SyntaxAnalysis

Dr. S. Suresh
Assistant Professor
Department of Computer Science
Banaras Hindu University
Varanasi – 211 005, India.

Syntax Analyzer

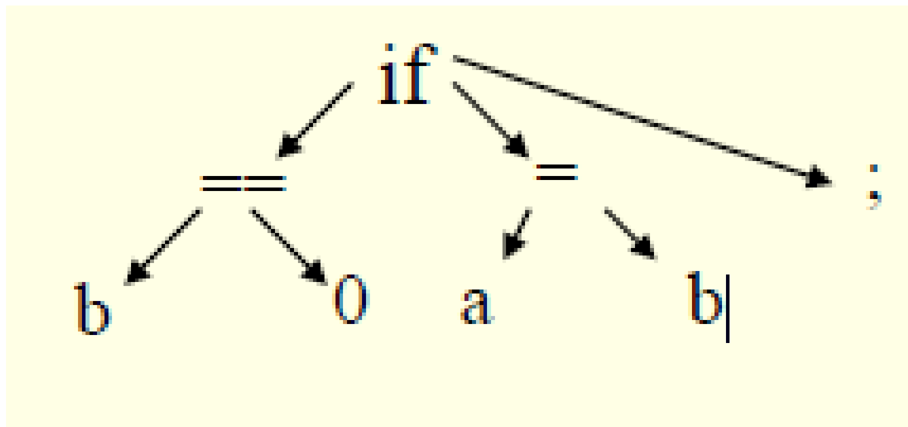
- Input: Sequence of Tokens

- Output: a representation of program
  - Often AST, but could be other things

- Error reporting and recovery
- Model using context free grammars
- Recognize using Push down automata/Table Driven Parsers

Syntax Analyzer

if	(	b	==	0	)	a	=	b	;



- Check syntax and construct abstract syntax tree and Error reporting

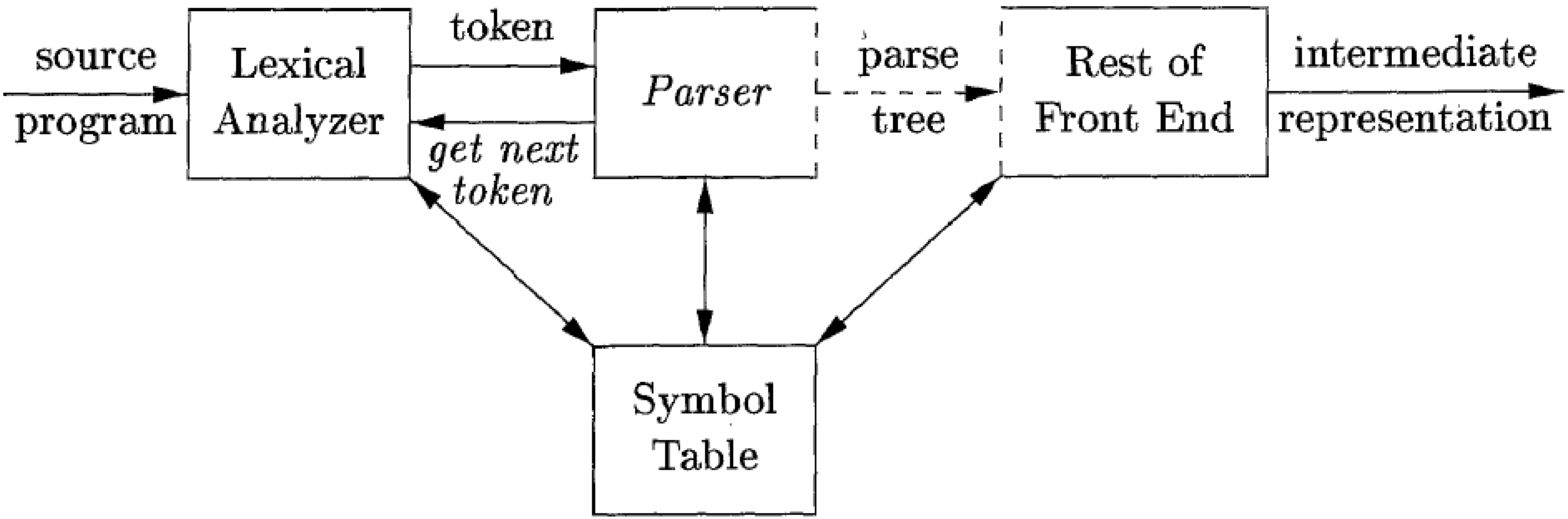
Benefits of Grammars

- Offers benefit for both language designers and compiler writers.
- Gives a precise, yet easy-to-understand, syntactic specification of a language
- For certain classes of grammars, parser can be constructed automatically
- Parser construction process can reveal syntactic ambiguities and trouble spots in the initial design of a language
- Useful for translating source programs into correct object code and for detecting errors
- Allows a language can to be evolved or developed iteratively

The role of the Parser

- Obtains a string of tokens from lexical analyzer
- Verifies that the string of token names can be generated by the grammar for the source language and constructs the parse tree
- Report syntax errors
- Recovers from commonly occurring errors to continue processing the remainder of the program
- Parsers can be combined with other phases of front end since they interacts often.

Position of parser in compiler model



Types of Parsers

- Universal
  - Cocke-Younger-Kasami algorithms
  - Earley’s algorithm
  - Can parse any grammar
  - Too inefficient to use in production compilers
- Top-down
  - Builds parse tree from the top (root) to the bottom (leaves)
- Bottom-up
  - Builds parse tree from the bottom (leaves) to the top (root)
- Both top-down and bottom-up case, input is scanned from left to right, one symbol at a time

Limitations of regular languages

- How to describe language syntax precisely and conveniently. Can regular expressions be used?
- Many languages are not regular, for example, string of balanced parentheses
  - ((((…))))
  - $\{ (i)^i \mid i \geq 0 \}$
  - There is no regular expression for this language
- A finite automata may repeat states, however, it cannot remember the number of times it has been to a particular state
- A more powerful language is needed to describe a valid string of tokens

Context Free Grammars (CFGs)

- Context free grammars <T, N, P, S>
  - T: a set of tokens (terminal symbols)
  - N: a set of nonterminal symbols
  - P: a set of productions or rule of the form

    nonterminal →String of terminals & non terminals
  - S: a start symbol
- A grammar derives strings by beginning with a start symbol and repeatedly replacing a nonterminal by the right hand side of a production for that non terminal.
- The strings that can be derived from the start symbol of a grammar G form the language L(G) defined by the grammar.

Context Free Grammars (CFGs)

- *Terminals* are the basic symbols from which strings are formed.
- "token name" is a synonym for "terminal"
- *Nonterminals* are syntactic variables that denote the sets of strings
- *Nonterminals* impose a hierarchical structure on language that is key to syntax analysis and translation
- One *nonterminal* is distinguished as start symbol and the set of strings it denotes is the language generated by the grammar

Notational Conventions

- Terminals
  – lowercase letters,
  – operators symbols (eg. +, -, *, etc.),
  – digits,
  – punctuation symbols, etc.

- Nonterminals
  – uppercase letters,
  – letters S for start symbol

Examples

- Grammar for arithmetic expressions
  – terminals are ***id + - * / ( )***
  – nonterminals are ***expression, term*** and ***factor***
  – start symbol is ***expression***
- Grammar

expression	→ *expression + term*
expression	→ *expression – term*
expression	→ *term*
term	→ *term * factor*
term	→ *term / factor*
term	→ *factor*
factor	→ *(expression)*
factor	→ *id*

Examples

- String of balanced parentheses

  $S \rightarrow (S) S \mid \epsilon$

- Grammar

  list $\rightarrow$ list + digit

  | list – digit

  | digit

  digit $\rightarrow$ 0 | 1 | … | 9

- Consists of the language which is a list of digit separated by + or -.

Examples

list $\rightarrow$ <u>list</u>+ digit

$\rightarrow$ <u>list</u>–digit + digit

$\rightarrow$ <u>digit</u>–digit + digit

$\rightarrow$ 9 –<u>digit</u>+ digit

$\rightarrow$ 9 –5 + <u>digit</u>

$\rightarrow$ 9 –5 + 2

- Therefore, the string 9-5+2 belongs to the language specified by the grammar
- The name context free comes from the fact that use of a production X $\rightarrow$ … does not depend on the context of X

Examples

- Simplified Grammar for C block

  block → ‘{‘decls statements‘}’

  statements → stmt-list | Є

  stmt–list → stmt-list stmt ‘;’

  | stmt‘;’

  decls → declsdeclaration | Є

  declaration → …

Syntax analyzers

- Testing for membership whether *w* belongs to L(G) is just a “yes” or “no” answer
- However the syntax analyzer
  – Must generate the parse tree
  – Handle errors gracefully if string is not in the language
- Form of the grammar is important
  – Many grammars generate the same language
  – Tools are sensitive to the grammar

What syntax analysis cannot do!

- To check whether variables are of types on which operations are allowed
- To check whether a variable has been declared before use
- To check whether a variable has been initialized
- These issues will be handled in semantic analysis

Derivation

- If there is a production $A \longrightarrow \alpha$ then we say that A derives α and is denoted by $A \Rightarrow \alpha$
- $\Rightarrow$ means "derives in one step"
- $\alpha A \beta \Rightarrow \alpha \gamma \beta$ if $A \Rightarrow \gamma$ is a production
- If $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$ rewrites α_1 to α_n we say that α_1 derives α_n (derives in zero or more steps)
- $\Rightarrow^*$ means "derives in one or more steps"
- $\Rightarrow^+$ means "derives in one or more steps"

Derivation

- Thus $\alpha \Rightarrow^* \alpha$, for any string α, and
- If $\alpha \Rightarrow^* \beta$ and $\beta \Rightarrow \delta$, then $\alpha \Rightarrow^* \delta$
- Given a grammar G and a string w of terminals in $L(G)$ we can write $S \Rightarrow^+ w$
- If $S \Rightarrow^* \alpha$ where α is a string of terminals and non terminals of G then we say that α is a sentential form of G
- Sentential form may contain both terminal and nonterminals and may be empty

Derivation

- Consider the following grammar

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

- The string *- (id + id)* is a sentence of grammar because

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

- E, - E, -(E), -(E + E), -(id + E), -(id + id) are the sentential form of this grammar

Derivation

- If in a sentential form only the leftmost non terminal is replaced then it becomes leftmost derivation
- Every leftmost step can be written as

  $$wA\gamma \Rightarrow^{lm*} w\delta\gamma$$

  where w is a string of terminals, $A \to \delta$ is a production and γ is a string of grammar symbols
- Similarly, rightmost derivation, left-sentential and right-sentential are also can be defined
- An ambiguous grammar is one that produces more than one leftmost(rightmost) derivation of a sentence

Parse Tree

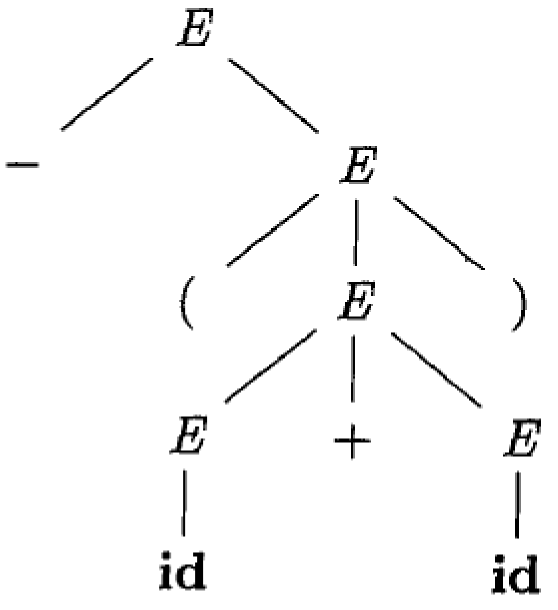
- It is a graphical representation of a derivation that filters out the order in which productions are applied to replace nonterminals
- Shows how the start symbol of a grammar derives a string in the language
- Root is labeled by the start symbol
- Leaf nodes are labeled by tokens
- Each internal node is labeled by a non terminal

Parse Tree

- If A is the label of a node and $x_1, x_2, \ldots x_n$ are labels of the children of that node then $A \rightarrow x_1 x_2 \ldots x_n$ is a production in the grammar
- To see the relationship between derivations and parse trees, consider any derivations $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$ where α_1 is a single nonterminal A.
- For each sentential form α_i in a derivation, we can construct a parse tree whose yield is α_i
- The process is an induction on *i*.

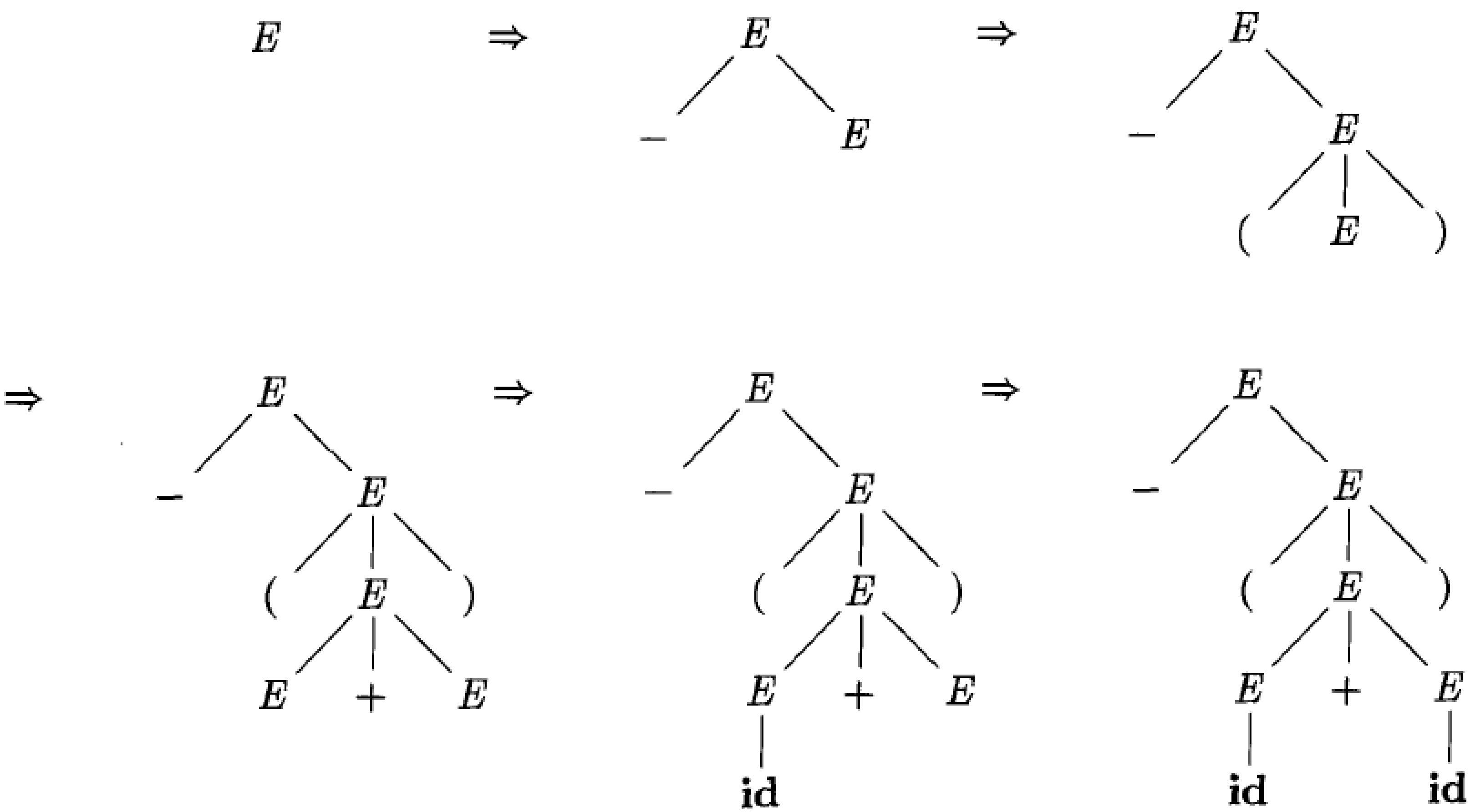
Example

- Parse tree for - (id + id)

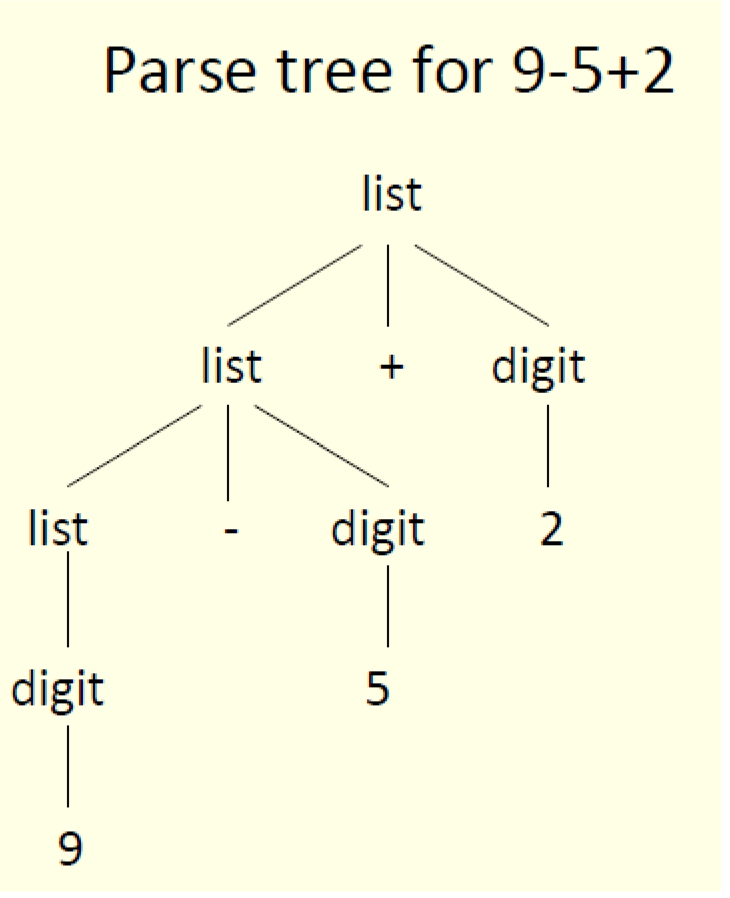


Example

- Sequence of parse trees



Example



Ambiguity

- A Grammar that produces more than one parse tree for some sentence is said ambiguous.

- Consider grammar

  list → list+ list

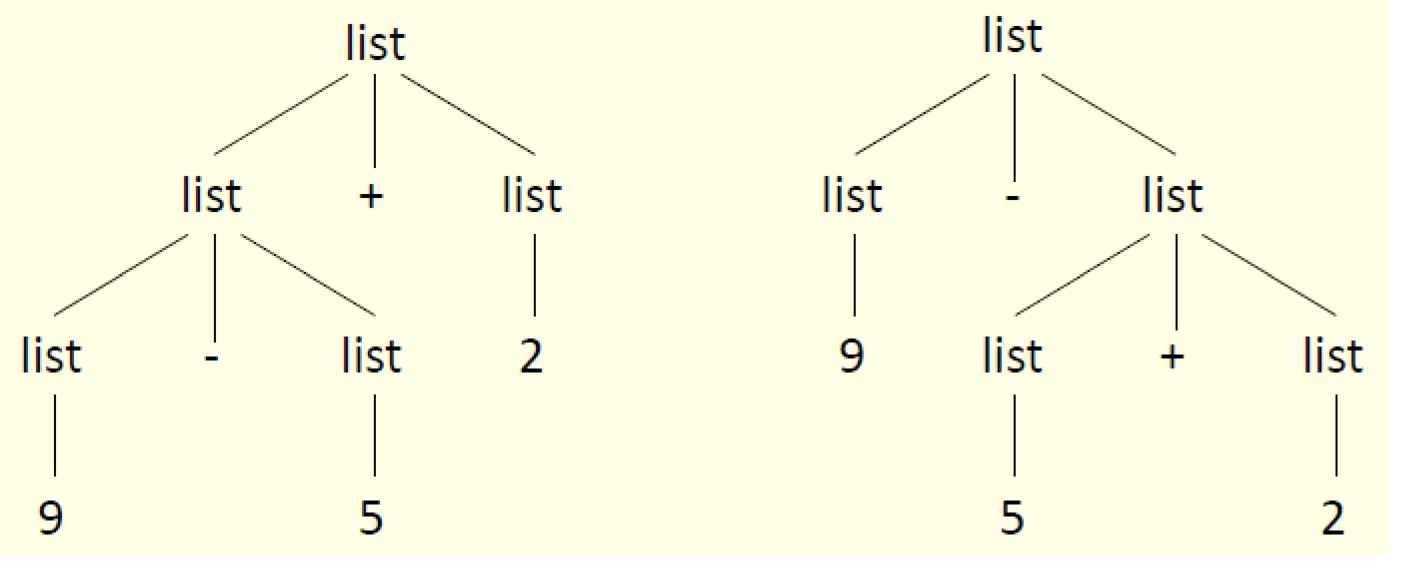
  | list–list

  | 0 | 1 | … | 9

- String 9-5+2 has two parse trees

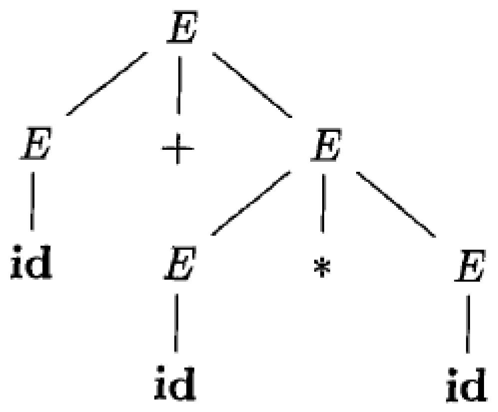
Example

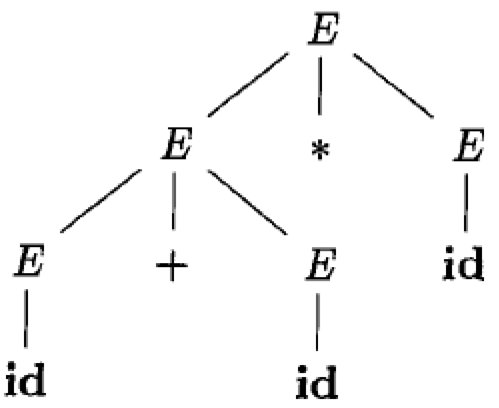
- String 9-5+2 has two parse trees



Example

- String id + id * id has two parse trees





Ambiguity …

- Ambiguity is problematic because meaning of the programs can be incorrect
- Ambiguity can be handled in several ways
  – Enforce associativity and precedence
  – Rewrite the grammar (cleanest way)
- There is no algorithm to convert automatically any ambiguous grammar to an unambiguous grammar accepting the same language
- Worse, there are inherently ambiguous languages!

Ambiguity …

- For most parsers, it is desirable that the grammar be made unambiguous
- Otherwise, we can’t uniquely determine the parse tree
- However, it is convenient to use carefully chosen ambiguous grammars with disambiguating rules that through away undesirable parse trees, leaving only one tree for each sentence

Ambiguity in Programming Lang.

- Dangling else problem

  stmt $\rightarrow$ if expr stmt

  | if expr stmtelse stmt

  | other

  other means any other statemant

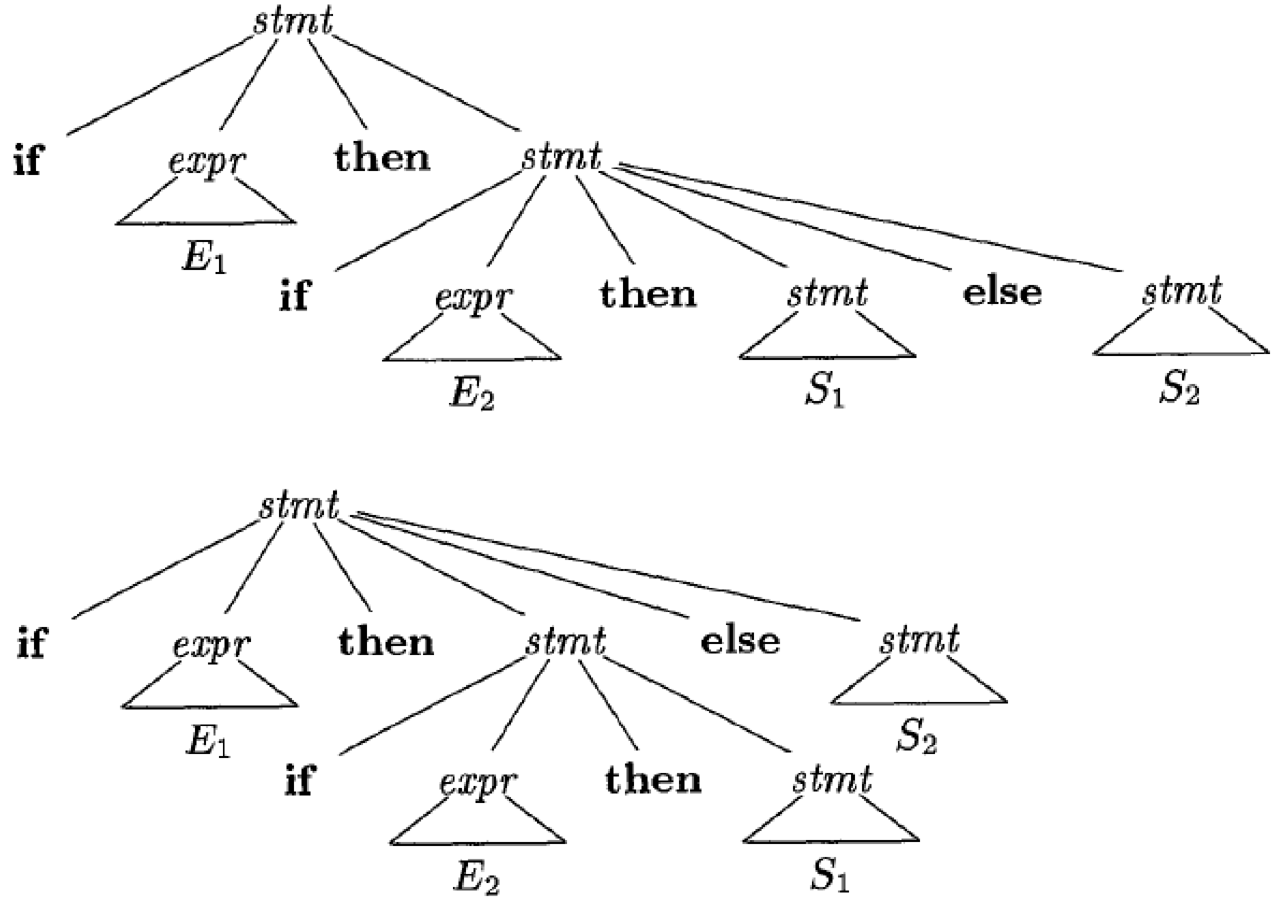
- For this grammar, the string

  if e1 then if e2 then s1 else s2

  has two parse trees

Ambiguity in Programming Lang.

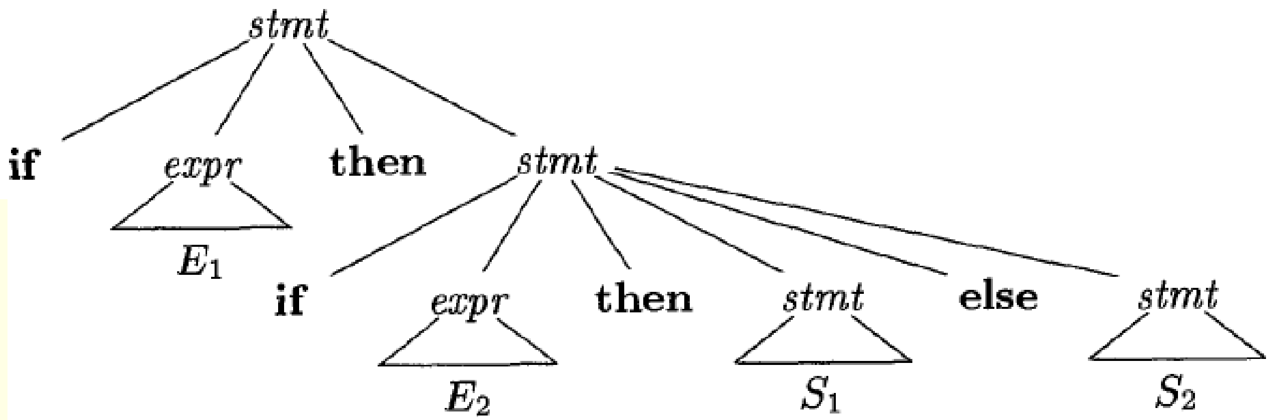
- Parse trees of if e1 then if e2 then s1 else s2



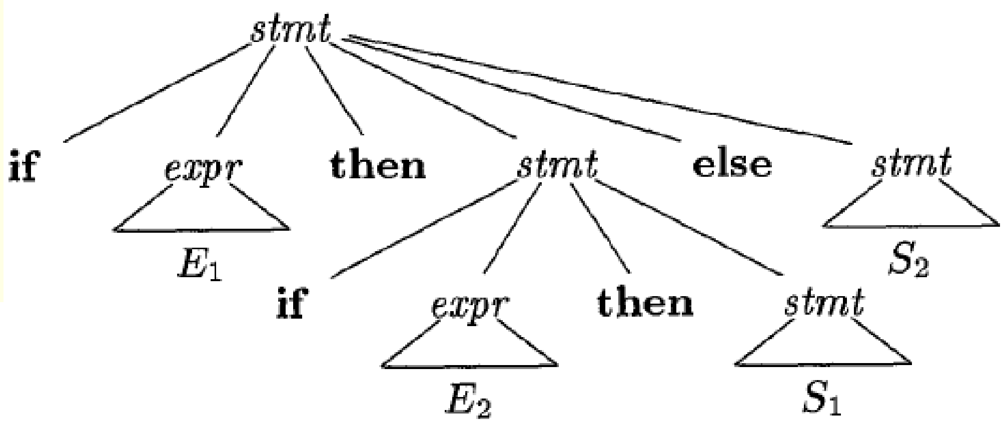
Ambiguity in Programming Lang.

- Parse trees of if e1 then if e2 then s1 else s2

```
if e1
   if e2
      s1
else s2
```




```
if e1
   if e2
      s1
  else s2
```



Resolving dangling else problem

- General rule: match each **else** with the closest previous **unmatched if**. The grammar can be rewritten as

```
stmt → matched-stmt
        | unmatched-stmt
matched-stmt → if expr  matched-stmt
                     else matched-stmt
                   |  others
unmatched-stmt → if expr stmt
                      | if expr matched-stmt
                        else unmatched-stmt
```

Associativity

- If an operand has operator on both the sides, the side on which operator takes this operand is the associativity of that operator
- In a+b+c   b is taken by left +
- +, -, *, / are left associative
- ^, = are right associative
- Grammar to generate strings with right associative operators

  right → letter = right | letter

  letter → a| b |…| z

Precedence

- String a+5*2 has two possible interpretations because of two different parse trees corresponding to

  (a+5)*2 and a+(5*2)
- Precedence determines the correct interpretation.
- Next, an example of how precedence rules are encoded in a grammar

Precedence/Associativity in the Grammar for Arithmetic Expressions

Ambiguous

$E \rightarrow E + E$

$\quad |\ E * E$

$\quad |\ (E)$

$\quad |\ \text{num} \mid \text{id}$

$3 + 2 + 5$

$3 + 2 * 5$

- Unambiguous, with precedence and associativity rules honored

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid \text{num}$

$\quad \mid \text{id}$

Parsing

- Process of determination whether a string can be generated by a grammar
- Parsing falls in two categories:
  - Top-down parsing:

    Construction of the parse tree starts at the root (from the start symbol) and proceeds towards leaves (token or terminals)
  - Bottom-up parsing:

    Construction of the parse tree starts from the leaf nodes (tokens or terminals of the grammar) and proceeds towards root (start symbol)

References

- Compiler Design by Amey Karkare, IIT Kanpur https://karkare.github.io/cs335/
- Compilers: Principles, Techniques, and Tools, Second edition, 2006. by Alfred V. Aho , Monica S. Lam , Ravi Sethi , Jeffrey D. Ullman