Parsing

Start with a CFG $G = (V, \Sigma, P, S)$ and a string $w \in \Sigma^*$.

- **Parsing** determines whether $w \in L(G)$.

- **Parsing** constructs, implicitly or explicitly, a derivation tree (parse tree) for $w$.

- **Parsing** may construct a leftmost derivation

  $$S \xrightarrow{*} L w.$$

- Or **parsing** may construct a rightmost derivation

  $$S \xrightarrow{*} R w.$$
Applications

- Compilers and interpreters
- Script processing
- Web browsers
- Search tools
- Natural language processing
- Symbolic computation
Example

\[ G_1 : \quad S \rightarrow AB \mid AaCbB \\
A \rightarrow aA \mid \lambda \\
B \rightarrow bB \mid \lambda \\
C \rightarrow baC \mid ba \]

- Give a derivation tree for \( w = aababbb \)

- Give a leftmost derivation for \( w \)

- What is \( L(G_1) \)?
Existence

**Theorem 4.1.** A string $w$ is in $L(G)$ if and only if there exists a leftmost derivation of $w$.

**Proof sketch:**

- **If:**

  A leftmost derivation of $w$ is a derivation of $w$.

- **Only If:**

  Take a derivation tree $T$ with yield $w$. A preorder traversal of $T$ results in a leftmost derivation of $w$. 
Another Example

Is a leftmost derivation always unique?

For $G_1$, the answer is “YES.”

Consider this CFG that also generates the language $L(G_1)$:

$$
G_2 : \quad S \rightarrow ACB \\
A \rightarrow aA \mid \lambda \\
B \rightarrow bB \mid \lambda \\
C' \rightarrow abC \mid \lambda.
$$
Another Example
(Continued)

Two leftmost derivations for \( w = abb \) in \( G_2 \):

\[
\begin{align*}
S & \rightarrow_L ACB \\
   & \rightarrow_L CB \\
   & \rightarrow_L abCB \\
   & \rightarrow_L abB \\
   & \rightarrow_L abbB \\
   & \rightarrow_L abb
\end{align*}
\]

\[
\begin{align*}
S & \rightarrow_L ACB \\
   & \rightarrow_L aACB \\
   & \rightarrow_L aCB \\
   & \rightarrow_L aB \\
   & \rightarrow_L abB \\
   & \rightarrow_L abbB \\
   & \rightarrow_L abb
\end{align*}
\]

CS 4114 Lecture Notes
Yet Another Example

Consider a third CFG for the same language:

\[ G_3 : \begin{align*}
S &\rightarrow ACB \\
A &\rightarrow AA \mid a \mid \lambda \\
B &\rightarrow bB \mid \lambda \\
C &\rightarrow abC \mid \lambda.
\end{align*} \]

In \( G_3 \), there are an infinite number of leftmost derivations for \( w = abb \).

**Lesson**: \( \lambda \)-rules must be used carefully.
Ambiguity

A grammar is **unambiguous** if every sentence in the grammar has a unique leftmost derivation. A grammar is **ambiguous** if it is not unambiguous.

**EXAMPLE.**

$G_1$ is an unambiguous grammar generating $L(G_1)$. $G_2$ and $G_3$ are ambiguous grammars generating the same language.

**Lesson:** Avoid ambiguous grammars, when possible.
Inherent Ambiguity

A CFL $L$ is **inherently ambiguous** if every CFG that generates $L$ is ambiguous.

**EXAMPLE.** The language

$$L_4 = \{a^i b^j c^k \mid i, k \geq 0 \text{ and } i = j \text{ or } j = k\}$$

is inherently ambiguous. A CFG to generate $L_4$ is

$$G_4 : \quad S \rightarrow AB \mid EF$$

$$A \rightarrow aAb \mid \lambda$$

$$B \rightarrow cB \mid \lambda$$

$$E \rightarrow aE \mid \lambda$$

$$F \rightarrow bFc \mid \lambda.$$ 

**EXERCISE.** Show that there are two derivation trees for $a^ib^ic^i$. 

---

CS 4114 Lecture Notes
Graph of a Grammar

Start with a CFG $G = (V, \Sigma, P, S)$.

**Leftmost Graph of $G$**

Directed graph $g_L(G) = (N_L, A_L)$:

$$N_L = \left\{ w \in (V \cup \Sigma)^* \mid S \xrightarrow{L}^* w \right\}$$

$$A_L = \left\{ (v, w) \mid v, w \in N_L \text{ and } v \xrightarrow{L} w \right\}$$

**Rightmost Graph of $G$**

Directed graph $g_R(G) = (N_R, A_R)$:

$$N_R = \left\{ w \in (V \cup \Sigma)^* \mid w \xrightarrow{R}^* v, \text{ where } v \in \Sigma^* \right\}$$

$$A_R = \left\{ (v, w) \mid v, w \in N_R \text{ and } v \xrightarrow{R} w \right\}$$
Example Graphs

Consider this grammar:

\[ G_1 : \quad S \rightarrow aSbB \mid aA \mid BS \]
\[ A \rightarrow Aa \mid S \mid \lambda \]
\[ B \rightarrow bB \mid S \mid b. \]

Draw the initial part of \( g_L(G_1) \) and \( g_R(G_1) \).
Observations

• The start symbol $S$ is a root for $g_L(G)$ and $g_R(G)$.

• Outdegree is always finite.

• Graphs are infinite if $L(G)$ is infinite. Also, if $G$ has recursion.

• Parsing a string $w \in L(G)$ can be thought of as finding a path from $S$ to $w$ in $g_L(G)$ or from $w$ back to $S$ in $g_R(G)$.

• Any parsing algorithm explores, explicitly or implicitly, some graph of $G$. 
Parsing Exercise

Again consider the grammar:

\[ G_1 : \quad S \rightarrow aSbB \mid aA \mid BS \]
\[ A \rightarrow Aa \mid S \mid \lambda \]
\[ B \rightarrow bB \mid S \mid b. \]

EXERCISE.

1. Using \( g_L(G_1) \), parse \( u = a \) ?

2. Using \( g_R(G_1) \), parse \( v = aabb \) ?

What style of search did you use ?
Further Definitions

Start with a CFG $G = (V, \Sigma, P, S)$ and a string $w \in \Sigma^*$ to be parsed.

- If $uv \in N_L(G)$, $u \in \Sigma^*$, and $v = \lambda$ or $v$ begins with a nonterminal, then $u$ is the terminal prefix of $uv$.

- If $x \in N_L(G)$ and the terminal prefix of $x$ is not a prefix of $w$, then $x$ is a dead end. If $x$ is a sentence other than $w$, then it is also a dead end.
Parsing Directions

- **Top-down:** Start with $S$. Search $g_L(G)$ for $w$. Gives a leftmost derivation.

  **Problem:** Left recursion:

  $A \xrightarrow{L} Au.$

- **Bottom-up:** Start with $w$. Search $g_R(G)$ for $S$. Each step in the search is a **reduction** of a sentential form by some rule. Get a rightmost derivation in reverse.

  **Problem:** Finding the substring to reduce when there is the possibility of unending reductions:

  $A \xrightarrow{*} \lambda$

  or

  $A \xrightarrow{\dagger} A.$
Graph Searching

Search paths in \( g_L(G) \) or \( g_R(G) \) until \( w \) is found or until a dead end is found. Search can be one of:

- **Breadth-first Search:** Implies a queue.

- **Depth-first Search:** Implies a stack.

**Four combinations of algorithms result:**

1. Breadth-first top-down parsing

2. Depth-first top-down parsing

3. Breadth-first bottom-up parsing

4. Depth-first bottom-up parsing
Breadth-First Top-Down Parsing

BF_TD_Parsing(\(G, w\))

\(\triangleright \) \(Q\) is a queue of sentential forms.
\(\text{}\)
\(\text{}\)
\(\text{INSERT}(S, Q)\)

repeat

\[ q \leftarrow \text{REMOVE}(Q) \]

Let \( q = uAv \), where \( u \in \Sigma^* \).

for \( A \rightarrow x \) an \( A\)-rule

\[ q' \leftarrow uxv \]

\[ \text{if} \ q' = w \]

\[ \quad \text{then return successful parse} \]

\[ \text{else} \ p \leftarrow \text{terminal prefix of } q' \]

\[ \text{if} \ p \text{ is a prefix of } w \text{ and } p \neq q' \]

\[ \quad \text{then } \text{INSERT}(q', Q) \]

\[ \text{else} \ \triangleright q' \text{ is a dead end.} \]

\[ \quad \triangleright \text{Discard it.} \]

until \( \text{EMPTY}(Q) \)

return unsuccessful parse
Example Grammar For Additive Expressions

The additive expression grammar is

\[ \text{AE} = \{V, \Sigma, P, S\}, \]

where

\[ V = \{S, A, T\} \]
\[ \Sigma = \{b, +, (, )\} \]

and \( P \) consists of these rules:

1. \( S \rightarrow A \)
2. \( A \rightarrow T \)
3. \( A \rightarrow A + T \)
4. \( T \rightarrow b \)
5. \( T \rightarrow (A) \)
Example of BF_TD_Parsing

Use BF_TD_Parsing to parse the sentence

$b$.

Trace the contents of $Q$.

Draw the graph that illustrates the portion of $g_L(AE)$ that is searched.
Another Example of BF_TD_Parsing

Use BF_TD_Parsing to parse the non-sentence

\[ b++ \]

Trace the contents of \( Q \).

Draw the graph that illustrates the portion of \( g_L(AE) \) that is searched.
Depth-First Top-Down Parsing

Use the recursion stack to implement DFS.

DF_TD_Parsing(z)

Let $z = uAv$, where $u \in \Sigma^*$.

for $A \rightarrow x$ an $A$-rule

do $z' \leftarrow uxv$

if $z' = w$

then return successful parse

else $p \leftarrow$ terminal prefix of $z'$

if $p$ is a prefix of $w$ and $p \neq z'$

then DF_TD_Parsing($z'$)

if successful parse

then return successful parse

else \(\triangleright\) Do nothing.

else \(\triangleright\) $z'$ is a dead end.

\(\triangleright\) Discard it.

return unsuccessful parse

Initial call is DF_TD_Parsing($S$).
Example of DF_TD_Parsing

Use DF_TD_Parsing to parse the sentence

b.

Trace the contents of the recursion stack.

Draw the graph that illustrates the portion of $g_L(AE)$ that is searched.
Observations on Top-Down Parsing

- BFS always successfully parses every sentence. It may not terminate for every non-sentence.

- DFS does not always terminate for sentences or non-sentences.

- Problem is left recursion.

- Queue in BFS may grow exponentially large.

- DFS generally uses less space if there is no infinite path followed.
Bottom-Up Parsing

Let’s try an example with our additive expression grammar:

$$AE = \{V, \Sigma, P, S\},$$

where

$$V = \{S, A, T\}$$

$$\Sigma = \{b, +, (, )\}$$

and $P$ consists of these rules:

1. $S \rightarrow A$
2. $A \rightarrow T$
3. $A \rightarrow A + T$
4. $T \rightarrow b$
5. $T \rightarrow (A)$
Example

Let the string to be parsed be \( w = b + b + (b) \).

- What is the parse tree for \( w \)?

- What is the rightmost derivation for \( w \)?

- What is the sequence of reductions done? Identify the right-hand sides of rules that are reduced.

- Encode the sequence of reductions and rules.
Shift-Reduce Parsing

Think of bottom-up parsing as producing a sequence of reduced strings:

\[ w = x_0, x_1, x_2, \ldots, x_k, \ldots, \]

where

\[ x_k \xrightarrow[R]{r_k} x_{k-1}. \]

Let \( r_k \) be the number of the rule used.

Each unreduced string can be written as

\[ x_{k-1} = u_{k-1}v_{k-1}, \]

where \( v_{k-1} \in \Sigma^* \) and the right-hand side of rule \( r_k \) is a suffix of \( u_{k-1} \).

The right-hand side of rule \( r_k \) is called the handle of the reduction of \( x_{k-1} \) to \( x_k \).

**EXERCISE.**

Apply this terminology to the previous example.
Conceptual Shift-Reduce Parsing

Shift-Reduce_Parsing\((G, w)\)

\(u \leftarrow \lambda\)
\(v \leftarrow w\)

repeat
  choose ▷ Shift one symbol from \(v\) to \(u\).
  \(a \leftarrow \) first symbol of \(v\)
  \(v \leftarrow \) remaining suffix of \(v\)
  \(u \leftarrow u \cdot a\)
  or ▷ Reduce by a rule.
    choose \(A \rightarrow z\), a rule
    if \(u = yz\), for some \(y\)
    then \(u \leftarrow yA\)
    if \(uv = S\)
    then return successful parse
  end choose
end choose
until forever
Breadth-First Bottom-up

BF_BU_Parsing($G, w$)
▷ $Q$ is a queue of $(u, v)$ pairs, where $uv$ is a reduced form of $w$.
$INSERT((\lambda, w), Q)$
repeat
  $(u, v) \leftarrow REMOVE(Q)$
  if $v \neq \lambda$
    then ▷ Shift $(u, v)$.
      $(u', v') \leftarrow SHIFT(u, v)$
      $INSERT((u', v'), Q)$
  for $A \rightarrow z$, a rule
    do if $u = yz$, for some $y$
      then ▷ Reduce by $A \rightarrow z$.
        $INSERT((yA, v), Q)$
      if $yAu = S$
        then return successful parse
  until $EMPTY(Q)$
return unsuccessful parse
Example of BF_BU_Parsing

Use BF_BU_Parsing to parse the sentence

\[ b + (b). \]

Trace the contents of \( Q \).

Draw the graph that illustrates the portion of \( g_R(AE) \) that is searched.
Another Example of \texttt{BF\_BU\_Parsing}

Use \texttt{BF\_BU\_Parsing} to parse the non-sentence \texttt{b++}.

Trace the contents of \texttt{Q}.

Draw the graph that illustrates the portion of \( g_R(AE) \) that is searched.
Depth-First Bottom-up

DF_BU_Parsing(\(u, v\)) \hspace{1em} \triangleright \text{Recursive subroutine}

\textbf{if} \(v \neq \lambda\)
\textbf{then} \hspace{1em} \triangleright \text{Shift} \((u, v)\).
\((u', v') \leftarrow \text{SHIFT}(u, v)\)
DF_BU_Parsing\((u', v')\)
\textbf{if} \text{successful parse}
\textbf{then return} \text{successful parse}

\textbf{for} \(A \rightarrow z\), a rule
\textbf{do if} \(u = yz\), for some \(y\)
\textbf{then} \hspace{1em} \triangleright \text{Reduce by} \(A \rightarrow z\).
\textbf{if} \(yAv = S\)
\textbf{then return} \text{successful parse}
DF_BU_Parsing\((yz, v)\)
\textbf{if} \text{successful parse}
\textbf{then return} \text{successful parse}

\textbf{return} \text{unsuccessful parse}

Initial call is DF_BU_Parsing\((\lambda, w)\).
Example of DF_BU_Parsing

Use DF_BU_Parsing to parse the sentence

\[ b + (b). \]

Trace the contents of the recursion stack.

Draw the graph that illustrates the portion of \( g_R(AE) \) that is searched.
Observations on Bottom-up Parsing

- BFS always successfully parses every sentence. It may not terminate for every non-sentence.

- DFS does not always terminate for sentences or non-sentences.

- Problem occurs if $A \xrightarrow{*} \lambda$ or $A \xrightarrow{+} A$, for some nonterminal $A$. 
Keys to Efficient Parsing

- Good grammars (see Chapter 5 on normal forms)

- Finite lookahead
  - Top-Down: Which rule to apply?
  - Bottom-Up: Shift or reduce? If reduce, which rule?

- Our example grammar AE can be parsed with one-symbol lookahead bottom-up, but not with any bounded amount of lookahead top-down.