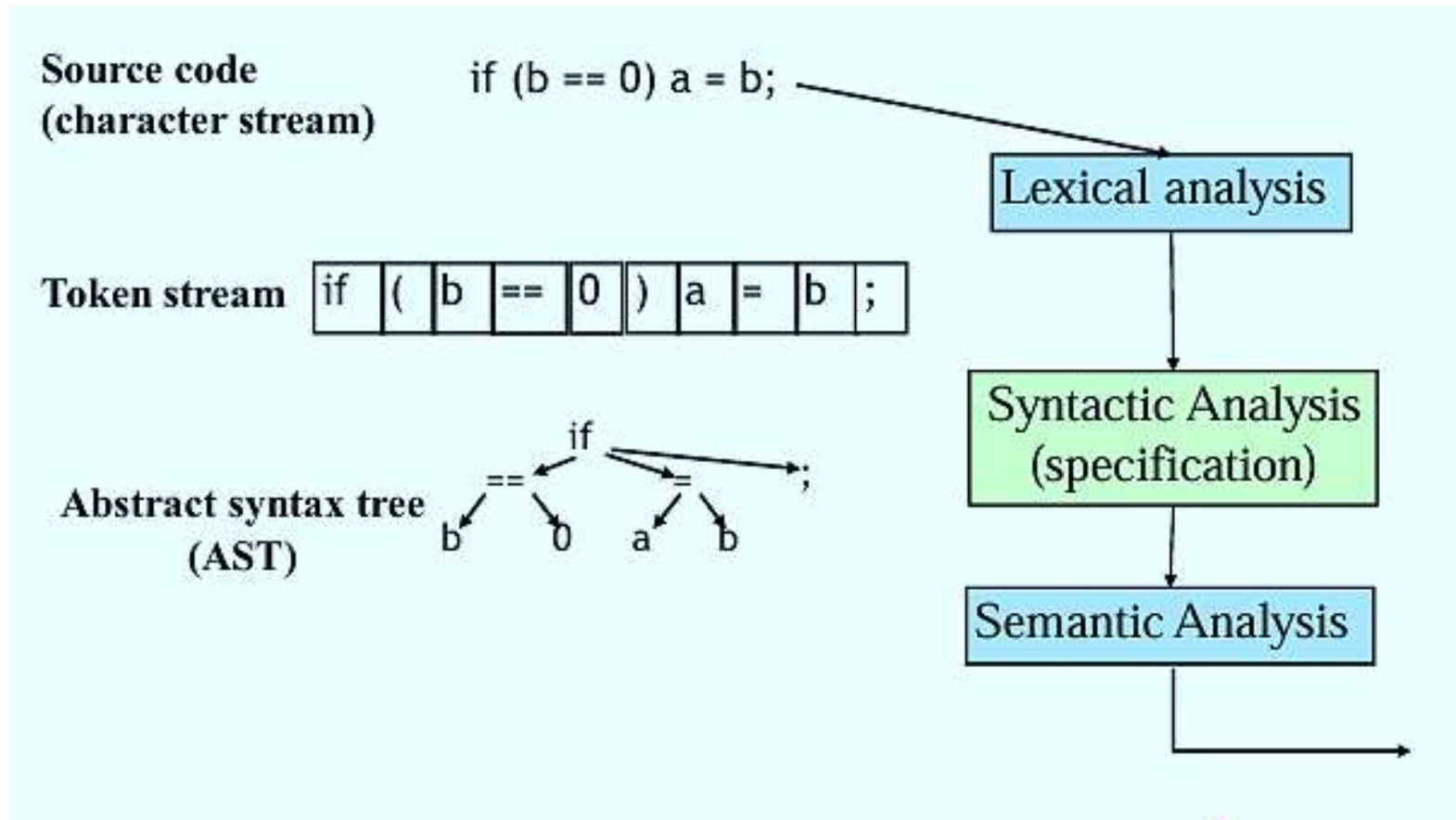


IT 302 Compiler Design

iNTRODUCTION TO pARSING
aMBIGUITY AND sYNTAX eRRORS

Where we are



What is Syntactic Analysis?

- `int a;`
- `a = "hello";`

// Error: assigning string to int

- **Syntax analysis**: Says “grammar is fine”.
- **Semantic analysis**: Says “type mismatch: int variable cannot hold a string”.

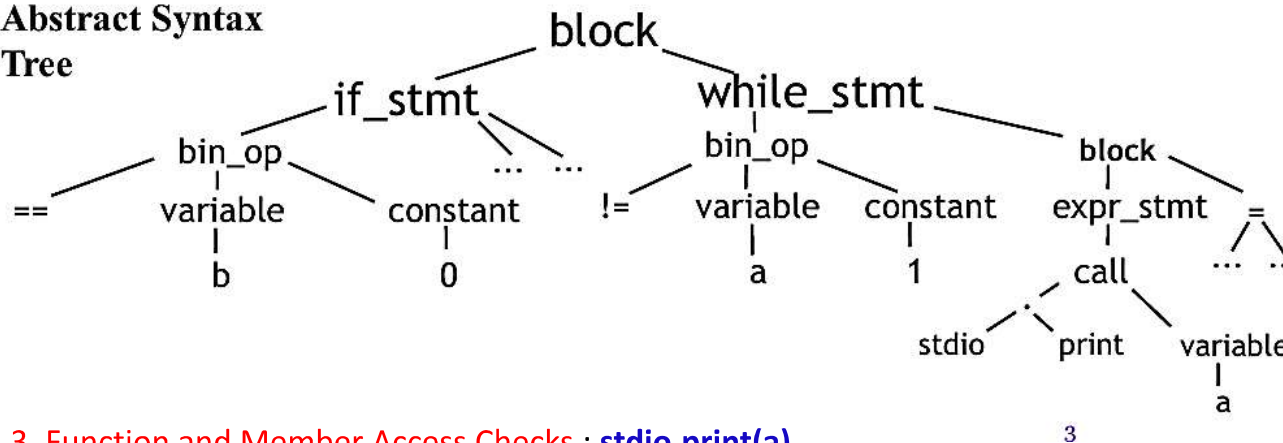
It verifies things such as:

- **Type checking** → Example: You can't add an integer to a string unless the language defines how to do that.
- **Variable declarations** → Every variable must be declared before it is used.
- **Scope rules** → A variable can only be used in the scope it's visible in.
- **Function usage** → The number and types of arguments must match the function definition.
- **Control flow rules** → Statements like **break must be inside loops**, **return must be inside functions**.

What is Syntactic Analysis?

```
Source code      {
(token stream)  if (b == (0)) a = b;
                while (a != 1) {
                  stdio.print(a);
                  a = a - 1;
                }
                }
```

Abstract Syntax Tree



3. Function and Member Access Checks : `stdio.print(a)`

1. `stdio` must be a valid identifier in scope (library/module).
2. `print` must be a valid member/function of `stdio`.
3. Arguments passed (`a`) must match the expected type of `print`.

1. Variable declaration check: (1) Is **b** declared before the if (**b == 0**)?, (2) Is **a** declared before use?

2. Type Checking: (1) `b == 0` → Both sides should be comparable (ex: int with int), (2) `a != 1` → must be comparable types, (3) `a = b;` → If a and b have different types (int vs. string) → **type mismatch error**. (4) `a = a - 1;` → The `-` operator must be valid for the type (numeric).

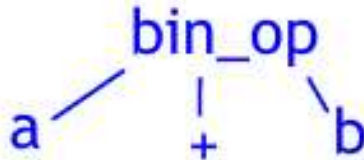
5. **Scope Checking:** Here, **a** is used in both the **if** and **while** — so **a** must be in the outer scope.

4. Control Flow Rules: (1) **if_stmt** → Condition must be a Boolean expression (result of `b == 0` must be Boolean). (2) **while_stmt** → Condition must also be Boolean (`a != 1`).

Syntactic Analysis .. Contd.,

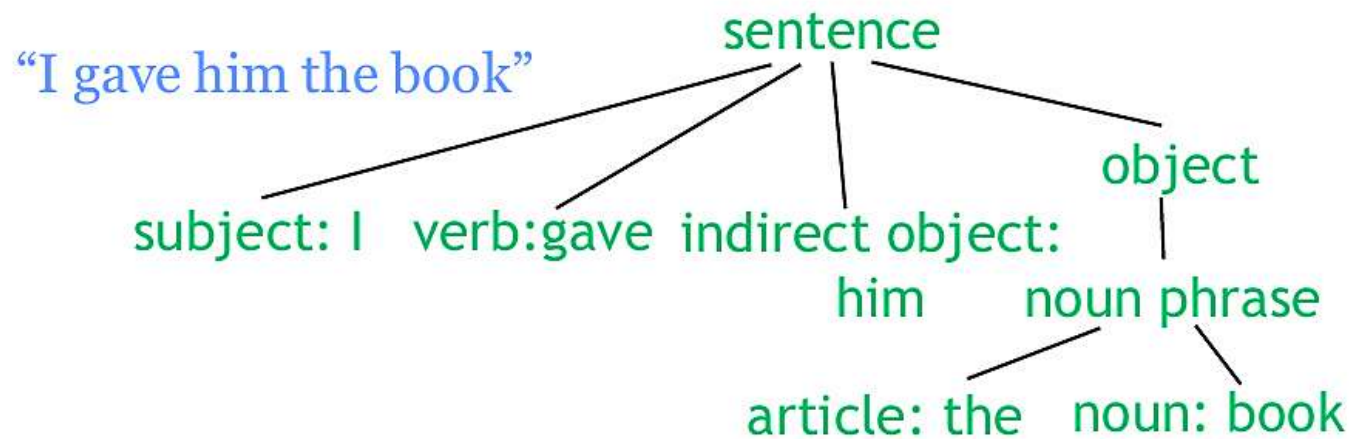
- **Input:** stream of tokens
- **Output:** abstract syntax tree
 - Abstract syntax tree removes extra syntax

$a + b \approx (a) + (b) \approx ((a) + ((b)))$



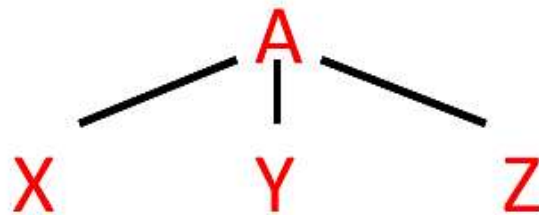
Parsing

- **Parsing:** recognizing whether a program (or sentence) is grammatically well formed & identifying the function of each component.



Parsing Trees

- A parse tree pictorially shows: How the start symbol of a grammar derives strings in language.



1. The root is labeled by start symbol
2. Each leaf is labeled by a terminal (T) or by ϵ
3. Each interior node is labeled by a non-terminals (NT)
4. If NT, X_1, X_2, \dots, X_n are labeled children of A from left to right then there must be production $A \rightarrow X_1 X_2 \dots X_n$, where X_1, X_2, \dots, X_n are either NT or T,
5. If $A \rightarrow \epsilon$, then A may have single child ϵ

Parsing Trees: Properties

- A tree consists of one or more nodes
- Exactly one root node in a Tree
 - Root have no-parent, it is top node
 - Other node have exactly one parent
- Leaf: node with no children
 - N is parent of M,
 - M is child of N,
 - Children of one node is Siblings,
 - Ordered from left to right
 - Descendent (self, child*), Ancestor (self, parent*)

The Functionality of the Parser

- **Input:** sequence of tokens from lexer
- **Output:** parse tree of the program

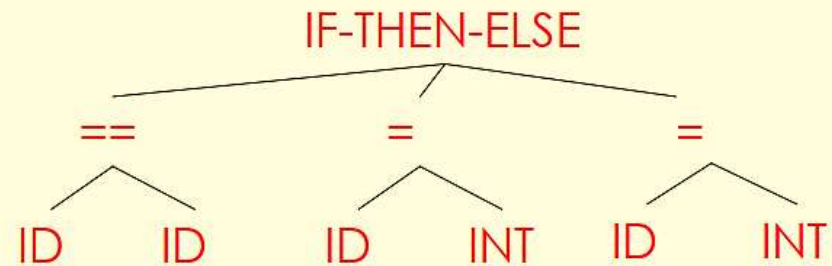
- If-then-else statement

if (x == y) then z = 1; else z = 2;

- Parser input

IF (ID == ID) THEN ID = INT; ELSE ID = INT;

- Possible parser output



<i>Phase</i>	<i>Input</i>	<i>Output</i>
Lexer	Sequence of characters	Sequence of tokens
Parser	Sequence of tokens	Parse tree

What Parsing doesn't do

- Doesn't check many things: type agreement, variables declared, variables initialized, etc.
 - `int x = true;`
 - `int y; z = f(y);`
- Postponed until semantic analysis

Parsing \neq Full Program Correctness

- The parser's job is **only** to check whether the program's structure matches the grammar rules — the "shape" of the code.
- It does not:
 - Check if types match (**type agreement**).
 - Verify if variables are declared before use.
 - Ensure variables are initialized before being read.

Recap: Languages and Automata

- Formal languages are very important in CS
 - Especially in programming languages
- Regular languages
 - The weakest formal languages widely used
 - Many applications

Specifying Language Syntax

- **First problem:** how to describe language syntax precisely and conveniently
- **Last time:** can describe tokens using regular expressions
- Regular expressions easy to implement, efficient (by converting to DFA)
- **Why not use regular expressions (on tokens) to specify programming language syntax?**
 - Programming languages need **nested, recursive structures**, but **regular expressions can't handle recursion**.

if (x) { y = 1; } else { z = 2; }

This involves:

- Matching **balanced parentheses {} or ()**.
- Handling **nested if-statements**.
- Matching **expressions** that can contain subexpressions.
- Regular languages **cannot** match:
 - **Balanced brackets:** { { } { { } } }

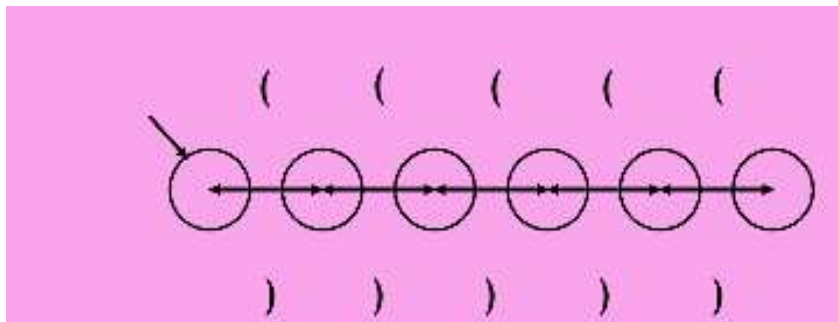
Limitations of Regular Languages

- A finite automaton that runs long enough must repeat states
 - A finite automaton cannot remember # of times it has visited a particular state
 - Because a finite automaton has finite memory
 - Only enough to store in which state it is
 - Cannot count, except up to a finite limit
 - Many languages are not regular
 - **Ex:** language of balanced parentheses is not regular: $\{ (i) \mid i \geq 0 \}$

Problem: need to keep track of number of parentheses seen so far: *unbounded counting*

Contd.,

- RE = DFA
- DFA has only finite number of states; cannot perform unbounded counting



maximum depth: 5 parenthesis

The Role of the Parser

- Not all sequences of tokens are programs . . .
- . . . Parser must distinguish between valid and invalid sequences of tokens
- We need
 - A language for describing valid sequences of tokens
 - A method for distinguishing valid from invalid sequences of tokens

Regexes

Compiler
Writer



Scotty, we need more power!

Context-Free Grammars

- Many programming language constructs have a recursive structure
- A **STMT** is of the form
 - if COND then STMT else STMT , or
 - while COND do STMT , or
 - ...
- Context-free grammars are a natural notation for this recursive structure

CFGs (Cont.)

- A CFG consists of
 - A set of terminals **T**
 - A set of non-terminals **N**
 - A start symbol **S** (a non-terminal)
 - A set of productions

Assuming $X \in N$ the productions are of the form

$$X \rightarrow \varepsilon$$

, or

$$X \rightarrow Y_1 Y_2 \dots Y_n$$

where $Y_i \in N \cup T$

Contd., Let's begin by defining the parts of a CFG

- A **terminal** is a **discrete symbol** that can appear in the language, other wise known as a token.
 - Examples of terminals are **keywords, operators, and identifiers**.
 - We will use *lower-case letters to represent terminals*.
- A **non-terminal** represents a **structure that can occur in a language**, but is not a literal symbol.
 - Example of non-terminals are **declarations, statements, and expressions**.
 - We will use *upper-case letters to represent non-terminals*: P for program, S for statement, E for expression, etc.
- A **sentence** \rightarrow a valid sequence of terminals in a language, while a **sentential form** \rightarrow a valid sequence of terminals and non-terminals.
 - Greek symbols to represent sentential forms. Ex: **α , β , γ** and represent (possibly) **mixed sequences of terminals and non-terminals**.
 - We will use a sequence like **$Y_1, Y_2, \dots Y_n$** to indicate the individual symbols in a sentential form: **Y_i may be either a terminal or a non-terminal**.

Contd., Let's begin by defining the parts of a CFG

- A **context-free grammar (CFG)** is a list of rules that formally describe the allowable sentences in a language.
 - The **left-hand side** of each rule is always a single non-terminal.
 - The **right-hand side** of a rule is a sentential form that describes an allowable form of that non-terminal.

Ex: the rule $A \rightarrow xXy$

- indicates the non-terminal A represents a terminal x followed by a non-terminal X and a terminal y.
 - The right-hand side of a rule can be indicated ϵ that the rule produces nothing.
- **The first rule is special**: it is the top-level definition of a program and its non-terminal known as start symbol.

For example, here is a simple CFG that describes expressions involving addition, integers, and identifiers:

1. $P \rightarrow E$
2. $E \rightarrow E + E$
3. $E \rightarrow \text{ident}$
4. $E \rightarrow \text{int}$

- This grammar can be read as follows:
 - (1) A complete program consists of one expression.
 - (2) An expression can be any expression plus any expression.
 - (3) An expression can be an identifier.
 - (4) An expression can be an integer.
- *For shortness:* a common left-hand side by combining all of the right-hand sides with a logical-or symbol, like this:

$$E \rightarrow E + E \mid \text{ident} \mid \text{int}$$

Notational Conventions

- In this Parsing
 - Non-terminals are written upper-case
 - Terminals are written lower-case
 - The start symbol is the left-hand side of the first production

Examples of CFGs

- A fragment of our example language (simplified)

```
STMT → if COND then STMT else STMT
      | while COND do STMT
      | id = int
```

Grammar for simple
arithmetic expressions

```
E → E * E
   | E + E
   | ( E )
   | id
```

The Language of a CFG

- Read productions as replacement rules

$$X \rightarrow Y_1 \dots Y_n$$

Means X can be replaced by $Y_1 \dots Y_n$

$$X \rightarrow \varepsilon$$

Means X can be erased (replaced with empty string)

Key Idea

- (1) Begin with a string consisting of the start symbol “S”
- (2) Replace any non-terminal X in the string by a right-hand side of some production
- (3) Repeat (2) until there are no non-terminals in the string

The Language of a CFG (Cont.)

Describing **formal derivations** in **context-free grammars** in simpler terms:

More formally, we write

$$X_1 \cdots X_i \cdots X_n \rightarrow X_1 \cdots X_{i-1} Y_1 \cdots Y_m X_{i+1} \cdots X_n$$

if there is a production

$$X_i \rightarrow Y_1 \cdots Y_m$$

Write

$$X_1 \cdots X_n \xrightarrow{*} Y_1 \cdots Y_m$$

if

$$X_1 \cdots X_n \rightarrow \cdots \rightarrow \cdots \rightarrow Y_1 \cdots Y_m$$

in 0 or more steps

- **In simple words:**

\rightarrow means “one step of replacement”

\rightarrow^* means “any number of steps of replacement” (including zero)

The Language of a CFG

- Let **G** be a context-free grammar with start symbol **S**. Then the language of **G** is:

$$\left\{ a_1 \dots a_n \mid S \xrightarrow{*} a_1 \dots a_n \text{ and every } a_i \text{ is a terminal} \right\}$$

- This is the **language defined by the grammar**:

$L(G) = \{\text{all terminal strings derivable from the start symbol}\}$ Where **G** is your CFG, **S** is its start symbol.

- This condition ensures that the derived string contains **no nonterminals left** — it's a complete sentence in the language.

Terminals

- Terminals are tokens and there are no rules for replacing them
- Once generated, terminals are permanent

Examples

$L(G)$ is the language of the CFG G



Strings of balanced parentheses $\{(^i)^i \mid i \geq 0\}$

Two grammars:

$$\begin{array}{lcl} S \rightarrow (S) & & S \rightarrow (S) \\ S \rightarrow \varepsilon & \text{OR} & | \varepsilon \end{array}$$

CFG for Arithmetic Example

Simple arithmetic expressions:

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

Some elements of the language:

id		id + id
(id)		id * id
(id) * id		id * (id)

We have one non-terminal: **E** (stands for “expression”)

Production rules:

$E \rightarrow E + E$	// addition
$E \rightarrow E * E$	// multiplication
$E \rightarrow (E)$	// parentheses
$E \rightarrow id$	// identifier (variable or number)

context-free grammar for **simple arithmetic expressions** and some example strings it can generate.

Derivations and Parse Trees

- A derivation is a sequence of productions

$S \rightarrow \dots \rightarrow \dots \rightarrow \dots$

- A derivation can be drawn as a tree
 - Start symbol is the tree's root
- For a production $X \rightarrow Y_1 \dots Y_n$ add children $Y_1 \dots Y_n$ to node X

Derivation Example

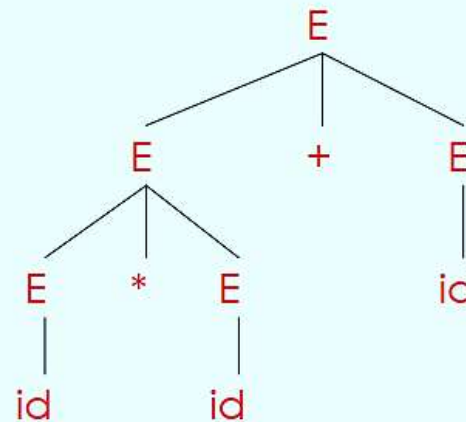
- Grammar

$$E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$$

- String

$$\text{id} * \text{id} + \text{id}$$

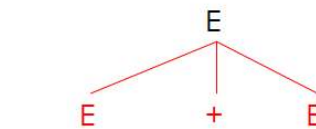
E
 $\rightarrow E + E$
 $\rightarrow E * E + E$
 $\rightarrow \text{id} * E + E$
 $\rightarrow \text{id} * \text{id} + E$
 $\rightarrow \text{id} * \text{id} + \text{id}$



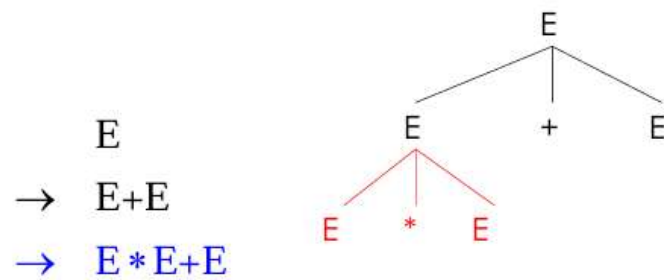
Derivation in Detail



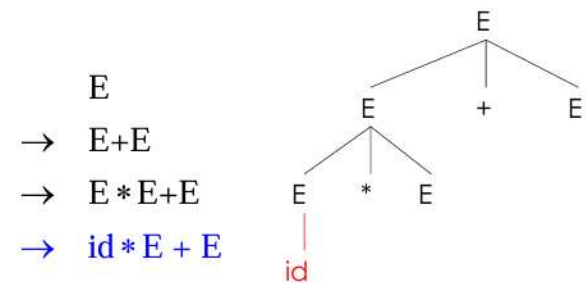
Step 1



Step 2

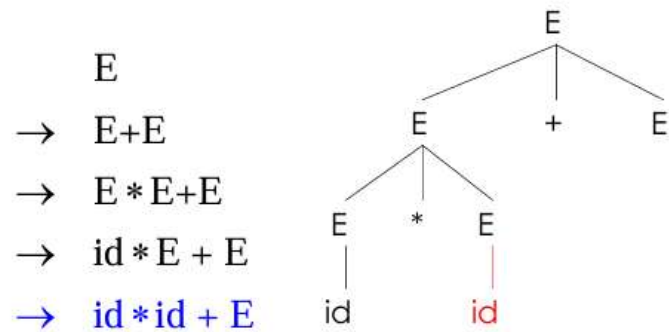


Step 3

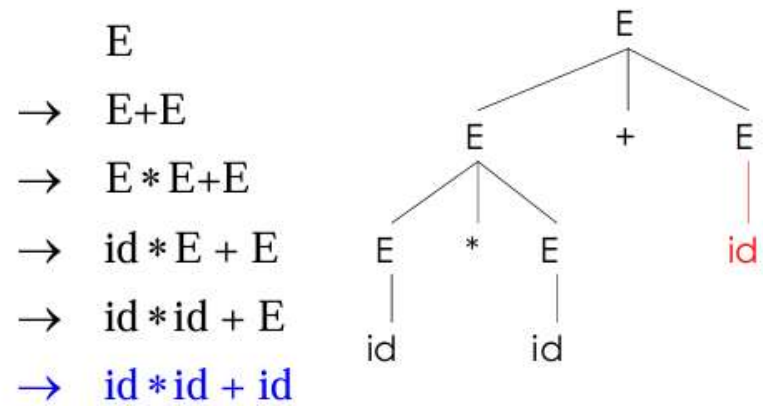


Step 4

Derivation in Detail .. Contd.,



Step 5



Step 6

Notes on Derivation

- A parse tree has
 - Terminals at the leaves
 - Non-terminals at the interior nodes
- An in-order traversal of the leaves is the original input
- The parse tree shows the association of operations; the input string does not

Left-most and Right-most Derivations

- The example is a *left-most derivation*

- At each step, replace the left-most non-terminal

- There is an equivalent notion of a *right-most derivation*

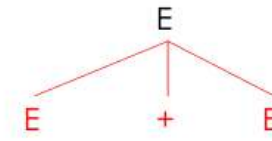
$$\begin{aligned} & E \\ \rightarrow & E + E \\ \rightarrow & E + id \\ \rightarrow & E * E + id \\ \rightarrow & E * id + id \\ \rightarrow & id * id + id \end{aligned}$$

Right-most Derivation in Detail

E

E

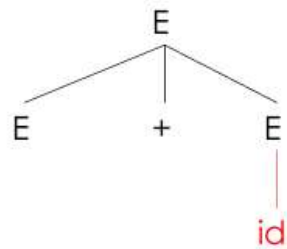
E
→ E+E



Step 1

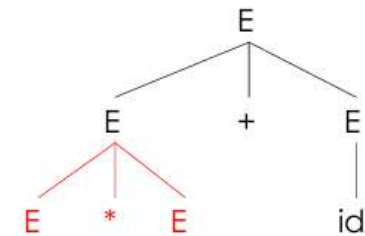
Step 2

E
→ E+E
→ E+id



Step 3

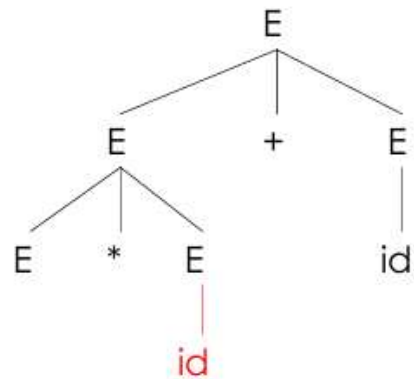
E
→ E+E
→ E+id
→ E * E + id



Step 4

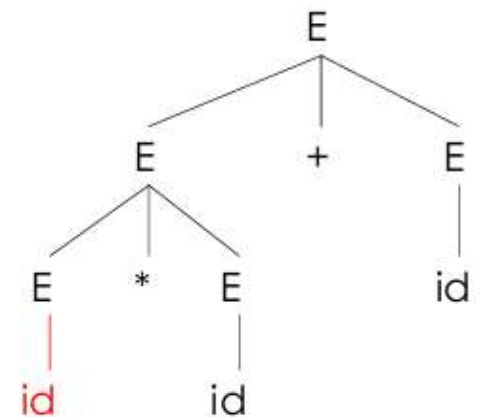
Contd.,

E
 $\rightarrow E + E$
 $\rightarrow E + id$
 $\rightarrow E * E + id$
 $\rightarrow E * id + id$



Step 5

E
 $\rightarrow E + E$
 $\rightarrow E + id$
 $\rightarrow E * E + id$
 $\rightarrow E * id + id$
 $\rightarrow id * id + id$



Step 6

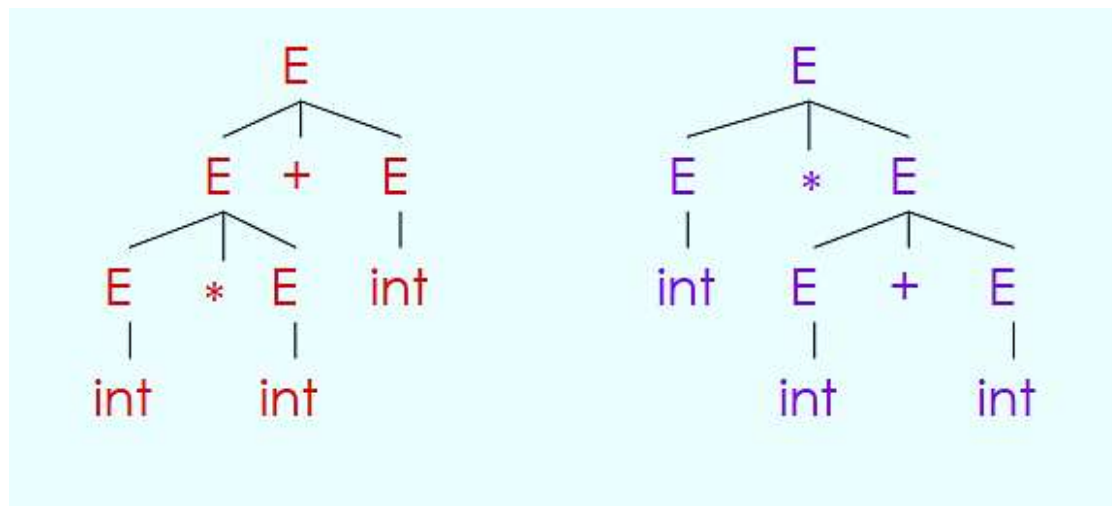
Derivations and Parse Trees

- **Note** that right-most and left-most derivations have the same parse tree
- The difference is just *in the order in which branches are added*
- We are not just interested in whether $s \in L(G)$
 - We need a parse tree for s
- A derivation defines a parse tree
 - But one parse tree may have many derivations
- Left-most and right-most derivations are important in parser implementation

Ambiguity

- Grammar: $E \rightarrow E + E \mid E * E \mid (E) \mid \text{int}$
- String: `int * int + int`

This string has two parse trees



Ambiguity (Cont.)

- A grammar is **ambiguous** if it has more than one parse tree for some string
 - Equivalently, there is more than one right-most or left-most derivation for some string
- **Ambiguity is bad**
 - Leaves meaning of some programs ill-defined
- **Ambiguity is common in programming languages**
 - Arithmetic expressions
 - IF-THEN-ELSE

Dealing with Ambiguity

- There are several ways to handle ambiguity
- Most direct method is to **rewrite grammar unambiguously**

$$\begin{aligned} E &\rightarrow T + E \mid T \\ T &\rightarrow \text{int} * T \mid \text{int} \mid (E) \end{aligned}$$

Enforces precedence of * over +

Ambiguity: The Dangling Else

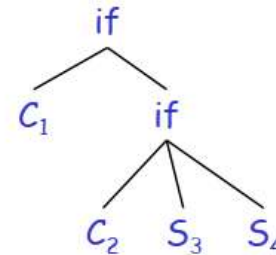
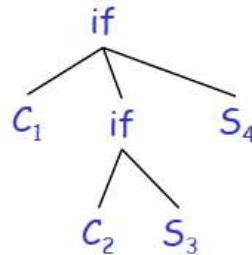
- Consider the following grammar

$S \rightarrow \text{if } C \text{ then } S$
| $\text{if } C \text{ then } S \text{ else } S$
| OTHER

This grammar is also ambiguous

The Dangling Else: Example

- The expression
if C_1 then if C_2 then S_3 else S_4
has two parse trees



Typically, we want the second form