UCS1602: COMPILER DESIGN

Introduction to

Syntax analyser

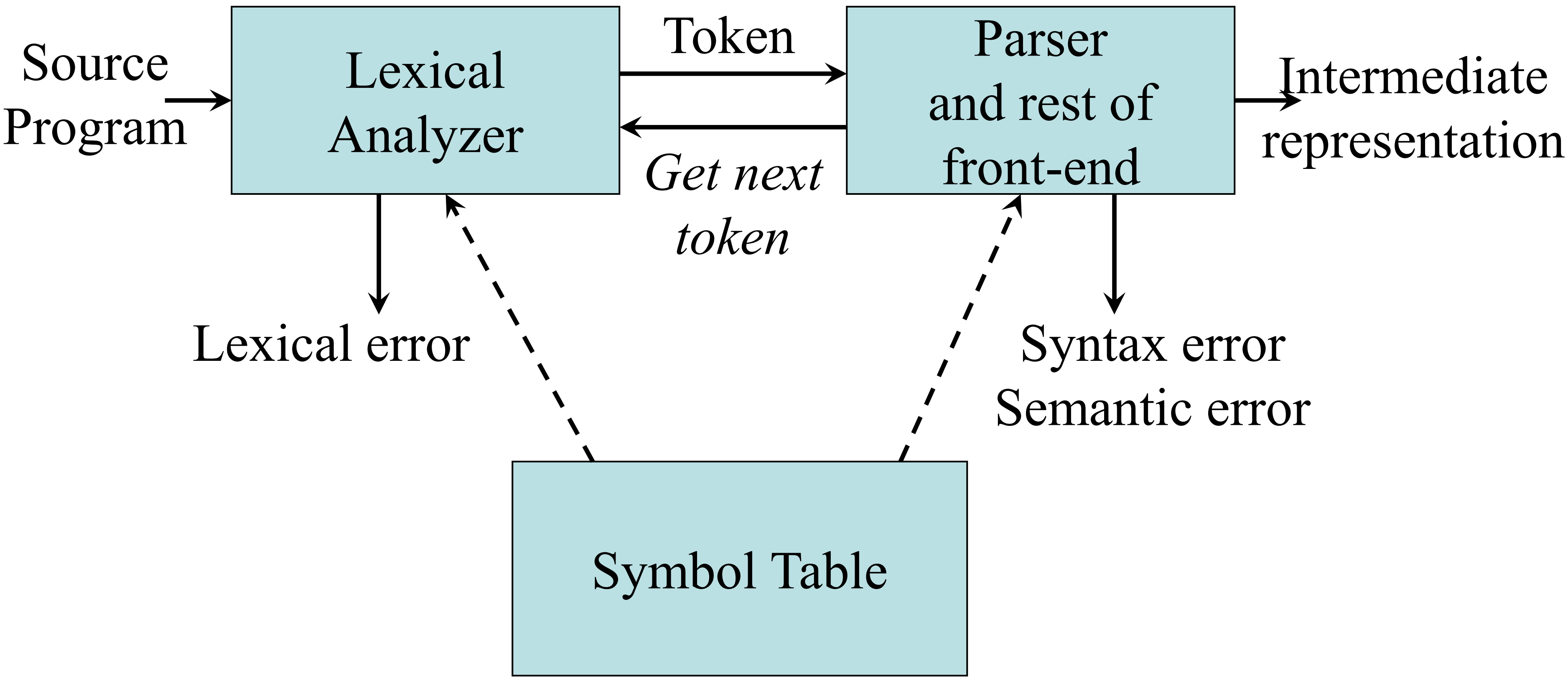
Session Outcomes

- At the end of this session, participants will be able to
  - Understand the concepts of syntax analyzer
  - Understand about the concepts of context free grammar

Outline

- Syntax analyzer
- Types of parsers
- Error recovery
- Context free grammar
- Ambiguous grammar

Role of parser



Syntax Analyzer

- Creates the syntactic structure of the given source program.
- This syntactic structure is mostly a *parse tree*.
- Syntax Analyzer is also known as *parser*.
- The syntax of a programming is described by a *context-free grammar (CFG)*. We will use BNF (Backus-Naur Form) notation in the description of CFGs.

Syntax Analyzer Cont…

- The syntax analyzer (parser) checks whether a given source program satisfies the rules implied by a context-free grammar or not.
  - If it satisfies, the parser creates the parse tree of that program.
  - Otherwise the parser gives the error messages.

- A context-free grammar
  - gives a precise syntactic specification of a programming language.
  - the design of the grammar is an initial phase of the design of a compiler.
  - a grammar can be directly converted into a parser by some tools.

Parsers

1. **Top-Down Parser**
   – the parse tree is created top to bottom, starting from the root.
   – LL parsing
2. **Bottom-Up Parser**
   – the parse is created bottom to top; starting from the leaves
   – LR parsing

- Both top-down and bottom-up parsers scan the input from left to right (one symbol at a time).

Error Handling

- A good compiler should assist in identifying and locating errors.
- Errors can be
  - *Lexical errors*: such as misspelling an identifier, keyword or operator.
  - *Syntax errors*: such as an arithmetic expression with unbalanced parentheses.
  - *Semantic errors*: such as an operator applied to an incompatible operand.
  - *Logical errors*: such as an infinitely recursive call.

Error Recovery Strategies

- *Panic mode*
  - Discard input until a token in a set of designated synchronizing tokens is found
- *Phrase-level recovery*
  - Perform local correction on the input to repair the error
- *Error productions*
  - Augment grammar with productions for erroneous constructs
- *Global correction*
  - Choose a minimal sequence of changes to obtain a global least-cost correction

Context Free Grammar

Grammars

- Inherently recursive structures of a programming language are defined by a context-free grammar.
- Context-free grammar is a 4-tuple
  $G = (N, T, P, S)$ where
  - T is a finite set of tokens (*terminal* symbols)
  - N is a finite set of *nonterminals*
  - P is a finite set of *productions* of the form
    $\alpha \rightarrow \beta$
    where $\alpha \in N$ and $\beta \in (N \cup T)^*$
  - $S \in N$ is a designated *start symbol*

Example

$E \rightarrow E + E$

$E \rightarrow E - E$

$E \rightarrow E * E$

$E \rightarrow E / E$

$E \rightarrow - E$

$E \rightarrow (E)$

$E \rightarrow id$

Notational Conventions Used

- Terminals
  $a,b,c,\ldots \in T$
  specific terminals: **0**, **1**, **id**, **+**
- Nonterminals
  $A,B,C,\ldots \in N$
  specific nonterminals: *expr*, *term*, *stmt*
- Grammar symbols
  $X,Y,Z \in (N \cup T)$
- Strings of terminals
  $u,v,w,x,y,z \in T^*$
- Strings of grammar symbols
  $\alpha,\beta,\gamma \in (N \cup T)^*$

Derivations

$E \Rightarrow E+E$

- 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.

Derivations Cont…

- In general a derivation step is

  $\alpha A\beta \Rightarrow \alpha\gamma\beta$ if there is a production rule $A\rightarrow\gamma$ in our grammar

  where α and β are arbitrary strings of terminal and non-terminal symbols

  $\alpha_1 \Rightarrow \alpha_2 \Rightarrow ... \Rightarrow \alpha_n$ (α_n derives from α_1 or α_1 derives α_n)

$\Rightarrow$: derives in one step

$\overset{*}{\Rightarrow}$: derives in zero or more steps

$\overset{+}{\Rightarrow}$: derives in one or more steps

Derivation Example

- 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**.

Left-Most and Right-Most Derivations

Left-Most Derivation

$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(id+E) \underset{lm}{\Rightarrow} -(id+id)$$

Right-Most Derivation

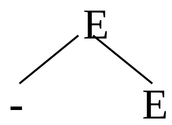
$$E \underset{rm}{\Rightarrow} -E \underset{rm}{\Rightarrow} -(E) \underset{rm}{\Rightarrow} -(E+E) \underset{rm}{\Rightarrow} -(E+id) \underset{rm}{\Rightarrow} -(id+id)$$

Parse Tree

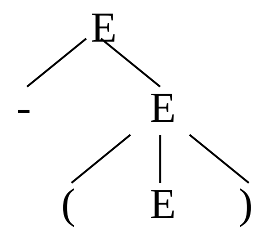
- Inner nodes of a parse tree are non-terminal symbols.
- The leaves of a parse tree are terminal symbols.

- A parse tree can be seen as a graphical representation of a derivation.

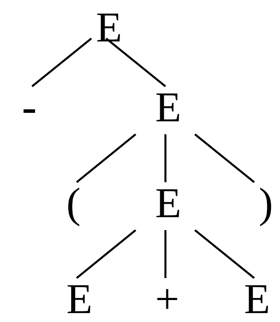
$E \Rightarrow -E$



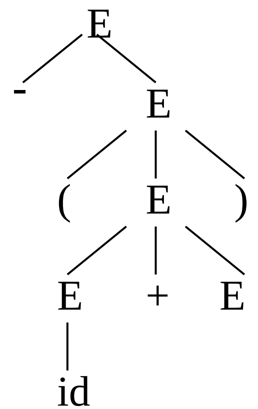
$\Rightarrow -(E)$



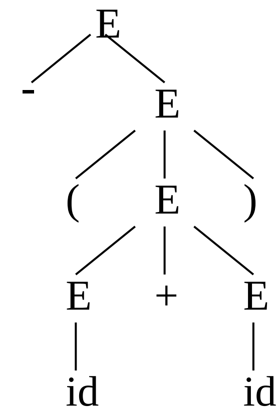
$\Rightarrow -(E+E)$



$\Rightarrow -(id+E)$



$\Rightarrow -(id+id)$



Capabilities of CFG

- Every construct that can be described by a regular expression can also be described by a CFG
- (e.g)

(a|b)*abb $\Rightarrow$

$A_0 \rightarrow aA_0 \mid bA_0 \mid aA_1$

$A_1 \rightarrow bA_2$

$A_2 \rightarrow bA_3$

$A_3 \rightarrow \varepsilon$

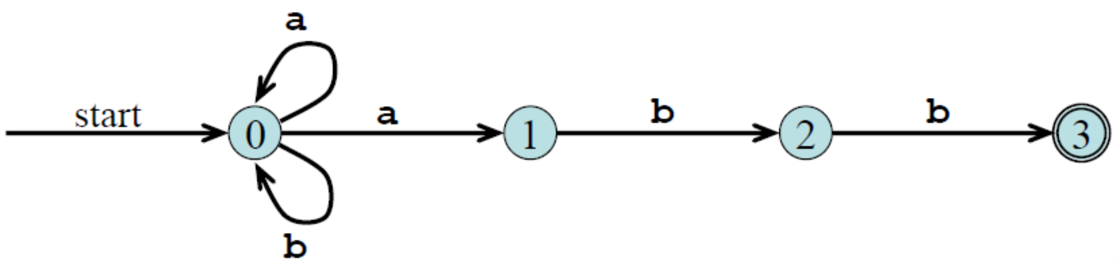
- Check for the string aababb

Algorithm to construct NFA to grammar

For each state i of the NFA, create a non terminal Ai
Begin
    If state I has a transition to state j on symbol a
        Introduce production Ai ->aAj
    If state I goes to state j on input ε
        Introduce production Ai -> Aj
End

If I is an accepting state
    Introduce Ai -> ε
If I is the start state
    Make Ai be the start symbol for the grammar

Example



- For the states 0 to 3 of NFA create NTs A0 to A3
- For A0

  a : A0 -> aA0 , A0 -> aA1

  b : A0 -> bA0
- For A1

  b : A1 -> bA2
- For A2

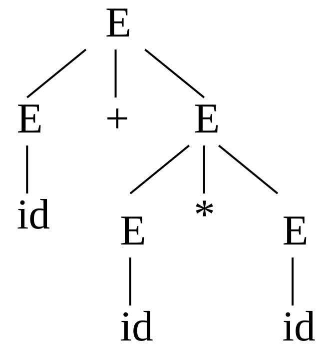
  b : A2 -> bA3
- For A3(accepting state)

  A3 -> ε
- 0 is the start state for NFA, hence A0 is the start state for the grammar

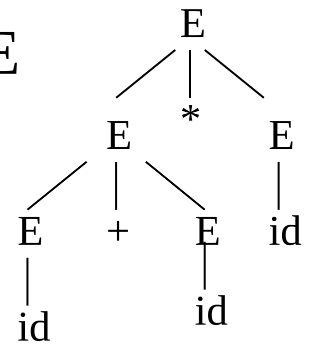
Ambiguity

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

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



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



Ambiguity cont…

- For the most parsers, the grammar must be unambiguous.
- unambiguous grammar

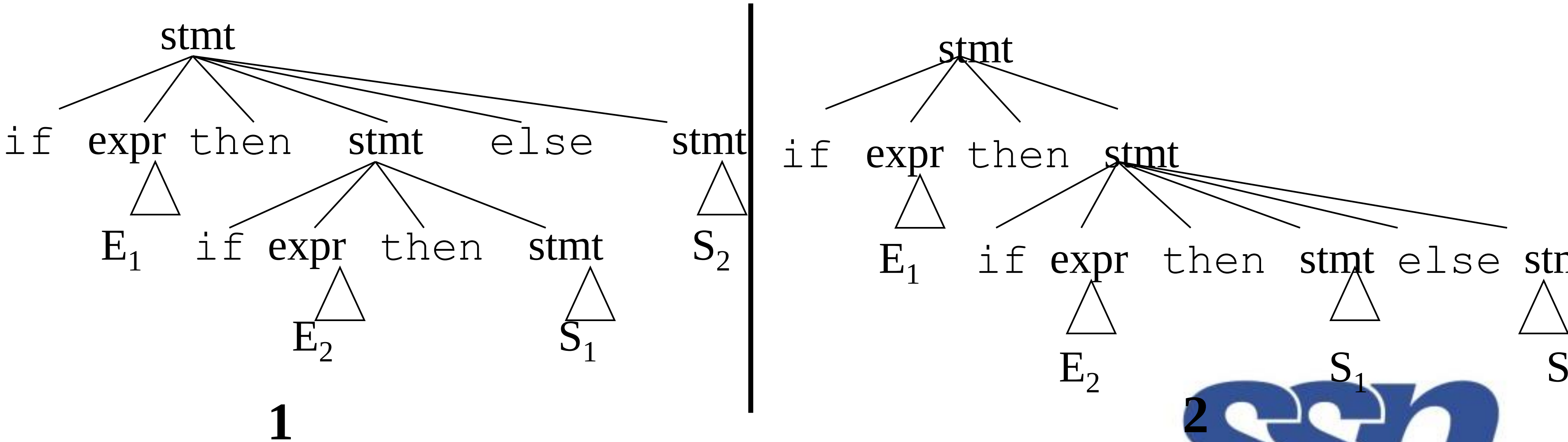
  ➔ unique selection of the parse tree for a sentence

- We should eliminate the ambiguity in the grammar during the design phase of the compiler.
- An unambiguous 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.

Ambiguity cont…

stmt → `if` expr `then` stmt |
`if` expr `then` stmt `else` stmt | otherstmts

`if` E_1 `then` `if` E_2 `then` S_1 `else` S_2



Ambiguity cont…

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

- The unambiguous grammar will be:

stmt → matchedstmt | unmatchedstmt

matchedstmt → `if` expr `then` matchedstmt `else` matchedstmt |otherstmts

unmatchedstmt → `if` expr `then` stmt |
`if` expr `then` matchedstmt `else` unmatchedstmt

Summary

- Role of parser
- Types of parser
- Context free grammar
- Writing a grammar
- Ambiguous grammar

Check your understanding?

- Consider the following CFG:

S → aABe

A → Abc | b

B → d

- Parse the sentence “abbcde” using right-most and left-most derivation
- Draw the parse tree