Recognition of strings in a language

Parser
A program that determines if a string \( w \in L(G) \) by constructing a derivation. Equivalently, it searches the graph of \( G \).
- Top-down parsers
  - Constructs the derivation tree from root to leaves.
  - Leftmost derivation.
- Bottom-up parsers
  - Constructs the derivation tree from leaves to root.
  - Rightmost derivation in reverse.

Parse trees (=Derivation Tree)
A parse tree is a graphical representation of a derivation sequence of a sentential form.

Tree nodes represent symbols of the grammar (nonterminals or terminals) and tree edges represent derivation steps.

Example:
Given the following grammar:
\[
E \rightarrow E + E \mid E \ast E \mid (E) \mid -E \mid \text{id}
\]
Is the string \(-\text{id} + \text{id}\) a sentence in this grammar?
Yes because there is the following derivation:
\[
E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(\text{id} + \text{id})
\]
FG: Parsing

**Example 1**

Consider the CFG grammar $G$

- $S \rightarrow SS | a^+b$  
- $a \rightarrow aT$  
- $b \rightarrow bT$  
- $T \rightarrow T \mid (A)$

Show that $(b)^+b \in L(G)$?

**Example 2**

Consider the CFG grammar $G$

- $S \rightarrow SS | a^+b$  
- $a \rightarrow aT$  
- $b \rightarrow bT$  
- $T \rightarrow T \mid (A)$

Show that $(b)^+b \in L(G)$?

**Practical Parsers**

- Language/Grammar designed to enable deterministic (directed and backtrack-free) searches.
  - Top-down parsers: LL(k) languages
    - E.g., Pascal, Ada, etc.
    - Better error diagnosis and recovery.
  - Bottom-up parsers: LALR(1), LR(k) languages
    - E.g., C/C++, Java, etc.
    - Handles left recursion in the grammar.
  - Backtracking parsers
    - E.g., Prolog interpreter.

**Flaws of Top-down Exhaustive Parsing**

- Obvious flaw: it will take a long time and a lot of memory for moderately long strings $w$. It is inefficient.
- For cases $w \in L(G)$, exhaustive parsing may never end.
- This will especially happen if we have rules like $A \rightarrow A^+$ that make the sentential forms ‘shrink’ so that we will never know if we went too far with our parsing attempts.
- Similar problems occur if the parsing can get in a loop according to $A \Rightarrow B \Rightarrow A \Rightarrow B$...
- Fortunately, it is always possible to remove problematic rules like $A \rightarrow A^+$ and $A \rightarrow B$ from a CFG $G$. 

**Top-down Exhaustive Parsing**

- Exhaustive parsing is a form of top-down parsing where you start with $S$ and systematically go through all possible (say leftmost) derivations until you produce the string $w$.
- You can remove sentential forms that will not work.
- Example: Can the CFG $G$?

  - $S \rightarrow SS | a^+b$  
  - $a \rightarrow aT$  
  - $b \rightarrow bT$  
  - $T \rightarrow T \mid (A)$

  Produce the string $w = aabb$, and how?
  - After one step: $S \Rightarrow SS \text{ or } aSb$
  - After two steps: $S \Rightarrow SS \Rightarrow aSb \text{ or } ab$
  - After three steps: $S \Rightarrow SS \Rightarrow aSb \Rightarrow aabb$.

**Examples of Top-down Exhaustive Parsing**

- Obvious flaw: it will take a long time and a lot of memory for moderately long strings $w$. It is inefficient.
- For cases $w \in L(G)$, exhaustive parsing may never end.
- This will especially happen if we have rules like $A \rightarrow A^+$ that make the sentential forms ‘shrink’ so that we will never know if we went too far with our parsing attempts.
- Similar problems occur if the parsing can get in a loop according to $A \Rightarrow B \Rightarrow A \Rightarrow B$...
- Fortunately, it is always possible to remove problematic rules like $A \rightarrow A^+$ and $A \rightarrow B$ from a CFG $G$. 

**Rightmost Derivation**

- $S \Rightarrow SS \Rightarrow SSb \Rightarrow ab$

**Leftmost Derivation**

- $S \Rightarrow SS \Rightarrow aS \Rightarrow ab$

**Derivation Trees**

- CFG: Parsing
  - Example 2
  - $S \rightarrow SS | a^+b$
  - $a \rightarrow aT$
  - $b \rightarrow bT$
  - $T \rightarrow T \mid (A)$

**Practical Parsers**

- Language/Grammar designed to enable deterministic (directed and backtrack-free) searches.
  - Top-down parsers: LL(k) languages
    - E.g., Pascal, Ada, etc.
    - Better error diagnosis and recovery.
  - Bottom-up parsers: LALR(1), LR(k) languages
    - E.g., C/C++, Java, etc.
    - Handles left recursion in the grammar.
  - Backtracking parsers
    - E.g., Prolog interpreter.
Definition: a string is derived ambiguously in a context-free grammar if it has two or more different parse trees.

Definition: a grammar is ambiguous if it generates some string ambiguously.

Grammar Ambiguity

A string $w \in L(G)$ is derived ambiguously if it has more than one derivation tree (or equivalently: if it has more than one leftmost derivation (or rightmost)).

A grammar is ambiguous if some strings are derived ambiguously.

Typical example: rule $S \rightarrow 0 \mid 1 \mid S+S \mid S \times S$.

$$
\begin{align*}
S &\Rightarrow S+S \\
S &\Rightarrow S \times S+S \\
S &\Rightarrow 0 \\
S &\Rightarrow 0 \times S+S \\
S &\Rightarrow 0 \times 1+S \\
S &\Rightarrow 0 \times 1+1
\end{align*}
$$

versus

$$
\begin{align*}
S &\Rightarrow S+S \\
S &\Rightarrow S \times S \\
S &\Rightarrow 0 \\
S &\Rightarrow 0 \times S+S \\
S &\Rightarrow 0 \times 1+S \\
S &\Rightarrow 0 \times 1+1
\end{align*}
$$

The ambiguity of $0 \times 1+1$ is shown by the two different parse trees:

Note that the two different derivations:

$$
\begin{align*}
S &\Rightarrow S+S \\
S &\Rightarrow S \times S+S \\
S &\Rightarrow S+1 \\
S &\Rightarrow 0+1
\end{align*}
$$

and

$$
\begin{align*}
S &\Rightarrow S+S \\
S &\Rightarrow S+1 \\
S &\Rightarrow 0+1
\end{align*}
$$

do not constitute an ambiguous string $0+1$ as they have the same parse tree.

Ambiguity causes troubles when trying to interpret strings like: "She likes men who love women who don’t smoke.”

Solutions: Use parentheses, or use precedence rules such as $a+(b\times c) = a+b \times c$.

Grammar Ambiguity

Inherently Ambiguous

Languages that can only be generated by ambiguous grammars are inherently ambiguous.

Example 5.13: $L = \{ab|c^m\} \cup \{ab|b^nc^m\}$.

$$L = \{a^ib^jc^k \mid i = j \lor j = k\}$$

The way to make a CFG for this $L$ somehow has to involve the step $S \rightarrow S_1|S_2$ where $S_1$ produces the strings $a^ib^jc^m$ and $S_2$ the strings $a^ib^jc^n$.

This will be ambiguous on strings $a^ib^jc^i$. 

Example

$$\begin{align*}
\text{<EXPR>} &\?\text{ <EXPR> + <EXPR>} \\
\text{<EXPR>} &\?\text{ <EXPR> \times <EXPR>} \\
\text{<EXPR>} &\?\text{ ( <EXPR> )} \\
\text{<EXPR>} &\?\text{ a}
\end{align*}$$

Build a parse tree for $a + a \times a$.
Which derivation tree is correct?

Find a derivation for the expression: \( \text{id} + \text{id} \times \text{id} \)

\[
E \rightarrow E + E \mid E \times E \mid (E) \mid -E \mid \text{id}
\]

According to the grammar, both are correct.

A grammar that produces more than one parse tree for any input sentence is said to be an ambiguous grammar.

One way to resolve ambiguity is to associate precedence to the operators.

- * has precedence over +
  
  \[
  1 + 2 \times 3 = 1 + (2 \times 3)
  \]
  
- Associativity and precedence information is typically used to disambiguate non-fully parenthesized expressions containing unary prefix/postfix operators or binary infix operators.

Is the following grammar ambiguous?

\[
S \rightarrow PC \mid AQ
P \rightarrow aPb \mid \lambda
C \rightarrow cC \mid \lambda
Q \rightarrow bQc \mid \lambda
A \rightarrow aA \mid \lambda
\]

Yes: consider the string abc

Is the following grammar ambiguous?

\[
S \rightarrow aS \mid Sb \mid ab
\]

Yes: consider ab
Grammar Ambiguity

Is the following grammar ambiguous? \( S \rightarrow SS | \lambda \)

Yes

(IIllustrates ambiguous grammar with cycles.)

Simple Grammar

Definition

A CFG \( (V,T,S,P) \) is a **simple grammar** \( (s-grammar) \) if and only if all its productions are of the form \( A \rightarrow ax \) with \( A \in V \), \( a \in T \), \( x \in V^* \) and any pair \( (A,a) \) occurs at most once.

- Note, for simple grammars a left most derivation of a string \( w \in L(G) \) is straightforward and requires time \( |w| \).
- Example: Take the s-grammar \( S \rightarrow aS|bSS|aSS|c \)
  
  \[ S \Rightarrow aS \Rightarrow aaaS \Rightarrow aabSS \Rightarrow aabcS \Rightarrow aabcc. \]

Quiz: is the grammar \( S \rightarrow aS|bSS|aSS|c \) a **s-grammar**?

**No**

Why?

The pair \((S,a)\) occurs twice

Normal Forms

Chomsky Normal Form
Griebach Normal Form

Chomsky Normal Form

Definition 6.4: A CFG is in **Chomsky normal form** if and only if all production rules are of the form

- \( A \rightarrow BC \)
- \( A \rightarrow x \)

with variables \( A,B,C \in V \) and \( x \in T \).

(Sometimes rule \( S \rightarrow ? \) is also allowed.)

CFGs in CNF can be parsed in time \( O(|w|^3) \).

Named after Noam Chomsky who in the 60s made seminal contributions to the field of theoretical linguistics. (cf. Chomsky hierarchy of languages.)

Chomsky Normal Form CNF

Even though we can’t get every grammar into right-linear form, or in general even get rid of ambiguity, there is an especially simple form that general CFG’s can be converted into:

Chomsky Normal Form CNF

Noam Chomsky came up with an especially simple type of context free grammars which is able to capture all context free languages.

Chomsky's grammatical form is particularly useful when one wants to prove certain facts about context free languages. This is because assuming a much more restrictive kind of grammar can often make it easier to prove that the generated language has whatever property you are interested in.
Significance of CNF

- Length of derivation of a string of length \( n \) in CNF = \((2^n - 1)\) (Cf. Number of nodes of a strictly binary tree with \( n \)-leaves)
- Maximum depth of a parse tree = \( n \)
- Minimum depth of a parse tree = \( \lceil \log_2 n \rceil + 1 \)

Chomsky Normal Form (CNF)

A CFG is said to be in Chomsky Normal Form if every rule in the grammar has one of the following forms:

- \( A \rightarrow BC \) (dyadic variable productions)
- \( A \rightarrow a \) (unit terminal productions)
- \( S \rightarrow \lambda \) (? for empty string sake only)

where \( B, C \in V - \{S\} \)

Where \( S \) is the start variable, \( A, B, C \) are variables and \( a \) is a terminal. Thus \( \lambda \) symbols may only appear on the right hand side of the start symbol and other RHS are either 2 variables or a single terminal.

Significance of CNF

- Theorem: There is an algorithm to construct a grammar \( G' \) in CNF that is equivalent to a CFG \( G \).

CFG \( \rightarrow \) CNF: Construction

- Obtain an equivalent grammar that does not contain \( \lambda \)-rules, chain rules, and useless variables.
- Apply following conversion on rules of the form:

  \[
  A \rightarrow bBcC
  \]

  \[
  A \rightarrow PQ \quad P \rightarrow b
  \]

  \[
  Q \rightarrow BR \quad R \rightarrow WC
  \]

  \[
  W \rightarrow c
  \]

Let’s see how this works on the following example grammar:

\[
S \rightarrow \lambda | a | b | aSa | bSb
\]

Converting a general grammar into Chomsky Normal Form works in four steps:
1. Ensure that the start variable doesn’t appear on the right hand side of any rule.
2. Remove all \( \lambda \)-rules productions, except from start variable.
3. Remove unit variable productions of the form \( A \rightarrow B \) where \( A \) and \( B \) are variables.
4. Add variables and dyadic variable rules to replace any longer non-dyadic or non-variable productions.
1. Start Variable
Ensure that start variable doesn’t appear on the right hand side of any rule.

\[ S' \rightarrow S \]
\[ S \rightarrow ? | a | b | aSa | bSb \]

2. Remove \( ? \)-rules
Remove all epsilon productions, except from start variable.

\[ S' \rightarrow S | ? \]
\[ S \rightarrow ? | a | b | aSa | bSb | aa | bb \]

3. Remove variable units
Remove unit variable productions of the form \( A \rightarrow B \).

\[ S' \rightarrow S | ? | a | b | aSa | bSb | aa | bb \]
\[ S \rightarrow ? | a | b | aSa | bSb | aa | bb \]

4. Longer production rules
Add variables and dyadic variable rules to replace any longer productions.

\[ S' \rightarrow ? | a | b | AB | CD | AA | CC \]
\[ S \rightarrow a | b | aSa | bSb | aa | bb | AB | CD | AA | CC \]
\[ A \rightarrow a \]
\[ B \rightarrow SA \]
\[ C \rightarrow b \]
\[ D \rightarrow SC \]

5. Result

\[ S' \rightarrow ? | a | b | AB | CD | AA | CC \]
\[ S \rightarrow a | b | AB | CD | AA | CC \]
\[ A \rightarrow a \]
\[ B \rightarrow SA \]
\[ C \rightarrow b \]
\[ D \rightarrow SC \]
4. Convert all remaining rules into the proper form

\[
S_0 \rightarrow \text{OS1} \\
S_0 \rightarrow \text{A}_1 \text{A}_2 \\
A_1 \rightarrow 0 \\
A_2 \rightarrow \text{SA}_3 \\
A_3 \rightarrow 1 \\
S_0 \rightarrow \text{T#} \\
S_0 \rightarrow TA_4 \\
A_4 \rightarrow \# \\
S_0 \rightarrow 01
\]

Chomsky Normal Form CNF

Exercise

• Write into Chomsky Normal Form the CFG:

\[
\begin{align*}
S & \rightarrow aA | aBB | a \\
A & \rightarrow aaA | a \\
B & \rightarrow bC | bbC \\
C & \rightarrow B
\end{align*}
\]

Chomsky Normal Form CNF

Answer

• Answer (1): First you remove the \(\Rightarrow \) productions (\(A \Rightarrow \)):

\[
\begin{align*}
S & \rightarrow aA | aBB | a \\
A & \rightarrow aaA | a \\
B & \rightarrow bC | bbC \\
C & \rightarrow B
\end{align*}
\]

• Answer (2): Next you remove the unit-productions from:

\[
\begin{align*}
S & \rightarrow aA\text{BB} | a \\
A & \rightarrow aaA | a \\
B & \rightarrow bC | bbC \\
C & \rightarrow B
\end{align*}
\]

• Removing \(C \Rightarrow B\), we have to include the \(C \Rightarrow *B\) possibility, which can be done by substitution and gives:

\[
\begin{align*}
S & \rightarrow aA\text{BB} | a \\
A & \rightarrow aaA | a \\
B & \rightarrow bC | bbC \\
C & \rightarrow bC | bbC
\end{align*}
\]

Chomsky Normal Form CNF

Answer (3): Next, we determine the useless variables in:

\[
\begin{align*}
S & \rightarrow aA\text{BB} | a \\
A & \rightarrow aaA | aa \\
B & \rightarrow bC | bbC \\
C & \rightarrow bC | bbC
\end{align*}
\]

The variables \(B\) and \(C\) can not terminate and are therefore useless. So, removing \(B\) and \(C\) gives:

\[
\begin{align*}
S & \rightarrow aA | a \\
A & \rightarrow aaA | aa
\end{align*}
\]
Answer(4): To make the CFG in Chomsky normal form, we have to introduce terminal producing variables for

\[ S \rightarrow aA | a \]
\[ A \rightarrow aaA | aa, \]

which gives

\[ S \rightarrow X_a a \]
\[ A \rightarrow X_a X_a | X_a X_a \]
\[ X_a \rightarrow a. \]

Answer(5): Finally, we have to ‘chain’ the variables in

\[ S \rightarrow X_a A | a \]
\[ A \rightarrow X_a X_a | X_a X_a \]
\[ X_a \rightarrow a, \]

which gives

\[ S \rightarrow X_a A | a \]
\[ A \rightarrow X_a A_2 | X_a X_a \]
\[ X_a \rightarrow a. \]

Chomsky Normal Form CNF

Griebach Normal Form GNF

• A CFG is in Griebach Normal Form if each rule is of the form

\[ A \rightarrow aA_1 A_2 \ldots A_n \]
\[ A \rightarrow a \]
\[ S \rightarrow \lambda \]

where \( A \in V = \{S\} \)

Griebach Normal Form GNF

• Theorem: There is an algorithm to construct a grammar \( G \) in GNF that is equivalent to a CFG \( G \).

Griebach Normal Form GNF

• A CFG is in Griebach Normal Form if each rule is of the form

\[ A \rightarrow aA_1 A_2 \ldots A_n \]
\[ A \rightarrow a \]
\[ S \rightarrow \lambda \]

where \( A \in V = \{S\} \)

Griebach Normal Form GNF

• The size of the equivalent GNF can be large compared to the original grammar.
  • Next Example CFG has 5 rules, but the corresponding GNF has 24 rules!!

Griebach Normal Form GNF

• Length of the derivation in GNF = Length of the string.

Griebach Normal Form GNF

• GNF is useful in relating CFGs (“generators”) to pushdown automata (“recognizers”/“acceptors”).

Griebach Normal Form GNF

• Theorem: There is an algorithm to construct a grammar \( G \) in GNF that is equivalent to a CFG \( G \).
An even more general form of grammars exists. In general, a non-context free grammar is one in which whole mixed variable/terminal substrings are replaced at a time. For example with $\Sigma = \{a, b, c\}$ consider:

- $S \rightarrow ? | ASBC$
- $A \rightarrow a$
- $CB \rightarrow BC$
- $aB \rightarrow ab$
- $bB \rightarrow bb$
- $bC \rightarrow bc$
- $cC \rightarrow cc$

For technical reasons, when length of LHS always $\leq$ length of RHS, these general grammars are called context sensitive.

Find the language generated by the CSG:

$S \rightarrow ? | ASBC$

$A \rightarrow a$

$CB \rightarrow BC$

$aB \rightarrow ab$

$bB \rightarrow bb$

$bC \rightarrow bc$

$cC \rightarrow cc$

Answer is \{a^n b^n c^n\}. In a future class we’ll see that this language is not context free. Thus perturbing context free-ness by allowing context sensitive productions expands the class.

So far we studied 3 grammars:

1. Regular Grammars (RG)
2. Context Free Grammars (CFG)
3. Context Sensitive Grammars (CSG)

The relation between these 3 grammars is as follow:

Grammar Applications

Programming languages are often defined as Context Free Grammars in Backus-Naur Form (BNF).

Example:

- $<\text{if}\_\text{statement}> ::= \text{IF} <\text{expression}> <\text{then\_clause}> <\text{else\_clause}>$
- $<\text{expression}> ::= <\text{term}> | <\text{expression}> + <\text{term}>$
- $<\text{term}> ::= <\text{factor}> | <\text{term}> * <\text{factor}>$

The variables as indicated by <a variable name>

The arrow $\rightarrow$ is replaces by $::=$

Here, IF, + and * are terminals.

“Syntax Checking” is checking if a program is an element of the CFG of the programming language.
Applications of CFG

Parsing is where we use the theory of CFGs.

The theory is especially relevant when dealing with Extensible Markup Language (XML) files and their corresponding Document Type Definitions (DTDs).

Document Type Definitions define the grammar that the XML files have to adhere to. Validating XML files equals parsing it against the grammar of the DTD.

The nondeterminism of NPDAs can make parsing slow. What about deterministic PDAs?