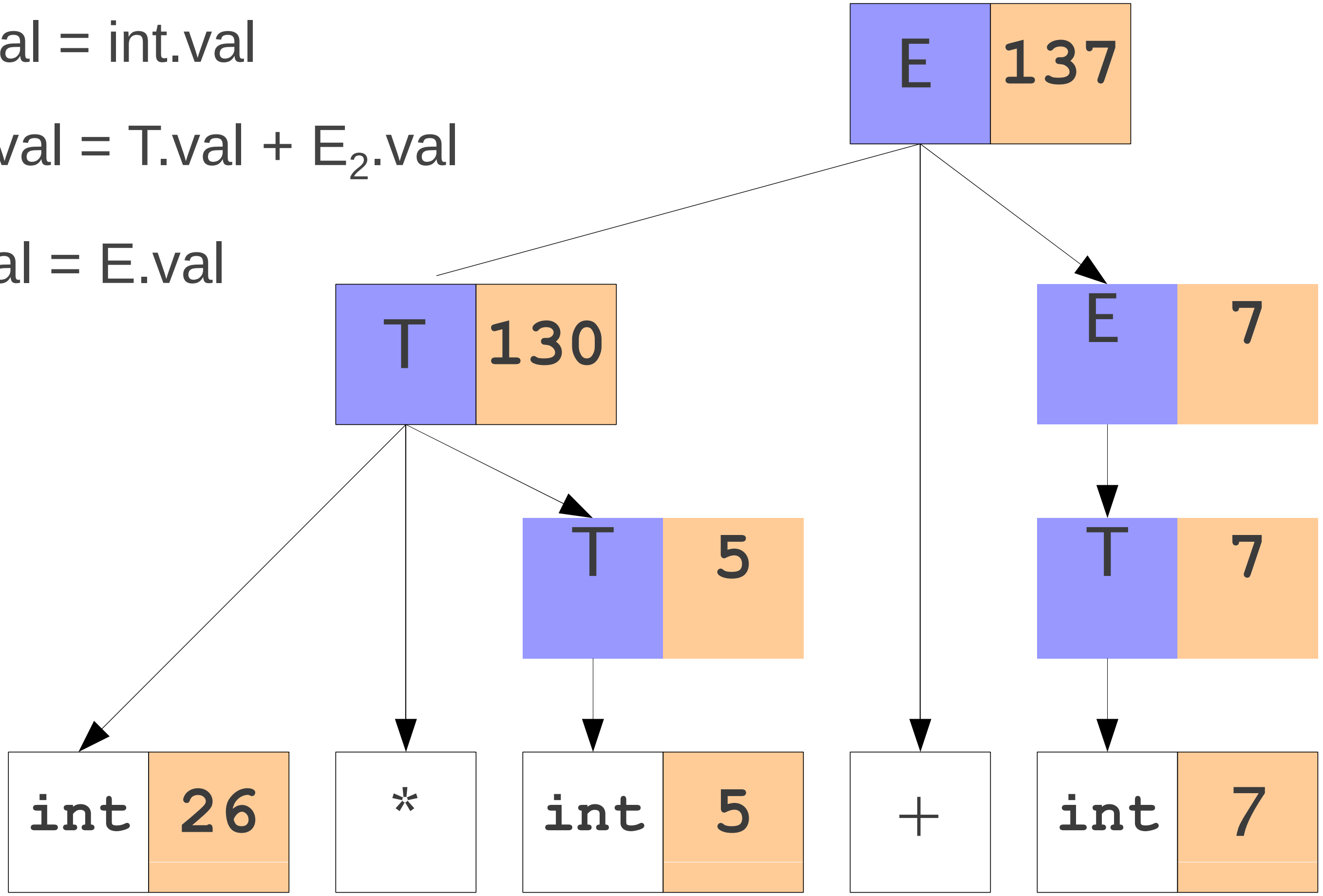
Simple Semantic Actions

E → T + E	E_1.val = T.val + E_2.val
E → T	E.val = T.val
T → **int**	E.val = int.val
T → **int** * T	E_1.val = T.val + E_2.val
T → (E)	T.val = E.val



Simple Semantic Actions

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



Semantic Actions to Build ASTs

R → S	`R.ast = S.ast;`
R → R "|" S	`R₁.ast = new Or(R₂.ast, S.ast);`
S → T	`S.ast = T.ast;`
S → ST	`S₁.ast = new Concat(S₂.ast, T.ast);`
T → U	`T.ast = U.ast;`
T → T*	`T₁.ast = new Star(T₂.ast);`
U → a	`U.ast = new SingleChar('a');`
U → "ε"	`U.ast = new Epsilon();`
U → (R)	`U.ast = R.ast;`

Non-Context Free Language Constructs

- There are some language constructions in the programming languages which are not context-free. This means that, we cannot write a context-free grammar for these constructions.

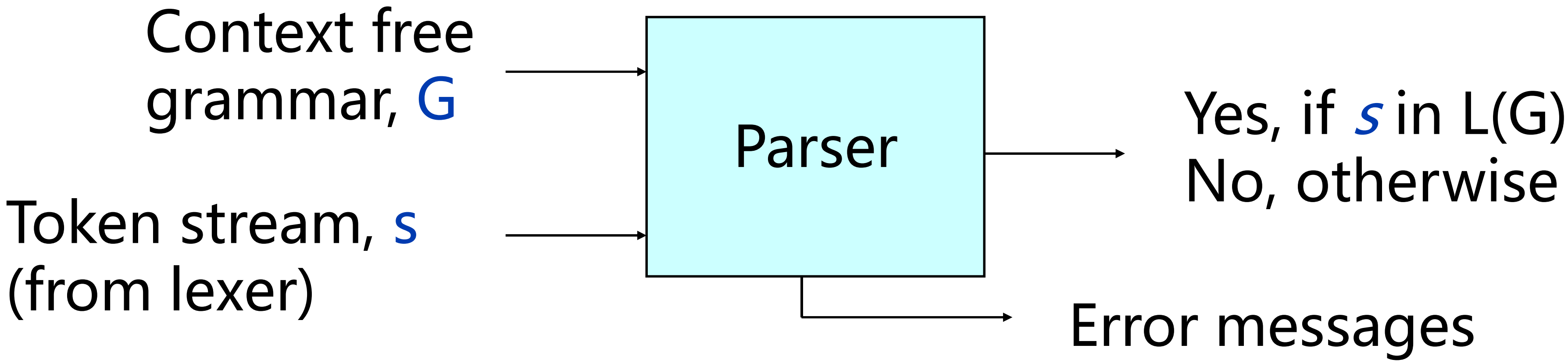
  $L1 = \{ \omega c \omega \mid \omega$ is in $(a|b)^* \}$ is not context-free

  – declaring an identifier and checking whether it is declared or not later. We cannot do this with a context-free language. We need semantic analyzer (which is not context-free).

  $L2 = \{a^n b^m c^n d^m \mid n \geq 1$ and $m \geq 1 \}$ is not context-free

  – declaring two functions (one with n parameters, the other one with m parameters), and then calling them with actual parameters.

A Parser



- Syntax analyzers (parsers) = CFG acceptors which also output the corresponding derivation when the token stream is accepted

- Various kinds: **LL(k), LR(k), SLR, LALR**

Summary

- Languages are usually specified by **context-free grammars (CFGs).**
- Syntax analysis (**parsing**) extracts the structure from the tokens produced by the scanner.
- 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.

Next Time

- Top-Down Parsing
  - Recursive descent parsing
  - Predictive parsing
  - LL(1)
- Bottom-Up Parsing
  - Shift-Reduce Parsing
  - LR parser

Reference

- Compilers Principles, Techniques & Tools (Second Edition) Alfred Aho., Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, Addison-Wesley, 2007
- Coursera Course – Compiler, http://www. Coursera.org
- Stanford Course CS143 by Keith Schwarz, http://cs143.stanford.edu