Sub: Compiler Construction
Syntax Analysis PART 1

Compiled for: 7th Sem, CE, DDU

Compiled by: Niyati J. Buch

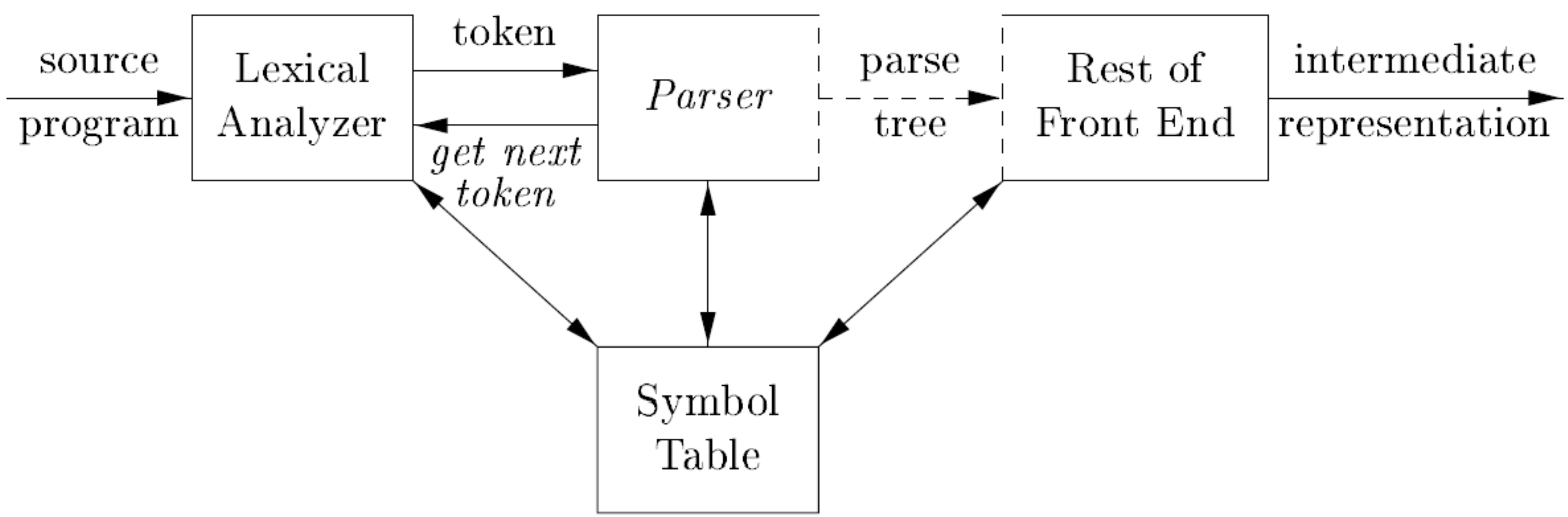
Ref. : Compilers: Principles, Techniques, and Tools, 2nd Edition
Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman

Topics Covered

Introduction to Syntax Analysis

Role of a parser

- A parser uses a grammar to check structure of tokens.
- It produces a parse tree.
- It checks for syntactic errors and recovery.
- It recognize correct syntax.
- It report errors.



3 Types of Parsers

1. **Universal parsing methods**
   – Such as Cocke-Younger-Kasami algorithm and Earley's algorithm can parse any grammar.
   – But they are too inefficient to use in production compilers.
2. **Top down methods**
   – Build the parse tree from top(root) to the bottom(leaves)
3. **Bottom up methods**
   – Builds the parse tree from bottom(leaves) to the top(root)

Types of parser

- Top-down: **LL** → scan **L**eft to right and consider **L**eft most derivative
- Bottom-up: **LR** → scan **L**eft to right and consider **R**ight most derivative in reverse
- In both top-down and bottom up, the input to the parser is scanned from **left to right** , one symbol at a time.
- Parsers implemented by hand often use LL grammar(ex. predictive parsing)
- Parsers for the larger class of LR grammars are usually constructed using automated tools.

Representative Grammars

- Associativity and precedence are captured in the following grammar, for describing expressions, terms, and factors.
- E represents expressions consisting of terms separated by + signs
- T represents terms consisting of factors separated by * signs
- F represents factors that can be either parenthesized expressions or identifiers:

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid id$$

Can you observe any recursion??

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

Can you observe any recursion??

E → E + T | T

T → T * F | F

F → (E) | id

- Recursion on **left side** is observed.
- **LR grammar** is suitable for **bottom up parsing**.
- It cannot be used for top-down parsing because it is left recursive.

Non-left-recursive variant

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

$E \rightarrow T E'$

$E' \rightarrow + T E' \mid \varepsilon$

$T \rightarrow F T'$

$T' \rightarrow * F T' \mid \varepsilon$

$F \rightarrow (E) \mid id$

Common Programming Errors

- **Lexical Errors**
  - Misspellings of identifiers, keywords, or operators
- **Syntactic Errors**
  - Misplaced semicolons
  - Extra or missing braces
- **Semantic Errors**
  - Type mismatch between operators and operands
- **Logical Errors**
  - Incorrect reasoning like use of assignment operator = instead of the comparison operator ==

Syntax Error Handling

- The precision of parsing methods allows syntactic errors to be detected very efficiently.
- Several parsing methods, such as the LL and LR methods, detect an error as soon as possible; that is, when the stream of tokens from the lexical analyzer cannot be parsed further according to the grammar for the language.
- More precisely, they have the **viable-prefix property**, meaning that they detect that an error has occurred as soon as they see a prefix of the input that cannot be completed to form a string in the language.

Syntax Error Handling

- Another reason for emphasizing error recovery during parsing is that many errors appear syntactic, whatever their cause, and are exposed when parsing cannot continue.
- A few semantic errors, such as type mismatches, can also be detected efficiently; however, accurate detection of semantic and logical errors at compile time is in general a difficult task.

Goals of error handler in a parser

- ✓ Report the presence of errors clearly and accurately.
- ✓ Recover from each error quickly enough to detect subsequent errors.
- ✓ Add minimal overhead to the processing of correct programs.

- It must report the place in the source program where an error is detected, because there is a good chance that the actual error occurred within the previous few tokens.
- A common strategy is **to print the offending line with a pointer to the position** at which an error is detected.

Error Recovery Strategies

1. Panic Mode Recovery
2. Phrase-Level Recovery
3. Error Productions
4. Global Correction

Panic-Mode Recovery

- On discovering an error, the parser discards input symbols one at a time until one of a designated set of synchronizing tokens is found.
- The synchronizing tokens are usually delimiters, such as semicolon or }, whose role in the source program is clear and unambiguous.
- The compiler designer must select the synchronizing tokens appropriate for the source language.
- While panic-mode correction often skips a considerable amount of input without checking it for additional errors, it has the advantage of simplicity, and is guaranteed not to go into an infinite loop.

Phrase-Level Recovery

- On discovering an error, a parser may perform local correction on the remaining input; that is, it may replace a prefix of the remaining input by some string that allows the parser to continue.
- A typical local correction is to replace a comma by a semicolon, delete an extraneous semicolon, or insert a missing semicolon.
- Phrase-level replacement has been used in several error-repairing compilers, as it can correct any input string.
- Its major drawback is the difficulty it has in coping with situations in which the actual error has occurred before the point of detection.

Error Production

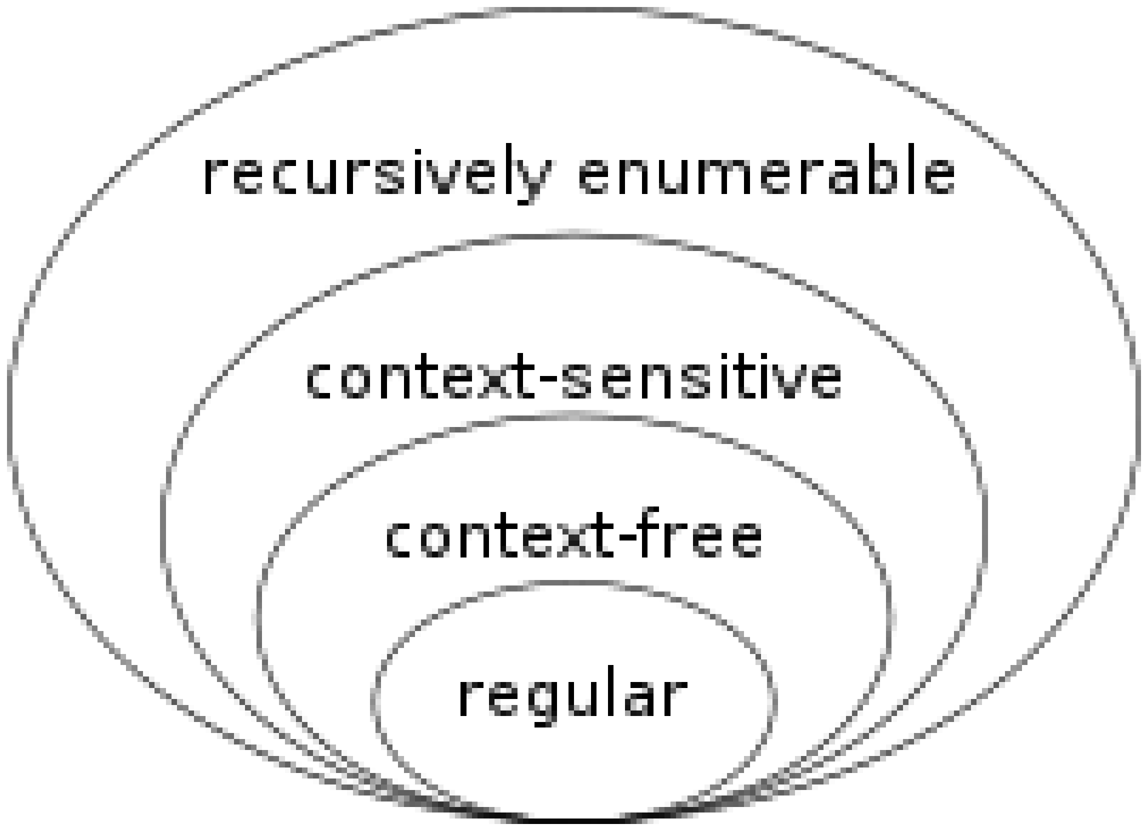
- By anticipating common errors that might be encountered, we can augment the grammar for the language at hand with productions that generate the erroneous constructs.
- A parser constructed from a grammar augmented by these error productions detects the anticipated errors when an error production is used during parsing.
- The parser can then generate appropriate error diagnostics about the erroneous construct that has been recognized in the input.

Global Correction

- Ideally, we would like a compiler to make as few changes as possible in processing an incorrect input string.
- There are algorithms for choosing a minimal sequence of changes to obtain a globally least-cost correction.
- Given an incorrect input string x and grammar G, these algorithms will find a parse tree for a related string y, such that the number of insertions, deletions, and changes of tokens required to transform x into y is as small as possible.
- Unfortunately, these methods are in general too costly to implement in terms of time and space, so these techniques are currently only of theoretical interest.
- Do note that a closest correct program may not be what the programmer had in mind.

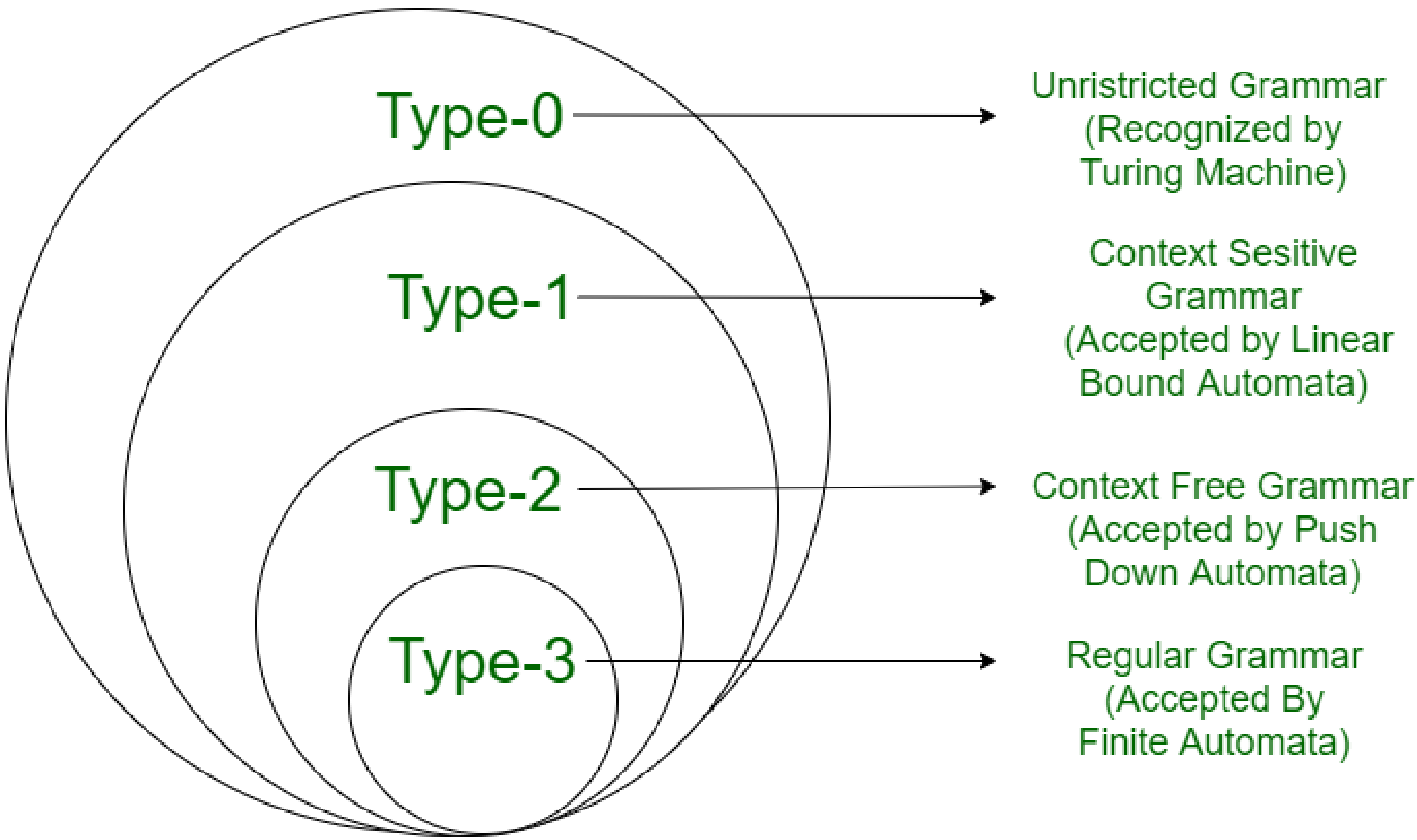
Context-Free Grammars

Chomsky hierarchy



https://en.wikipedia.org/wiki/Chomsky_hierarchy

Chomsky hierarchy



https://www.geeksforgeeks.org/chomsky-hierarchy-in-theory-of-computation/

The formal definition of Context-free grammars

A context-free grammar (grammar for short) consists

1. The **Terminals** are the basic symbols from which strings are formed.
2. The **Nonterminals** are syntactic variables that denote sets of strings.
   – The sets of strings denoted by nonterminals help define the language generated by the grammar.
   – Nonterminals impose a hierarchical structure on the language that is key to syntax analysis and translation.
3. In a grammar, one nonterminal is distinguished as the **start symbol**, and the set of strings it denotes is the language generated by the grammar.
   – Conventionally, the productions for the start symbol are listed first.
4. The **productions of a grammar** specify the manner in which the terminals and nonterminals can be combined to form strings.

The formal definition of Context-free grammars

Each production consists of:

1. A nonterminal called **the head or left side of the production**; this production defines some of the strings denoted by the head.
2. The **symbol →.**
    - Sometimes ::= has been used in place of the arrow.
3. A **body or right side** consisting of zero or more terminals and nonterminals.
    - The components of the body describe one way in which strings of the nonterminal at the head can be constructed.

Grammar for simple arithmetic expressions

expression → expression + term

expression → expression - term

expression → term

term → term * factor

term → term / factor

term → factor

factor → (expression)

factor → id

Conventions

- These symbols are **terminals:**
  - Lowercase letters early in the alphabet, such as a, b, c.
  - Operator symbols such as +, *, and so on.
  - Punctuation symbols such as parentheses, comma, and so on.
  - The digits 0; 1,…, 9.
  - Boldface strings such as **id** or **if**, each of which represents a single terminal symbol.
- These symbols are **nonterminals:**
  - Uppercase letters early in the alphabet, such as A, B, C.
  - The letter S, which, when it appears, is usually the start symbol.
  - Lowercase, italic names such as *expr* or *stmt*.
  - When discussing programming constructs, uppercase letters may be used to represent nonterminals for the constructs.

Conventions

- Uppercase letters late in the alphabet, such as X, Y, Z, represent grammar symbols; that is, either nonterminals or terminals.
- Lowercase letters late in the alphabet, chiefly u, v, …,z, represent (possibly empty) strings of terminals.
- Lowercase Greek letters α, β, γ, for example represent (possibly empty) string s of grammar symbols.
- Unless stated otherwise, **the head of the first production is the start symbol.**

Some context-free grammars

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

- $S \rightarrow 0S0$
  $S \rightarrow 1S1$
  $S \rightarrow 0$
  $S \rightarrow 1$
  $S \rightarrow \varepsilon$

- $S \rightarrow aSb$
  $S \rightarrow \varepsilon$

- $S \rightarrow aB|bA$
  $A \rightarrow a \mid aS \mid bAA$
  $B \rightarrow b \mid bS \mid aBB$

Sentinel Form (Left most derivation)

- $E \rightarrow E + E$

  $E \rightarrow E * E$

  $E \rightarrow (E)$

  $E \rightarrow id$

- Here, we replace the left most non-terminal of production by appropriate grammar rule. This is called **left most derivation**.

For input: a + b * c

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

Canonical derivation (Right most derivation)

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

- Here, we replace the right most non-terminal of production by appropriate grammar rule. This is called **right most derivation**.

For input: a + b * c

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

Ambiguity

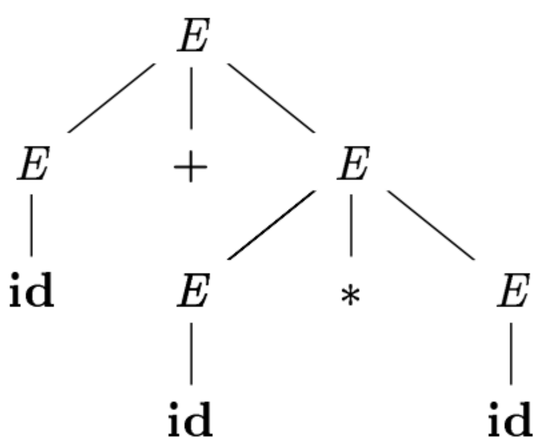
- A grammar that produces more than one parse tree for some sentence is said to be ambiguous.
- Put another way, an ambiguous grammar is one that produces more than one leftmost derivation or more than one rightmost derivation for the same sentence.
- Consider the grammar: E → E + E | E * E | (E) | **id**
- Input **id + id *id** generates two distinct leftmost derivatives:

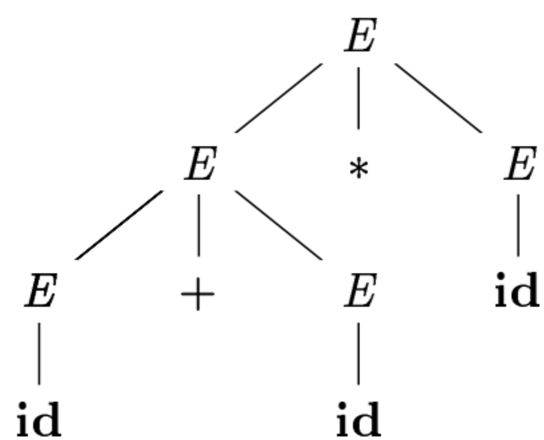
$$\begin{array}{lll} E & \Rightarrow & E+E \\ & \Rightarrow & \mathbf{id}+E \\ & \Rightarrow & \mathbf{id}+E*E \\ & \Rightarrow & \mathbf{id}+\mathbf{id}*E \\ & \Rightarrow & \mathbf{id}+\mathbf{id}*\mathbf{id} \end{array} \qquad \begin{array}{lll} E & \Rightarrow & E*E \\ & \Rightarrow & E+E*E \\ & \Rightarrow & \mathbf{id}+E*E \\ & \Rightarrow & \mathbf{id}+\mathbf{id}*E \\ & \Rightarrow & \mathbf{id}+\mathbf{id}*\mathbf{id} \end{array}$$

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

input: id + id *id

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

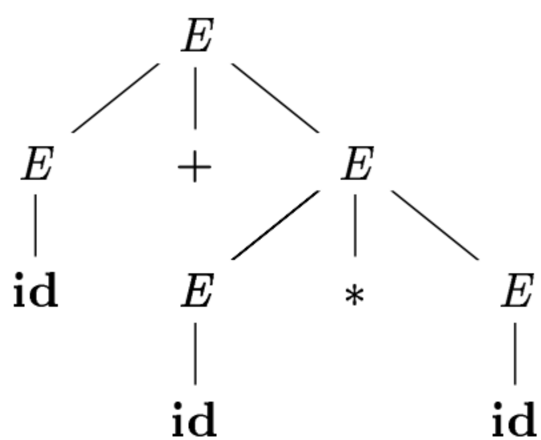


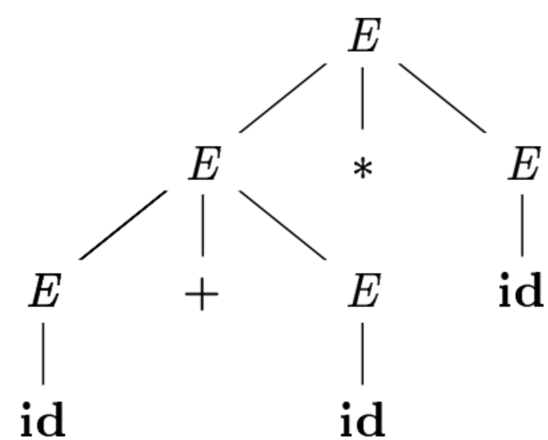


$E \rightarrow E + E \mid E * E \mid (E) \mid \textbf{id}$

input: id + id *id

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





Why use regular expressions to define the lexical syntax of a language?

1. Separating the syntactic structure of a language into lexical and nonlexical parts provides a convenient way of modularizing the front end of a compiler into two manageable-sized components.
2. The lexical rules of a language are frequently quite simple, and to describe them we do not need a notation as powerful as grammars.
3. Regular expressions generally provide a more concise and easier-to-understand notation for tokens than grammars.
4. More efficient lexical analyzers can be constructed automatically from regular expressions than from arbitrary grammars.

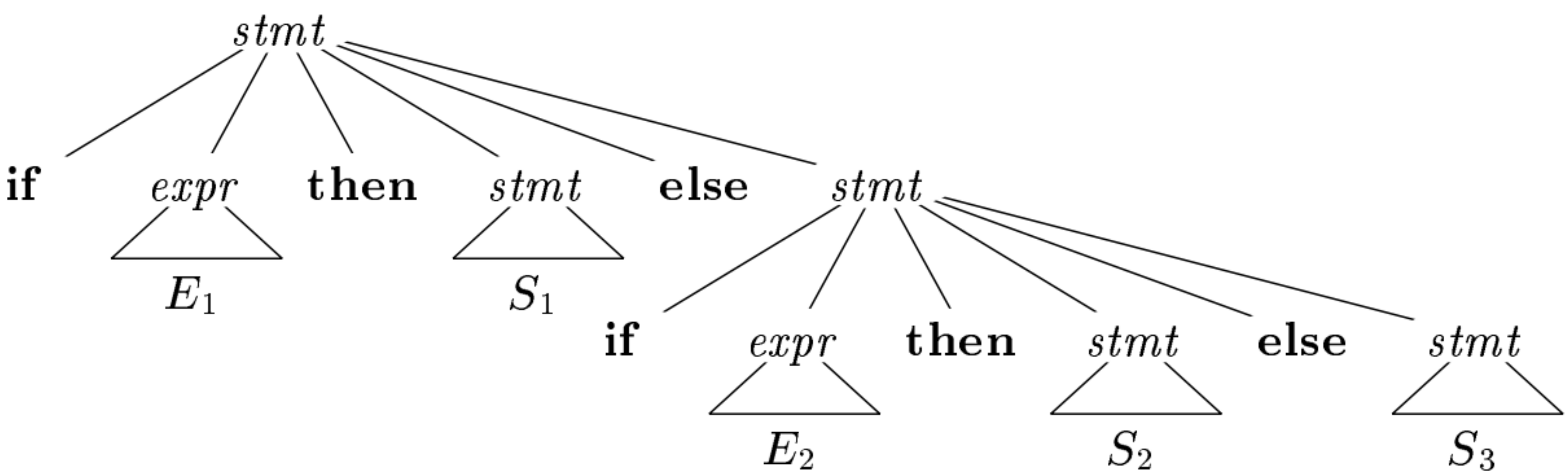
Observe the grammar

stmt	$\rightarrow$	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *stmt* **else** *stmt*
	|	other

Here "other" stands for any other statement.

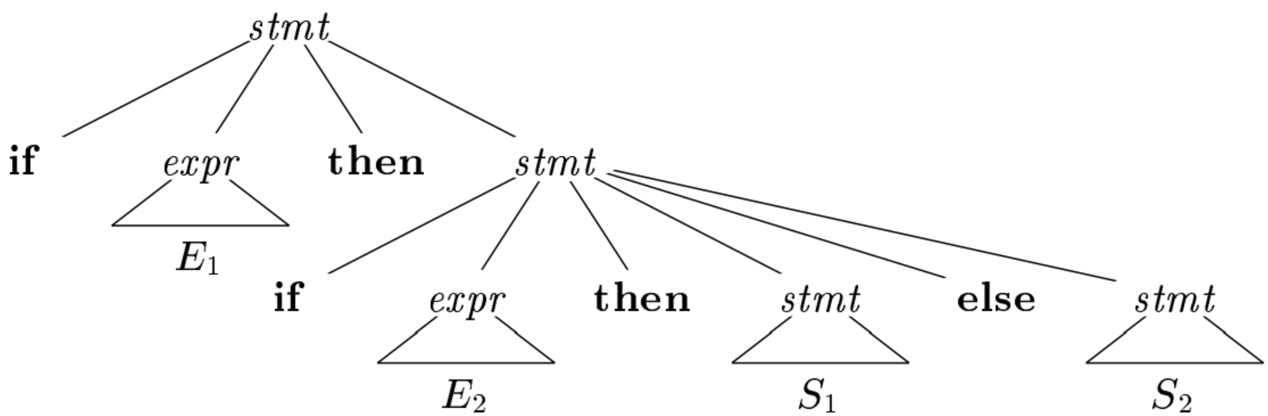
stmt	$\rightarrow$	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *stmt* **else** *stmt*
	|	other

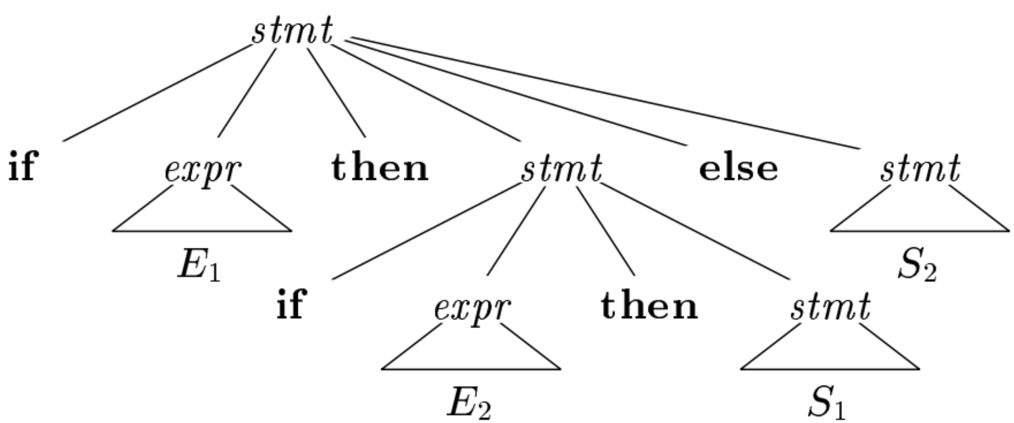
$\textbf{if } E_1 \textbf{ then } S_1 \textbf{ else if } E_2 \textbf{ then } S_2 \textbf{ else } S_3$



stmt	→	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *stmt* **else** *stmt*
	|	other

if E_1 **then** **if** E_2 **then** S_1 **else** S_2





Re-writing dangling else grammar

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

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

- Note:
  - No general techniques for handling ambiguity
  - Impossible to convert automatically an ambiguous grammar to an unambiguous one

Removing Useless Variables

S → abS | abA | abB

A → cd

B → aB

C → dc

?

Removing Useless Variables

S → abS | abA | abB

A → cd

B → aB

C → dc

S → abS | abA | abB

A → cd

B → aB

C is Non-reachable so remove it.

Removing Useless Variables

S → abS | abA | abB

A → cd

B → aB

C → dc

S → abS | abA | abB

A → cd

B → aB

B will never lead to a terminal.

S → abS | abA

A → cd

Removing Useless Variables

S → abS | abA | abB

A → cd

B → aB

C → dc

S → abS | abA | abB

A → cd

B → aB

B will never lead to a terminal.

S → abS | abA

A → cd

Remove useless variables

1. $S \rightarrow AB/a$
   $A \rightarrow BC/b$
   $B \rightarrow aB/C$
   $C \rightarrow aC/B$

2. $S \rightarrow AB/AC$
   $A \rightarrow aAb/bAa/a$
   $B \rightarrow bbA/aaB/AB$
   $C \rightarrow abCA/aDb$
   $D \rightarrow bD/aC$

3. $S \rightarrow aB$ / bX
   $A \rightarrow Bad$ / bSX / a
   $B \rightarrow aSB$ / bBX
   $X \rightarrow SBD$ / aBx / ad

4. $S \rightarrow AB|a$
   $A \rightarrow b$

5. $S \rightarrow AB|CA$
   $B \rightarrow BC \mid AB$
   $A \rightarrow a$
   $C \rightarrow AB|b$

Remove useless variables

1. S → AB/a
   A → BC/b
   B → aB/C
   C → aC/B

S → AB/a
A → BC/b
B → aB/C
C → aC/B

S → AB/a
A → BC/b
B → aB/C
C → aC/B

S → a
A → b

S → a

Remove useless variables

2. S → AB/AC
   A → aAb/bAa/a
   B → bbA/aaB/AB
   C → abCA/aDb
   D → bD/aC

S → AB/AC
A → aAb/bAa/a
B → bbA/aaB/AB
C → abCA/aDb
D → bD/aC

S → AB/AC
A → aAb/bAa/a
B → bbA/aaB/AB
C → abCA/aDb
D → bD/aC

S → AB
A → aAb/bAa/a
B → bbA/aaB/AB

Remove useless variables

3. S → aB / bX
A → Bad / bSX / a
B → aSB / bBX
X → SBD / aBx / ad

S → aB / bX
A → Bad / bSX / a
B → aSB / bBX
X → SBD / aBx / ad

S → aB / bX
A → Bad / bSX / a
B → aSB / bBX
X → SBD / aBx / ad

S → bX
A → bS X / a
X → ad

S → bX
X → ad

Remove useless variables

4. $S \rightarrow AB \mid a$
   $A \rightarrow b$

$S \rightarrow AB \mid a$
$A \rightarrow b$

$S \rightarrow AB \mid a$
$A \rightarrow b$

$S \rightarrow a$

Remove useless variables

5. $S \rightarrow AB \mid CA$

   $B \rightarrow BC \mid AB$

   $A \rightarrow a$

   $C \rightarrow AB \mid b$

$S \rightarrow AB \mid CA$

$B \rightarrow BC \mid AB$

$A \rightarrow a$

$C \rightarrow AB \mid b$

$S \rightarrow AB \mid CA$

$B \rightarrow BC \mid AB$

$A \rightarrow a$

$C \rightarrow AB \mid b$

$S \rightarrow CA$

$A \rightarrow a$

$C \rightarrow b$

Elimination of Left Recursion

- A Grammar G (V, T, P, S) is left recursive if it has a production in the form.

  $A \rightarrow A\ \alpha \mid \beta$

- The above Grammar is left recursive because the left of production is occurring at a first position on the right side of production.
- It can eliminate left recursion by replacing a pair of production with

  $A \rightarrow \beta A'$

  $A' \rightarrow \alpha A' \mid \epsilon$

Elimination of Left Recursion

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

$E \rightarrow TE'$
$E' \rightarrow +TE' | \epsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' | \epsilon$
$F \rightarrow (E) | id$

- Comparing $E \rightarrow E + T \; | T$ with $A \rightarrow A \alpha \; | \beta$.
- Here, A is E, α is +T and β is T
- On eliminating left recursion, using

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' | \epsilon$$

$E \rightarrow TE'$
$E' \rightarrow +TE' | \epsilon$

- Similarly for $T \rightarrow T * F | F$

$T \rightarrow FT'$
$T' \rightarrow *FT' | \epsilon$

Removal of left recursion

- $S \rightarrow a|\wedge|(T)$

  $T \rightarrow T, S|S$

Removal of left recursion

- $S \rightarrow a|\wedge|(T)$

  $T \rightarrow T, S|S$

$A \rightarrow A\,\alpha\;|\beta$

After removal,

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A'|\epsilon$

- Comparing $T \rightarrow T, S|S$ with $A \rightarrow A\,\alpha\;|\beta$.
- Here, A is T, α is ,S and β is S
- On eliminating left recursion,

  $T \rightarrow ST'$

  $T' \rightarrow ,ST'\;|\epsilon$
- So, finally

  $S \rightarrow a|\wedge|(T)$

  $T \rightarrow ST'$

  $T' \rightarrow ,ST'\;|\epsilon$

Remove left recursion

- $S \rightarrow Aa \mid b$

  $A \rightarrow Ac \mid Sd \mid \epsilon$

Remove left recursion

- $S \rightarrow Aa \mid b$

  $A \rightarrow Ac \mid Sd \mid \epsilon$

$A \rightarrow A\,\alpha \mid \beta$

After removal,

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$

- Eliminating the indirect left recursion.

  $S \rightarrow Aa \mid b$

  $A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$

- In $A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$,

  A is A

  α is c, ad

  β is $bd \mid \epsilon$

- So,

  $A \rightarrow (bd \mid \epsilon)A' \rightarrow A' \mid bdA'$

  $A' \rightarrow cA' \mid adA' \mid \epsilon$

Remove left recursion

- $S \rightarrow Aa \mid b$

  $A \rightarrow Ac \mid Sd \mid \epsilon$

- Finally,

  $S \rightarrow Aa \mid b$

  $A \rightarrow A' \mid bdA'$

  $A' \rightarrow cA' \mid adA' \mid \epsilon$

- Eliminating the indirect left recursion.

  $S \rightarrow Aa \mid b$

  $A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$

- In $A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$,

  A is A

  α is c, ad

  β is $bd \mid \epsilon$

- So,

  $A \rightarrow (bd \mid \epsilon)A' \rightarrow A' \mid bdA'$

  $A' \rightarrow cA' \mid adA' \mid \epsilon$

Remove left recursion (try yourself)

- $S \rightarrow Sa \mid Sb \mid c \mid d$

- $A \rightarrow Br$

  $B \rightarrow Cd$

  $C \rightarrow At$

Elimination of Left factoring

- Left factoring is removing the common left factor that appears in two productions of the same non-terminal.
- It is done to avoid back-tracing by the parser.

- $A \rightarrow a\alpha 1 \mid a\alpha 2 \mid a\alpha 3$

  Here, a is a common prefix or factor.

- After removal,

  $A \rightarrow aA'$

  $A' \rightarrow \alpha 1 \mid \alpha 2 \mid \alpha 3$

Remove left factoring

- $A \rightarrow aAB \,/\, aBc \,/\, aAc$

$A \rightarrow a\alpha1 \mid a\alpha2 \mid a\alpha3$

After removal,

$A \rightarrow aA'$

$A' \rightarrow \alpha1 \mid \alpha2 \mid \alpha3$

- Here, common prefix a is observed.

  $A \rightarrow aA'$

  $A' \rightarrow AB \mid Bc \mid Ac$

- Again common prefix A is observed in $A' \rightarrow AB \mid Bc \mid Ac$

  $A' \rightarrow AD \mid Bc$

  $D \rightarrow B \mid c$

Remove left factoring

- $A \rightarrow aAB$ / aBc / aAc
- Here, common prefix a is observed.

  $A \rightarrow aA'$

  $A' \rightarrow AB \mid Bc \mid Ac$
- Again common prefix A is observed in $A' \rightarrow AB \mid Bc \mid Ac$

  $A' \rightarrow AD \mid Bc$

  $D \rightarrow B \mid c$
- Finally we have,

  $A \rightarrow aA'$

  $A' \rightarrow AD \mid Bc$

  $D \rightarrow B \mid c$

Remove left factoring

- $S \rightarrow bSSaaS/bSSaSb/bSb/a$

Remove left factoring

- S → bSSaaS/bSSaSb/bSb/a

- Here common prefix bS is observed.

  S → bSS' | a

  S' → SaaS | SaSb |b

- Again, Sa common prefix is observed

  S' → SaA | b

  A → aS | Sb

A → aα1 | aα2 | aα3

After removal,

A → aA'

A' → α1 | α2 |α3

Remove left factoring

- S → bSSaaS/bSSaSb/bSb/a
- Here common prefix bS is observed.

  S → bSS' | a

  S' → SaaS | SaSb | b
- Finally, we have

  S → bSS' | a

  S' → SaA | b

  A → aS | Sb
- Again, Sa common prefix is observed

  S' → SaA | b

  A → aS | Sb

Remove left factoring

- $S \rightarrow iEtS \mid iEtSeS \mid a$

  $E \rightarrow b$

Remove left factoring

- $S \rightarrow iEtS \mid iEtSeS \mid a$

  $E \rightarrow b$

$A \rightarrow a\alpha1 \mid a\alpha2 \mid a\alpha3$

After removal,

$A \rightarrow aA'$

$A' \rightarrow \alpha1 \mid \alpha2 \mid \alpha3$

- Here, common prefix iEtS is observed

  $S \rightarrow iEtSS' \mid a$

  $S' \rightarrow eS \mid \epsilon$

- So, finally we have,

  $S \rightarrow iEtSS' \mid a$

  $S' \rightarrow eS \mid \epsilon$

  $E \rightarrow b$

Remove left factoring

- S → aSSbS | aSaSb | abb |b

Remove left factoring

- S → aSSbS | aSaSb | abb |b
- Common prefix a is observed.

  S → aS' | b

  S' → SSbS | SaSb | bb
- Again common prefix S is observed.

  S' → SA | bb

  A → SbS | aSb

A → aα1 | aα2 | aα3

After removal,

A → aA'

A' → α1 | α2 |α3

Remove left factoring

- $S \rightarrow aSSbS \mid aSaSb \mid abb \mid b$
- Common prefix a is observed.

  $S \rightarrow aS' \mid b$

  $S' \rightarrow SSbS \mid SaSb \mid bb$
- Again common prefix S is observed.

  $S' \rightarrow SA \mid bb$

  $A \rightarrow SbS \mid aSb$
- Finally we have,

  $S \rightarrow aS' \mid b$

  $S' \rightarrow SA \mid bb$

  $A \rightarrow SbS \mid aSb$

Remove left factoring

- S → a | ab | abc | abcd

Remove left factoring

- $S \rightarrow a \mid ab \mid abc \mid abcd$

$A \rightarrow a\alpha1 \mid a\alpha2 \mid a\alpha3$

After removal,

$A \rightarrow aA'$

$A' \rightarrow \alpha1 \mid \alpha2 \mid \alpha3$

- Common prefix a is observed

  $S \rightarrow aS'$

  $S' \rightarrow \epsilon \mid b \mid bc \mid bcd$

- Common prefix b is observed

  $S' \rightarrow \epsilon \mid bA$

  $A \rightarrow \epsilon \mid c \mid cd$

- Common prefix c is observed

  $A \rightarrow \epsilon \mid cB$

  $B \rightarrow \epsilon \mid d$

Remove left factoring

- $S \rightarrow a \mid ab \mid abc \mid abcd$

- Finally we have

  $S \rightarrow aS'$

  $S' \rightarrow bA \mid \epsilon$

  $A \rightarrow cB \mid \epsilon$

  $B \rightarrow d \mid \epsilon$

- Common prefix a is observed

  $S \rightarrow aS'$

  $S' \rightarrow \epsilon \mid b \mid bc \mid bcd$

- Common prefix b is observed

  $S' \rightarrow \epsilon \mid bA$

  $A \rightarrow \epsilon \mid c \mid cd$

- Common prefix c is observed

  $A \rightarrow \epsilon \mid cB$

  $B \rightarrow \epsilon \mid d$

Remove left factoring

- $S \rightarrow aAd \mid aB$

  $A \rightarrow a \mid ab$

  $B \rightarrow ccd \mid ddc$

Remove left factoring

- S → aAd | aB

  A → a | ab

  B → ccd | ddc

⇒

- S → aS'

  S' → Ad | B

  A → aA'

  A' → b | ∈

  B → ccd | ddc

Elimination of ε productions

- Steps:
  - To remove A → ε, look for all productions whose right side contains A
  - Replace each occurrence of 'A' in each of these productions with ε
  - Add the resultant productions to the grammar

https://www.sanfoundry.com/automata-theory-cfg-eliminating-null-productions/

Elimination of ε productions

- $S \rightarrow ABA$

  $A \rightarrow aA \mid \varepsilon$

  $B \rightarrow bB \mid \varepsilon$

Elimination of ε productions

- $S \rightarrow ABA$
  $A \rightarrow aA \mid \varepsilon$
  $B \rightarrow bB \mid \varepsilon$

- As A and B are directly nullable variables:

  $A \rightarrow aA \mid a$

  $B \rightarrow bB \mid b$

  $S \rightarrow ABA \mid AB \mid BA \mid AA \mid A \mid B$

Elimination of ε productions

- $S \rightarrow aS \mid AB \mid a$

  $A \rightarrow \varepsilon$

  $B \rightarrow \varepsilon$

  $D \rightarrow b$

Elimination of ε productions

- $S \rightarrow aS \mid AB \mid a$

  $A \rightarrow \varepsilon$

  $B \rightarrow \varepsilon$

  $D \rightarrow b$

- As A and B are directly nullable variables:

$S \rightarrow aS \mid a$

$D \rightarrow b$

$S \rightarrow aS \mid a$

Remove null productions

- $S \rightarrow ABAC$
  $A \rightarrow aA \,/\, \epsilon$
  $B \rightarrow bB \,/\, \epsilon$
  $C \rightarrow c$

Remove null productions

- S $\rightarrow$ ABAC
  A $\rightarrow$ aA / ϵ
  B $\rightarrow$ bB / ϵ
  C $\rightarrow$ c

$$\Downarrow$$

- S $\rightarrow$ ABAC / ABC / BAC / BC / AAC / AC / C
  A $\rightarrow$ aA / a
  B $\rightarrow$ bB / b
  C $\rightarrow$ c

Remove null productions

- $S \rightarrow ABCd$

  $A \rightarrow BC$

  $B \rightarrow bB \mid \epsilon$

  $C \rightarrow cC \mid \epsilon$

Remove null productions

- $S \rightarrow ABCd$

  $A \rightarrow BC$

  $B \rightarrow bB \mid \epsilon$

  $C \rightarrow cC \mid \epsilon$

- $S \rightarrow ABCd \mid ABd \mid ACd \mid BCd \mid Ad \mid Bd \mid Cd \mid d$

  $A \rightarrow BC \mid B \mid C$

  $B \rightarrow bB \mid b$

  $C \rightarrow cC \mid c$

Remove null productions

- $S \rightarrow a|Ab|aBa$

  $A \rightarrow b|\epsilon$

  $B \rightarrow b|A$

Remove null productions

- $S \rightarrow a|Ab|aBa$

  $A \rightarrow b|\epsilon$

  $B \rightarrow b|A$

$$\Downarrow$$

- $S \rightarrow a|Ab|b|aBa|aa$

  $A \rightarrow b$

  $B \rightarrow b$

Remove null productions & unit productions

- S → a | Ab | aBa

  A → b | ϵ

  B → b | A

- S → a | Ab | b | aBa | aa

  A → b

  B → b

- S→ a | bb | b | aba | aa

Remove unit productions

- Steps:
  1. To remove $X \rightarrow Y$, add production $X \rightarrow a$ to the grammar rule whenever $Y \rightarrow a$ occurs in the grammar
  2. Now delete $X \rightarrow Y$ from the grammar
  3. Repeat Step 1 and 2 until all unit productions are removed

Remove unit productions

- $S \rightarrow 0A \mid 1B \mid C$

  $A \rightarrow 0S \mid 00$

  $B \rightarrow 1 \mid A$

  $C \rightarrow 01$

Remove unit productions

- S → 0A | 1B | C

  A → 0S |00

  B → 1 | A

  C → 01

- S → C is a unit production
- S → 0A | 1B | 01

  A → 0S |00

  B → 1 | A

  C → 01

Remove unit productions

- S → 0A | 1B | C
  A → 0S |00
  B → 1 | A
  C → 01

- S → C is a unit production
- S → 0A | 1B | 01
  A → 0S |00
  B → 1 | A
  C → 01
- B → A is also a unit production
- S → 0A |1B |01
- A → 0S | 00
- B → 1 |0S | 00
- C → 01

Remove unit productions

- S → 0A | 1B | C
  A → 0S |00
  B → 1 | A
  C → 01

- S → 0A |1B |01
  A → 0S | 00
  B → 1 |0S | 00
  C → 01

- S → C is a unit production
- S → 0A | 1B | 01
  A → 0S |00
  B → 1 | A
  C → 01
- B → A is also a unit production
- S → 0A |1B |01
  A → 0S | 00
  B → 1 |0S | 00
  C → 01

Remove unit productions

- $S \rightarrow Aa/B/c$
  $B \rightarrow A/bb$
  $A \rightarrow a/bc/B$

Remove unit productions

- $S \rightarrow Aa/B/c$

  $B \rightarrow A/bb$

  $A \rightarrow a/bc/B$

- First writing the productions with unit productions
- $S \rightarrow B$

  $B \rightarrow A$

  $A \rightarrow B$

Remove unit productions

- $S \rightarrow Aa/B/c$

  $B \rightarrow A/bb$

  $A \rightarrow a/bc/B$

- First writing the productions with unit productions
- For the production $S \rightarrow B$

  $S \rightarrow B \rightarrow bb$

  $S \rightarrow B \rightarrow A \rightarrow a$

  $S \rightarrow B \rightarrow A \rightarrow bc$

  $B \rightarrow A$

  $A \rightarrow B$

Remove unit productions

- $S \rightarrow Aa/B/c$

  $B \rightarrow A/bb$

  $A \rightarrow a/bc/B$

- First writing the productions with unit productions
- For the production $S \rightarrow B$

  $S \rightarrow B \rightarrow bb$

  $S \rightarrow B \rightarrow A \rightarrow a$

  $S \rightarrow B \rightarrow A \rightarrow bc$

- For the production $B \rightarrow A$

  $B \rightarrow A \rightarrow a$

  $B \rightarrow A \rightarrow bc$

  $A \rightarrow B$

Remove unit productions

- S → Aa/B/c
  B → A/bb
  A → a/bc/B

- First writing the productions with unit productions
- For the production S→B
  S→B →bb
  S →B→A→a
  S →B→A→bc
- For the production B→A
  B→A→a
  B→A→bc
- For the production A→B
  A→B→bb

Remove unit productions

- S → Aa/B/c

  B → A/bb

  A → a/bc/B

- First writing the productions with unit productions
- For the production S→B

  S→B →bb

  S →B→A→a

  S →B→A→bc
- For the production B→A

  B→A→a

  B→A→bc
- For the production A→B

  A→B→bb

Remove unit productions

- $S \rightarrow Aa/B/c$

  $B \rightarrow A/bb$

  $A \rightarrow a/bc/B$

- $S \rightarrow Aa \mid c \mid bb \mid a \mid bc$

  $B \rightarrow bb \mid a \mid bc$

  $A \rightarrow a \mid bc \mid bb$

- First writing the productions with unit productions
- For the production $S \rightarrow B$

  $S \rightarrow B \rightarrow bb$

  $S \rightarrow B \rightarrow A \rightarrow a$

  $S \rightarrow B \rightarrow A \rightarrow bc$
- For the production $B \rightarrow A$

  $B \rightarrow A \rightarrow a$

  $B \rightarrow A \rightarrow bc$
- For the production $A \rightarrow B$

  $A \rightarrow B \rightarrow bb$

Sequence of steps for elimination

1. Eliminate/Remove null production
2. Eliminate/Remove unit production
3. Eliminate/Remove useless symbol

Simplify the grammar

- $S \rightarrow Aa \mid B$

  $B \rightarrow a \mid bC$

  $C \rightarrow a \mid \varepsilon$

Simplify the grammar
Eliminate/Remove null production

- $S \rightarrow Aa \mid B$
  $B \rightarrow a \mid bC$
  $C \rightarrow a \mid \varepsilon$

- $C \rightarrow \varepsilon$ is a null production
- To remove it, add the production
- $B \rightarrow bC \rightarrow b\,\varepsilon \rightarrow b$

- So we have,
- $S \rightarrow Aa \mid B$
  $B \rightarrow a \mid bC \mid b$
  $C \rightarrow a$
- Now, no null productions.

Simplify the grammar
Eliminate/Remove null production

- $S \rightarrow Aa \mid B$
  $B \rightarrow a \mid bC$
  $C \rightarrow a \mid \varepsilon$

$\Downarrow$

- $S \rightarrow Aa \mid B$
  $B \rightarrow a \mid bC \mid b$
  $C \rightarrow a$

- $C \rightarrow \varepsilon$ is a null production
- To remove it, add the production
- $B \rightarrow bC \rightarrow b\,\varepsilon \rightarrow b$

- So we have,
- $S \rightarrow Aa \mid B$
  $B \rightarrow a \mid bC \mid b$
  $C \rightarrow a$
- Now, no null productions.

Simplify the grammar
Eliminate/Remove unit production

- $S \rightarrow Aa \mid B$
  $B \rightarrow a \mid bC$
  $C \rightarrow a \mid \varepsilon$

$$\Downarrow$$

- $S \rightarrow Aa \mid B$
  $B \rightarrow a \mid bC \mid b$
  $C \rightarrow a$

- Identifying unit productions
  $S \rightarrow B$

- Removing it, gives:
  $S \rightarrow B \rightarrow a \mid bC \mid b$

- So grammar is:
  $S \rightarrow Aa \mid a \mid bC \mid b$
  $B \rightarrow a \mid bC \mid b$
  $C \rightarrow a$

Simplify the grammar
Eliminate/Remove unit production

- $S \rightarrow Aa \mid B$

  $B \rightarrow a \mid bC$

  $C \rightarrow a \mid \varepsilon$

  $\Downarrow$

- $S \rightarrow Aa \mid B$

  $B \rightarrow a \mid bC \mid b$

  $C \rightarrow a$

  $\Downarrow$

- $S \rightarrow Aa \mid a \mid bC \mid b$

  $B \rightarrow a \mid bC \mid b$

  $C \rightarrow a$

- Identifying unit productions

  $S \rightarrow B$

- Removing it, gives:

  $S \rightarrow B \rightarrow a \mid bC \mid b$

- So grammar is:

  $S \rightarrow Aa \mid a \mid bC \mid b$

  $B \rightarrow a \mid bC \mid b$

  $C \rightarrow a$

- Now, no unit productions.

Simplify the grammar
Eliminate/Remove unit production

- S → Aa | B
  B → a | bC
  C → a | ε

⇩

- S → Aa | B
  B → a | bC | b
  C → a

⇩

- S → Aa | a | bC | b
  B → a | bC | b
  C → a

- Identifying unit productions
  S → B

- Removing it, gives:
  S → B → a | bC | b

- So grammar is:
  S → Aa | a | bC | b
  B → a | bC | b
  C → a

- Now, no unit productions.

Simplify the grammar
Eliminate/Remove useless variables

- S → Aa | B
  B → a | bC
  C → a | ε

⇩

- S → Aa | B
  B → a | bC | b
  C → a

⇩

- S → Aa | a | bC | b
  B → a | bC | b
  C → a

- B is a useless variable as cannot reach B from S.
- So remove it.
  S → Aa | a | bC | b
  C → a

Simplify the grammar
Eliminate/Remove useless variables

- $S \rightarrow Aa \mid B$

  $B \rightarrow a \mid bC$

  $C \rightarrow a \mid \varepsilon$

$\Downarrow$

- $S \rightarrow Aa \mid B$

  $B \rightarrow a \mid bC \mid b$

  $C \rightarrow a$

$\Downarrow$

- $S \rightarrow Aa \mid a \mid bC \mid b$

  $B \rightarrow a \mid bC \mid b$

  $C \rightarrow a$

- B is a useless variable as cannot reach B from S.
- So remove it.

  $S \rightarrow Aa \mid a \mid bC \mid b$

  $C \rightarrow a$

- Now, Aa is useless as no production head A.
- So remove it.

  $S \rightarrow a \mid bC \mid b$

  $C \rightarrow a$

Simplify the grammar
Eliminate/Remove useless variables

- $S \rightarrow Aa \mid B$
  $B \rightarrow a \mid bC$
  $C \rightarrow a \mid \varepsilon$

$\Downarrow$

- $S \rightarrow Aa \mid B$
  $B \rightarrow a \mid bC \mid b$
  $C \rightarrow a$

$\Downarrow$

- $S \rightarrow Aa \mid a \mid bC \mid b$
  $B \rightarrow a \mid bC \mid b$
  $C \rightarrow a$

- B is a useless variable as cannot reach B from S.
- So remove it.
  $S \rightarrow Aa \mid a \mid bC \mid b$
  $C \rightarrow a$

- Now, Aa is useless as no production head A.
- So remove it.
  $S \rightarrow a \mid bC \mid b$
  $C \rightarrow a$

- Now, $C \rightarrow a$ can be substituted
  $S \rightarrow a \mid ba \mid b$

Simplify the grammar
Eliminate/Remove useless variables

- $S \rightarrow Aa \mid B$
  $B \rightarrow a \mid bC$
  $C \rightarrow a \mid \varepsilon$

$\Downarrow$

- $S \rightarrow Aa \mid B$
  $B \rightarrow a \mid bC \mid b$
  $C \rightarrow a$

$\Downarrow$

- $S \rightarrow Aa \mid a \mid bC \mid b$
  $B \rightarrow a \mid bC \mid b$
  $C \rightarrow a$

- B is a useless variable as cannot reach B from S.
- So remove it.
  $S \rightarrow Aa \mid a \mid bC \mid b$
  $C \rightarrow a$

- Now, Aa is useless as no production head A.
- So remove it.
  $S \rightarrow a \mid bC \mid b$
  $C \rightarrow a$

- Now, $C \rightarrow a$ can be substituted
  $\boxed{S \rightarrow a \mid ba \mid b}$

Simplify the grammar
Eliminate/Remove useless variables

- $S \rightarrow Aa \mid B$
  $B \rightarrow a \mid bC$
  $C \rightarrow a \mid \varepsilon$

$\Downarrow$

- $S \rightarrow Aa \mid B$
  $B \rightarrow a \mid bC \mid b$
  $C \rightarrow a$

$\Downarrow$

- $S \rightarrow Aa \mid a \mid bC \mid b$
  $B \rightarrow a \mid bC \mid b$
  $C \rightarrow a$

$\Rightarrow$

$\boxed{S \rightarrow a \mid ba \mid b}$