Syntax Analysis (Parsing)

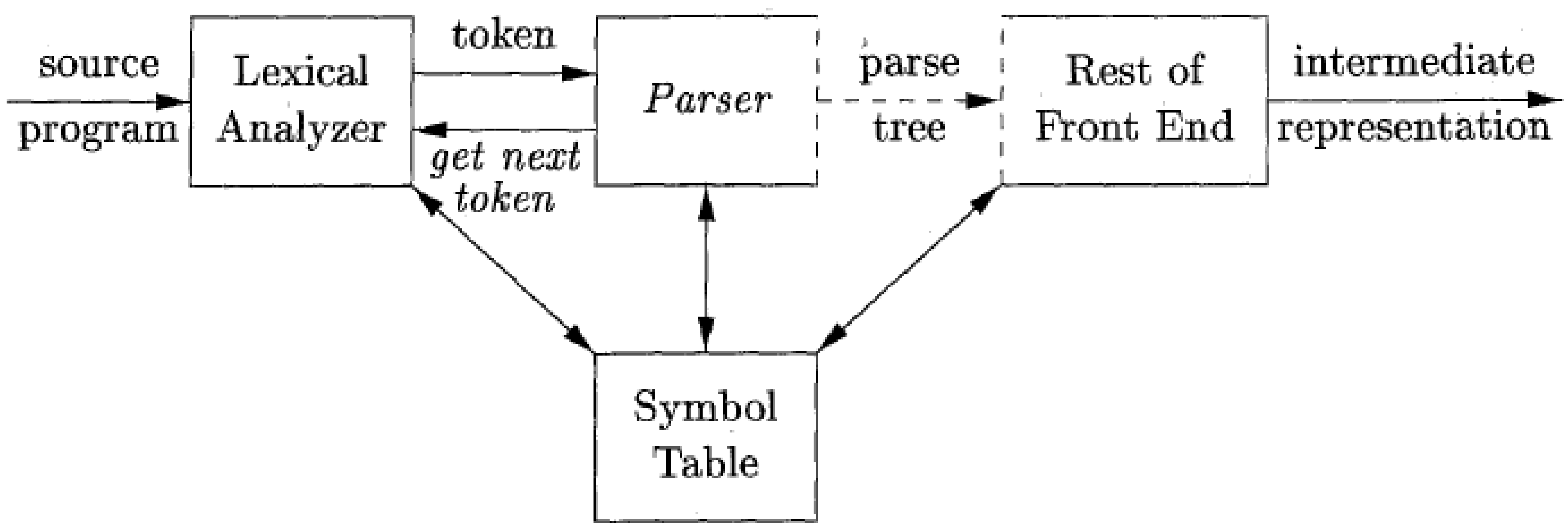
Reference:

Compilers : Principles, Techniques and Tools

Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman

The Role of the Parser

- The parser obtains a string of tokens from the lexical analyser, verifies that the string of token names can be generated by the grammar for the source language.



The Role of the Parser

- Syntax Analyzer creates the **syntactic structure** of the given source program.
- This syntactic structure is mostly a **parse tree**(Syntax tree).
- 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.

The Role of the Parser

- **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.

The Role of the Parser

- 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 reports error messages.

Syntax Error Handling

Common programming errors are:

- **Lexical errors -** misspellings of identifiers, keywords, or operators
  - use of an identifier *elipsesize* instead of *ellipsesize*
  - missing quotes around a string
- **Syntactic errors –** occurs when a stream of tokens cannot be parsed further according to the grammar.
  - misplaced semicolons or extra or missing braces
  - *case* statement without an enclosing *switch*

Syntax Error Handling

- **Semantic errors -** type mismatches between operators and operands
  - a *return* statement in a Java method with result type *void*

- **Logical errors** – an error due to incorrect reasoning on the part of the programmer
  - Use of assignment operator = instead of the comparison operator ==

Syntax Error Handling

The error handler in a parser:

1. should report the presence of errors clearly and accurately
2. should recover from error quickly to detect subsequent errors
3. should not significantly slow down the processing of correct programs

Error Recovering Strategies

1. **Panic Mode:**
   - the parser discards input symbols one at a time until one of a designated set of **synchronizing tokens** is found.
   - synchronizing tokens are usually delimiters, such as semicolon or } or comma
   - advantage of panic mode is its simplicity
   - Ex:
     - `int a, 5abcd, sum, $2;`
     - parser discards input symbol one at a time for `5abcd` and `$2` .
     - `int a,` ~~5abcd~~`, sum,` ~~$2~~`;`

2. Phrase Level

- On discovering an error, a parser may perform correction on the remaining input;
- Corrections are
  - replace a prefix of the remaining input with some string
  - delete an extraneous semicolon
  - insert a missing semicolon
- ex:

```
int a,b
```

  - // After recovery:

```
int a,b;
```

3. Error Productions

- If a user has knowledge of common errors that can be encountered then, these errors can be incorporated by **augmenting the grammar with error productions** that generate erroneous constructs.

- If this is used then, during parsing **appropriate error messages can be generated and parsing can be continued**.

4. Global Corrections

- For a incorrect input string x the parser tries to find out the closest match y (which is error-free) such that
- The closest match string y has less number of insertions, deletions, and changes of tokens required to transform x into y.

- Due to high time and space complexity, this method is not implemented practically.

Types of Parsers

- Commonly used methods for parsers

1. Top-down parsers

   build parse tree from top (root) to bottom(leaves)

2. Bottom-up Parsers

   build parse tree from bottom to top

- In both cases, input to the parser is scanned from left to right

4.2 Context-Free Grammars

- Inherently recursive structures of a programming language are defined by a context-free grammar.

Example : Conditional **if**

$$\texttt{if } \mathrm{E} \texttt{ then } \mathrm{S}_1 \texttt{ else } \mathrm{S}_2$$

S1 & S2 are statements and E is an expression.

Context-Free Grammars

- Using a syntactic variable **stmt** to denote statements and variable **expr** to denote expressions, the following grammar specifies the structure of conditional statement

$$stmt \rightarrow \text{if } (\; expr \;) \; stmt \text{ else } stmt$$

Note : this form of conditional statement **cannot** be expressed by regular expressions

Context-Free Grammars

- In a context-free grammar, we have:
  1) A finite set of **terminals**
     - "token name" is a synonym for '"terminal"
     - Alternatively, we will use the word "token" for terminal
     - In the previous grammar, the terminals are the keywords **if** and **else** and the symbols "**(**" and "**)** ."

2) A finite set of **non-terminals**

- A non-terminal is a syntactic-variable that denotes set of strings
- These strings help define the language generated by the grammar
- **stmt** and **expr** are non-terminals.

3) A **start symbol – is** one of the non-terminal symbol

- the set of strings denoted by a start symbol is the language generated by the grammar

4) A finite set of **productions** rules in the following form

$$A \rightarrow \alpha$$

- where A is a non-terminal α is a string of terminals and non-terminals (including the empty string)
- Production specifies the manner in which the terminals and nonterminals can be combined to form strings

Example: Grammar for simple arithmetic expressions

$$
\begin{aligned}
expression &\rightarrow expression + term \\
expression &\rightarrow expression - term \\
expression &\rightarrow term \\
term &\rightarrow term * factor \\
term &\rightarrow term \ / \ factor \\
term &\rightarrow factor \\
factor &\rightarrow (\ expression\ ) \\
factor &\rightarrow \mathbf{id}
\end{aligned}
$$

- Terminals: id + - * / ( )
- Nonterminals: *expression*, *term* and *factor*
- *Start symbol:* *expression*

Notational Conventions

1. These symbols are terminals:

   (a) Lowercase letters early in the alphabet, such as a, b, c.

   (b) Operator symbols such as $+$, $*$, and so on.

   (c) Punctuation symbols such as parentheses, comma, and so on.

   (d) The digits $0, 1, \ldots, 9$.

   (e) Boldface strings such as **id** or **if**, each of which represents a single terminal symbol.

Notational Conventions

2. These symbols are nonterminals:
   (a) Uppercase letters early in the alphabet, such as A, B, C.
   (b) The letter S, which, when it appears, is usually the start symbol.
   (c) Lowercase, italic names such as *expr* or *stmt*.

3. Uppercase letters late in the alphabet, such as X, Y, Z, represent *grammar symbols*; that is, either nonterminals or terminals.

4. Lowercase letters late in the alphabet, chiefly $u, v, \ldots, z$, represent (possibly empty) strings of terminals.

Notational Conventions

5. Lowercase Greek letters, α, β, γ for example, represent (possibly empty) strings of grammar symbols. Thus, a generic production can be written as $A \to \alpha$, where A is the head and α the body.

6. A set of productions $A \to \alpha_1$, $A \to \alpha_2, \ldots$, $A \to \alpha_k$ with a common head A (call them A-*productions*), may be written $A \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k$. Call $\alpha_1, \alpha_2, \ldots, \alpha_k$ the *alternatives* for A.

7. Unless stated otherwise, the head of the first production is the start symbol.

Notational Conventions

For Example

$$
\begin{aligned}
E &\rightarrow E + T \mid E - T \mid T \\
T &\rightarrow T * F \mid T \,/\, F \mid F \\
F &\rightarrow (\, E \,) \mid \mathbf{id}
\end{aligned}
$$

E, T, and F are non terminals, with E the start symbol.

The remaining symbols are terminals.

Derivations

Consider,

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

We can take a single E and repeatedly apply productions in any order to get a sequence of replacements. Ex:

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathbf{id})$$

A sequence of replacements of non-terminal symbols is called a **derivation** of –(id) from E.

Derivations

- **General Definition**

We write $\alpha A\beta \Rightarrow \alpha\gamma\beta$

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

When a sequence of derivation steps $\alpha_1 \Rightarrow \alpha_2 \Rightarrow ... \Rightarrow \alpha_n$ rewrites α_1 to α_n , we say, α_n derives from α_1 or α_1 **derives** α_n

Derivations

Notations:

$\Rightarrow$: derives in one step

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

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

Example:

1. $\alpha \stackrel{*}{\Rightarrow} \alpha$, for any string α, and

2. If $\alpha \stackrel{*}{\Rightarrow} \beta$ and $\beta \Rightarrow \gamma$, then $\alpha \stackrel{*}{\Rightarrow} \gamma$.

CFG - Terminology

- L(G) is the language of 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) iff $S \overset{+}{\Rightarrow} \omega$ where ω is a string of terminals of G.

CFG - Terminology

- If G is a context-free grammar, $L(G)$ is a *context-free language*.
- Two grammars are *equivalent* if they produce the same language.

- $S \stackrel{*}{\Rightarrow} \alpha$
  - If α contains non-terminals, it is called as a *sentential* form of G.
    - may contain both terminals and nonterminals, and may be empty.
  - If α does not contain non-terminals, it is called as a *sentence* of G.

Left-Most and Right-Most Derivations

- If we always choose the left-most non-terminal in each derivation step, this derivation is called as **left-most derivation**.

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

- If we always choose the right-most non-terminal in each derivation step, this derivation is called as **right-most derivation**.

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

Left-Most and Right-Most Derivations

- In *leftmost* derivations if $\alpha \Rightarrow \beta$ is a step, we write $\alpha \underset{lm}{\Rightarrow} \beta$

- In *rightmost* derivations if $\alpha \Rightarrow \beta$ is a step, we write $\alpha \underset{rm}{\Rightarrow} \beta$

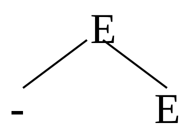
- If $S \overset{*}{\underset{lm}{\Rightarrow}} \alpha$, then we say that α is a left-sentential form of the grammar at hand.

Parse Tree

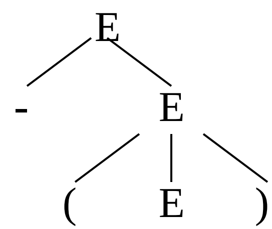
- A parse tree can be seen as a graphical representation of a derivation.

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

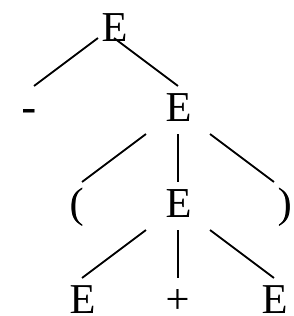
$E \Rightarrow -E$



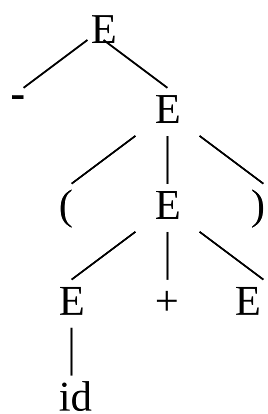
$\Rightarrow -(E)$



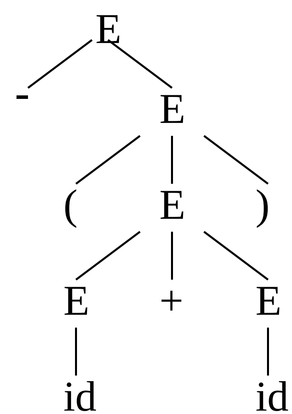
$\Rightarrow -(E+E)$



$\Rightarrow -(id+E)$



$\Rightarrow -(id+id)$



Parse Tree

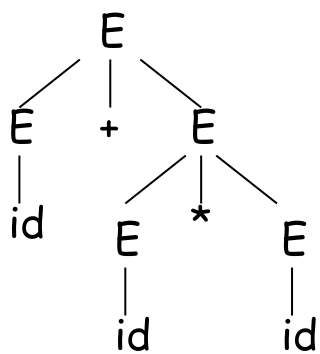
- Each interior node is labeled with nonterminal
- Leaves of a parse tree are labeled by nonterminals or terminals
    - constitute a sentential form,
    - called the **yield** or **frontier** of the tree
- There is a one-to-one relationship between parse trees and either leftmost or rightmost derivations

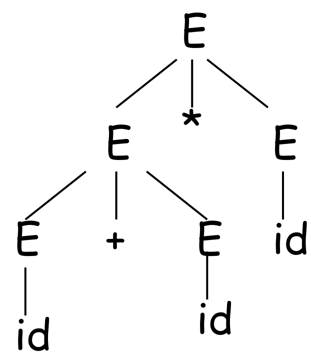
Parse Tree

- Get a parse tree for the derivations:

1. $E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+E*E \Rightarrow id+id*E \Rightarrow id+id*id$

2. $E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow id+E*E \Rightarrow id+id*E \Rightarrow id+id*id$



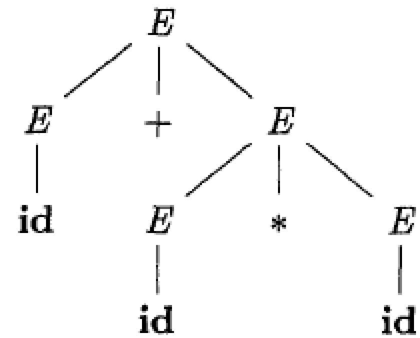


Ambiguity

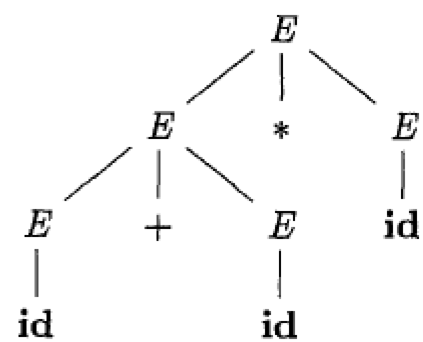
- A grammar that produces more than one parse tree for a sentence is alled as an ***ambiguous*** grammar.

The grammar $E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \mathbf{id}$ permits two distinct leftmost derivations for the sentence **id + id * id**

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



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



For the most parsers, the grammar must be unambiguous.

Ambiguity (cont.)

Note :

- 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.

Left-Most and Right-Most Derivations

Exercise-4.2.1:

Consider the context-free grammar:

$$S \rightarrow S\,S\,+ \mid S\,S\,* \mid a$$

and the string $aa + a*$.

a) Give a leftmost derivation for the string.

b) Give a rightmost derivation for the string.

c) Give a parse tree for the string.

d) Is the grammar ambiguous or unambiguous? Justify your answer.

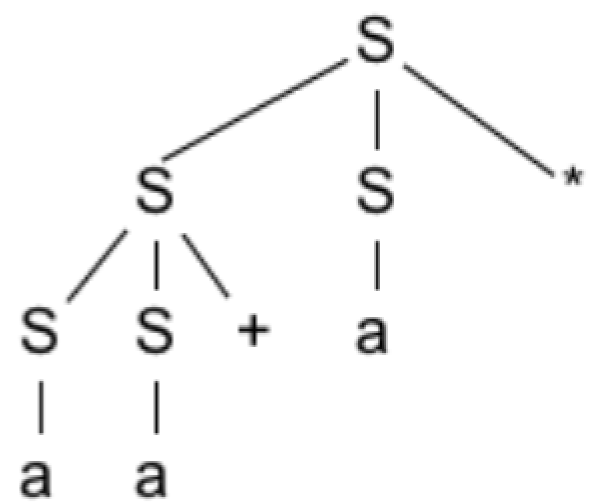
e) Describe the language generated by this grammar.

Ans

1. S =lm=> SS* => SS+S* => aS+S* => aa+S* => aa+a*

2. S =rm=> SS* => Sa* => SS+a* => Sa+a* => aa+a*

3.



4. Unambiguous

L = {Postfix expression consisting of digits, plus and multiply signs}

Exercise-4.2.2: Repeat Exercise 4.2.1 for each of the following grammars and strings:

a) $S \rightarrow 0\ S\ 1 \mid 0\ 1$ with string 000111.

b) $S \rightarrow +\ S\ S \mid *\ S\ S \mid a$ with string $+ * aaa$.

c) $S \rightarrow S\ (\ S\ )\ S \mid \epsilon$ with string (()()).

d) $S \rightarrow S\ +\ S \mid S\ S \mid (\ S\ ) \mid S\ * \mid a$ with string $(a+a)*a$.

e) $S \rightarrow (\ L\ ) \mid a$ and $L \rightarrow L\ ,\ S \mid S$ with string $((a, a), a, (a))$.

f) $S \rightarrow a\ S\ b\ S \mid b\ S\ a\ S \mid \epsilon$ with string $aabbab$.

g) The following grammar for boolean expressions:

$$
\begin{array}{lcl}
bexpr & \rightarrow & bexpr \ \mathbf{or} \ bterm \mid bterm \\
bterm & \rightarrow & bterm \ \mathbf{and} \ bfactor \mid bfactor \\
bfactor & \rightarrow & \mathbf{not} \ bfactor \mid (\ bexpr\ ) \mid \mathbf{true} \mid \mathbf{false}
\end{array}
$$

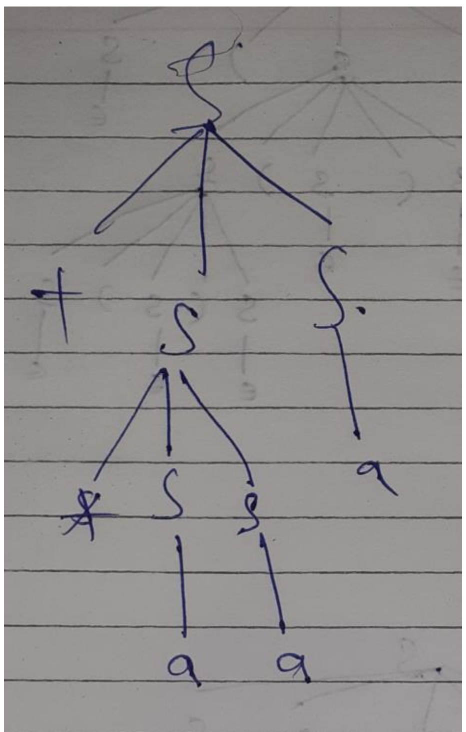
Ans-1

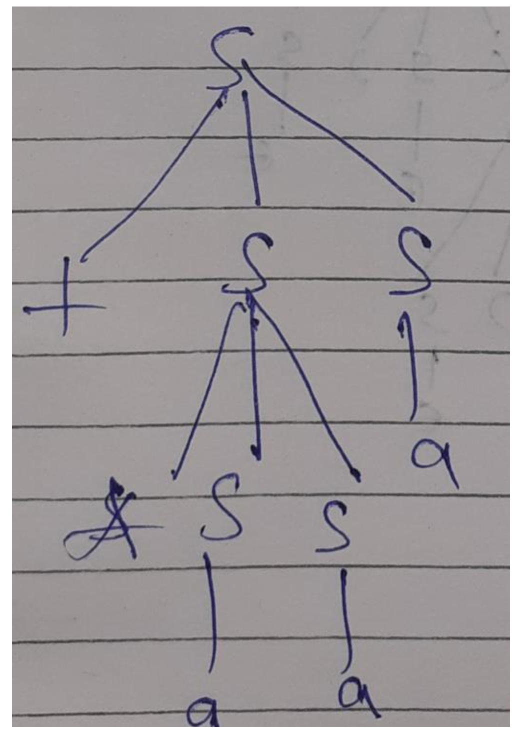
S -> 0 S 1 | 0 1 with string 00011l.

1. S =lm=> 0S1 => 00S11 => 000111
2. S =rm=> 0S1 => 00S11 => 000111
3.
4. Unambiguous
5. The set of all strings of 0s and followed by an equal number of 1s

2. S -> + S S | * S S | a with string + * aaa.

1. S =lm=> +SS => +*SSS => +*aSS => +*aaS => +*aaa
2. S =rm=> +SS => +Sa => +*SSa => +*Saa => +*aaa
3.
4. Unambiguous
5. The set of all prefix expressions consist of addition and multiplication.

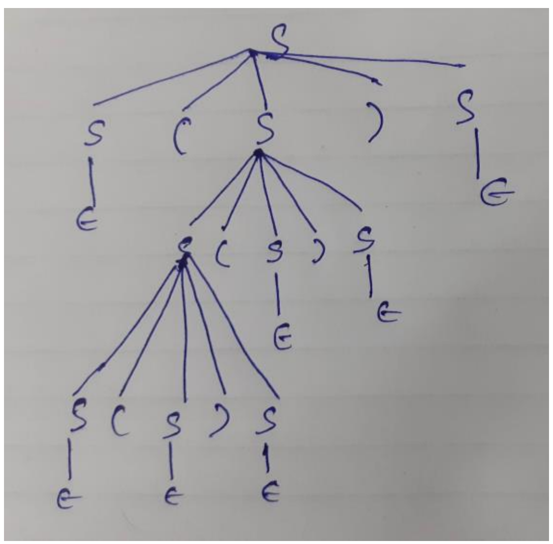


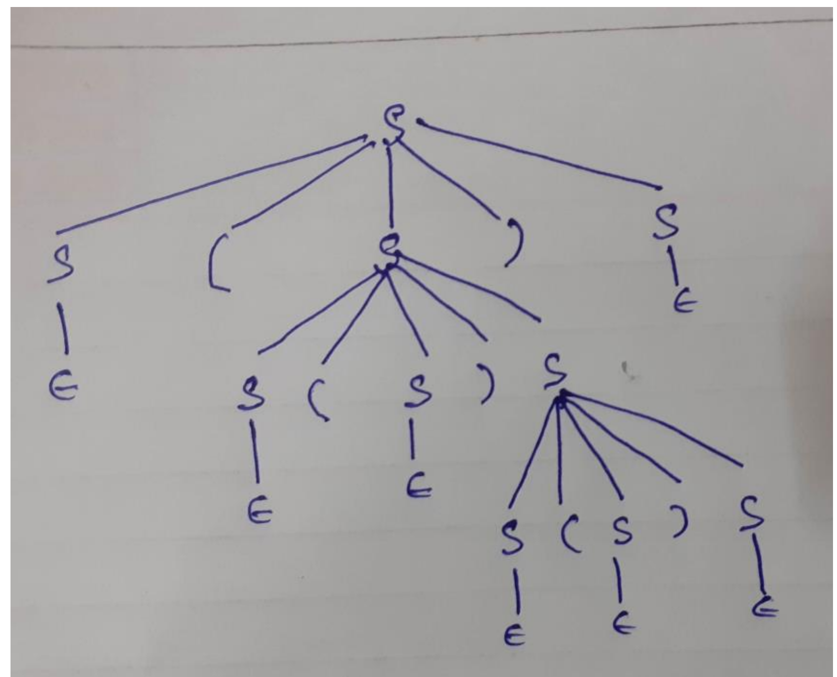


3

S -> S (S) S | ε with string (()())

1. S =lm=> S(S)S => (S)S => (S(S)S)S => ((S)S)S => (()S)S => (()S(S)S)S => (()(S)S)S => (()()S)S => (()())S => (()())
2. S =rm=> S(S)S => S(S) => S(S(S)S) => S(S(S)) => S(S()) => S(S(S)S()) => S(S(S)()) => S(S()()) => S(()()) => (()())
3.
4. Ambiguous
5. The set of all strings of symmetrical parentheses





4

S -> S + S | S S | (S) | S * | a with string (a+a)*a

1. S =lm=> SS => S*S => (S)*S => (S+S)*S => (a+S)*S => (a+a)*S => (a+a)*a
2. S =rm=> SS => Sa => S*a => (S)*a => (S+S)*a => (S+a)*a => (a+a)*a
3.
4. Ambiguous

5. The set of all strings of multiplication, ‘a’, and symmetrical parenthesis; and plus is not in the beginning and end; multiplication is not in the beginning of the string

5

S -> (L) | a
L -> L, S | S
with string ((a,a),a,(a))

1. S =lm=> (L) => (L, S) => (L, S, S) => ((S), S, S) => ((L), S, S)

=> ((L, S), S, S) => ((S, S), S, S) => ((a, S), S, S) => ((a, a), S, S) => ((a, a), a, S)

=> ((a, a), a, (L)) => ((a, a), a, (S)) => ((a, a), a, (a))

2. S =rm=> (L) => (L, S) => (L, (L)) => (L, (a)) => (L, S, (a)) => (L, a, (a)) => (S, a, (a))

=> ((L), a, (a)) => ((L, S), a, (a)) => ((S, S), a, (a))

=> ((S, a), a, (a)) => ((a, a), a, (a))

4. Unambiguous

5. Something like tuple in Python

6

S -> a S b S | b S a S | ε   with string aabbab

1. S =lm=> aSbS => aaSbSbS => aabSbS => aabbS => aabbaSbS => aabbabS => aabbab
2. S =rm=> aSbS => aSbaSbS => aSbaSb => aSbab => aaSbSbab => aaSbbab => aabbab
3.
4. Ambiguous
5. The set of all strings of 'a's and 'b's of the equal number of 'a's and 'b's

7

bexpr -> *bexpr* **or** *bterm* | *bterm*

bterm -> *bterm* **and** *bfactor* | *bfactor*

bfactor -> **not** *bfactor* | (*bexpr*) | **true** | **false**

- Grammar is Unambiguous
- Language generated by this grammar is **booean expression**

Lexical Versus Syntactic Analysis

- Everything that can be described by a regular expression can also be described by a grammar.
- Then, why use regular expressions to define the lexical syntax of a language?
- Reasons are:
  - modularizing into lexical and non-lexical parts
  - lexical rules of a language are quite simple, and to describe them we do not need a notation as powerful as grammars.
  - Regular expressions provide a more concise and easier-to-understand notation for tokens than grammars.
  - lexical analyzers can be constructed automatically from regular expressions than from arbitrary grammars.

- Regular expressions are most useful for describing the structure of identifiers, constants, keywords, and white space

- Grammars are useful for describing nested structures such as balanced parentheses, matching begin-end's, corresponding if-then-else's, and so on

Eliminating Ambiguity

Example1:

Ambiguous grammar

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

unambiguous grammar for the above

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

Because the only derivation for the string id+id*id is

$E \Rightarrow E + T \Rightarrow T + T$
$\Rightarrow F + T \Rightarrow id + T$
$\Rightarrow id + T * F \Rightarrow id + F * F$
$\Rightarrow id + id * F$
$\Rightarrow id + id * id$

So you have only one parse tree for id+id*id

- $E \rightarrow T * E | T$
- $T \rightarrow F + T | F$
- $F \rightarrow id | (E)$

- $E \rightarrow E + T | F$
- $T \rightarrow T * F | F$
- $F \rightarrow G \wedge F | G$
- $G \rightarrow id$

- Give an unambiguous grammar for the following grammar such that MINUS has higher priority, DIVIDE has less priority and both are right associative.

  $E \rightarrow E - E \mid E / E \mid id.$

Ans:

$E \rightarrow T / E \mid T$

$T \rightarrow F – T \mid F$

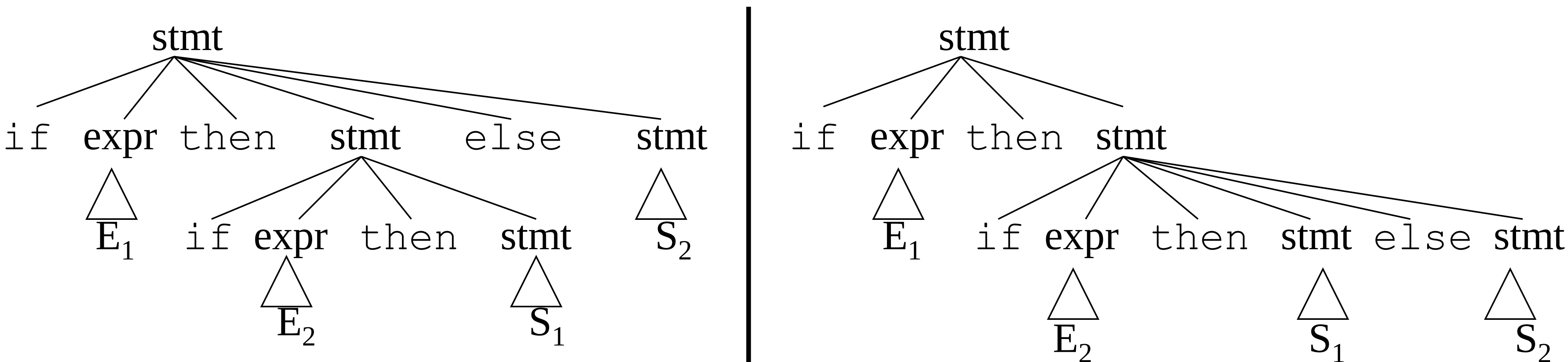
$F \rightarrow id$

Eliminating Ambiguity

$$
\begin{array}{lll}
stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\
 & | & \textbf{other}
\end{array}
$$

⟵ "dangling else" grammar

$\textbf{if } E_1 \textbf{ then if } E_2 \textbf{ then } S_1 \textbf{ else } S_2$ has two parsen trees



1 **2**

Eliminating Ambiguity

- We prefer the second parse tree (else matches with closest if).
- We can rewrite the dangling-else grammar as the following unambiguous grammar

$$
\begin{array}{rcl}
stmt & \rightarrow & matched\_stmt \\
 & | & open\_stmt \\
matched\_stmt & \rightarrow & \textbf{if } expr \textbf{ then } matched\_stmt \textbf{ else } matched\_stmt \\
 & | & \textbf{other} \\
open\_stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } matched\_stmt \textbf{ else } open\_stmt
\end{array}
$$

The idea is that a statement appearing between a then and an else must be "matched" ;

Left Recursion

A grammar is left recursive if it has a non-terminal A such that there is a derivation.

$A \Rightarrow A\alpha$ for some string α

- The left-recursion may appear in a single step of the derivation (immediate left-recursion), or may appear in more than one step of the derivation.

Left Recursion

Note

- Top-down parsing techniques **cannot** handle left-recursive grammars.
- So, we have to convert our left-recursive grammar into an equivalent grammar which is not left-recursive.

Elimination of Immediate Left-Recursion

$A \rightarrow A\ \alpha\ |\ \beta$ where β does not start with A

$\Downarrow$

$A \rightarrow \beta\ A'$

$A' \rightarrow \alpha\ A'\ |\ \varepsilon$ an equivalent grammar

Immediate Left-Recursion

In general,

$A \rightarrow A\ \alpha_1 \mid \ldots \mid A\ \alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$

where $\beta_1 \ldots \beta_n$ do not start with A

$\Downarrow$

$A \rightarrow \beta_1\ A' \mid \ldots \mid \beta_n\ A'$

$A' \rightarrow \alpha_1\ A' \mid \ldots \mid \alpha_m\ A' \mid \varepsilon$ an equivalent grammar

Elimination of Immediate Left-Recursion

Exercise

Eliminate immediate left recursion from the expression grammar

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
\longrightarrow
\begin{array}{lcl}
E & \rightarrow & T\, E' \\
E' & \rightarrow & +\, T\, E' \mid \epsilon \\
T & \rightarrow & F\, T' \\
T' & \rightarrow & *\, F\, T' \mid \epsilon \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$

Left-Recursion -- Problem

- A grammar need not be immediately left-recursive, but it still can be left-recursive. By just eliminating the immediate left-recursion, we may not get a grammar which is not left-recursive.

$S \rightarrow Aa \mid b$
$A \rightarrow Sc \mid d$

not immediately left-recursive, but still left-recursive

Since ,

$\underline{S} \Rightarrow Aa \Rightarrow \underline{S}ca$ or
$\underline{A} \Rightarrow Sc \Rightarrow \underline{A}ac$ causes to a left-recursion

- So, we have to eliminate all left-recursions from our grammar

Eliminate Left-Recursion -- Algorithm

- Arrange non-terminals in some order: $A_1 \ldots A_n$
- **for** i **from** 1 **to** n **do** {
    - **for** j **from** 1 **to** i-1 **do** {

        replace each production

        $A_i \rightarrow A_j \gamma$

        by

        $A_i \rightarrow \alpha_1 \gamma \mid \ldots \mid \alpha_k \gamma$

        where $A_j \rightarrow \alpha_1 \mid \ldots \mid \alpha_k$

        }
    - eliminate immediate left-recursions among A_i productions

}

Eliminate Left-Recursion -- Example

Exercise

1. Eliminate Left-Recursion from the grammar

$$S \to Aa \mid b$$
$$A \to Ac \mid Sd \mid f$$

Eliminate Left-Recursion – soln1

$S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid f$

- Order of non-terminals: S, A

for S:

- we do not enter the inner loop.
- there is no immediate left recursion in S.

for A:

- Replace $A \rightarrow Sd$ with $A \rightarrow Aad \mid bd$
  So, we will have $A \rightarrow Ac \mid Aad \mid bd \mid f$
- Eliminate the immediate left-recursion in A

$$A \rightarrow bdA' \mid fA'$$
$$A' \rightarrow cA' \mid adA' \mid \varepsilon$$

So, the resulting equivalent grammar which is not left-recursive is:

$$S \rightarrow Aa \mid b$$
$$A \rightarrow bdA' \mid fA'$$
$$A' \rightarrow cA' \mid adA' \mid \varepsilon$$

Eliminate Left-Recursion – soln2

$S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid f$

- Order of non-terminals: A, S

for A:

- we do not enter the inner loop.
- Eliminate the immediate left-recursion in A

$$A \rightarrow SdA' \mid fA'$$
$$A' \rightarrow cA' \mid \varepsilon$$

for S:

- Replace $S \rightarrow Aa$ with $S \rightarrow SdA'a \mid fA'a$
  So, we will have $S \rightarrow SdA'a \mid fA'a \mid b$
- Eliminate the immediate left-recursion in S

$$S \rightarrow fA'aS' \mid bS'$$
$$S' \rightarrow dA'aS' \mid \varepsilon$$

So, the resulting equivalent grammar which is not left-recursive is:

$S \rightarrow fA'aS' \mid bS'$
$S' \rightarrow dA'aS' \mid \varepsilon$
$A \rightarrow SdA' \mid fA'$
$A' \rightarrow cA' \mid \varepsilon$

Left-Factoring

- A predictive parser (a top-down parser without backtracking) insists that the grammar must be *left-factored*.

consider

$$\begin{array}{lll} stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\ & | & \textbf{if } expr \textbf{ then } stmt \end{array}$$

- when we see if, we cannot tell immediately which production rule to choose to re-write *stmt* in the derivation.

Left-Factoring (cont.)

- In general,

  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ where α is non-empty and the first symbols of β_1 and β_2 (if they have one)are different.

- when processing α we cannot know whether expand

  A to $\alpha\beta_1$ or

  A to $\alpha\beta_2$

- But, if we re-write the grammar as follows

  $A \rightarrow \alpha A'$

  $A' \rightarrow \beta_1 \mid \beta_2$ so, we can immediately expand A to $\alpha A'$

Left-Factoring -- Algorithm

For each non-terminal A with two or more alternatives (production rules) with a common non-empty prefix, let say

$$A \rightarrow \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \ldots \mid \gamma_m$$

convert it into

$$A \rightarrow \alpha A' \mid \gamma_1 \mid \ldots \mid \gamma_m$$

$$A' \rightarrow \beta_1 \mid \ldots \mid \beta_n$$

Left-Factoring

Exercise:

Eliminate Left recursion from the productions

1. $A \rightarrow abB \mid aB \mid cdg \mid cdeB \mid cdfB$
2. $A \rightarrow ad \mid a \mid ab \mid abc \mid b$

Left-Factoring – soln1

$$A \rightarrow \underline{a}bB \mid \underline{a}B \mid cdg \mid cdeB \mid cdfB$$

$$\Downarrow$$

$$A \rightarrow aA' \mid \underline{cd}g \mid \underline{cd}eB \mid \underline{cd}fB$$

$$A' \rightarrow bB \mid B$$

$$\Downarrow$$

$$A \rightarrow aA' \mid cdA''$$

$$A' \rightarrow bB \mid B$$

$$A'' \rightarrow g \mid eB \mid fB$$

Left-Factoring – soln2

$$A \to ad \mid a \mid ab \mid abc \mid b$$

$$\Downarrow$$

$$A \to aA' \mid b$$
$$A' \to d \mid \varepsilon \mid b \mid bc$$

$$\Downarrow$$

$$A \to aA' \mid b$$
$$A' \to d \mid \varepsilon \mid bA''$$
$$A'' \to \varepsilon \mid c$$

THANK YOU

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 = { $\omega c \omega$ | ω 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 = {$a^n b^m c^n d^m$ | $n \geq 1$ 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.

Additional

- A formal grammar is "context free" if its production rules can be applied regardless of the context of a nonterminal. No matter which symbols surround it, the single nonterminal on the left hand side can always be replaced by the right hand side. This is what distinguishes it from a context-sensitive grammar.

- A context-sensitive grammar (CSG) is a formal grammar in which the left-hand sides and right-hand sides of any production rules may be surrounded by a context of terminal and nonterminal symbols.

Ex:

- aB→ab
- bB→bb