Unrestricted Grammars

Chapter 23
Grammars, SD Languages, and Turing Machines

Grammar \( L \) SD Language

SD Language Accepts

Turing Machine
Unrestricted Grammars

An *unrestricted*, or *type 0*, or *phrase structure grammar* \( G \) is a quadruple \((V, \Sigma, R, S)\), where:

- \( V \) is an alphabet,
- \( \Sigma \) (the set of terminals) is a subset of \( V \),
- \( R \) (the set of rules) is a finite subset of \( (V^+ \times V^*) \),
- \( S \) (the start symbol) is an element of \( V - \Sigma \).

The language generated by \( G \) is:

\[ \{ w \in \Sigma^* : S \Rightarrow_G^* w \} . \]
Unrestricted Grammars

Example: $A^nB^nC^n = \{a^n_b^n c^n, \ n \geq 0\}$.

CFG attempt 1:

$$
S \rightarrow A \ B \ C \\
A \rightarrow aA \mid \varepsilon \\
B \rightarrow bB \mid \varepsilon \\
C \rightarrow cC \mid \varepsilon
$$
Unrestricted Grammars

Example: $A^nB^nC^n = \{a^n b^n c^n, \ n \geq 0\}$.

CFG attempt 2:

$$S \rightarrow a \ b \ S \ c \ | \ \varepsilon$$

ababcc
Unrestricted Grammars

Example: $A^nB^nC^n = \{a^nb^nc^n, \ n \geq 0\}$.

$$
\begin{align*}
S & \rightarrow aBSc \\
S & \rightarrow \varepsilon \\
Ba & \rightarrow aB \\
Bc & \rightarrow bc \\
Bb & \rightarrow bb
\end{align*}
$$

Proof:

- Only strings in $A^nB^nC^n$:
- All strings in $A^nB^nC^n$:
Another Example

\[\{w \in \{a, b, c\}^* : \#_a(w) = \#_b(w) = \#_c(w)\}\]
Another Example

\{w \in \{a, b, c\}^* : \#_a(w) = \#_b(w) = \#_c(w)\}

\[
\begin{align*}
S & \rightarrow ABCS \\
S & \rightarrow \varepsilon \\
AB & \rightarrow BA \\
BC & \rightarrow CB \\
AC & \rightarrow CA \\
BA & \rightarrow AB \\
CA & \rightarrow AC \\
CB & \rightarrow BC \\
A & \rightarrow a \\
B & \rightarrow b \\
C & \rightarrow c
\end{align*}
\]
WW = \{ww : w \in \{a, b\}^*\}
WW = \{ww : w \in \{a, b\}^*\}

One idea:

1. Generate a string in \(ww^R\), plus delimiters

   \[aaabb\text{C}bbaaa\#\]

2. Reverse the second half.
WW = \{ww : w \in \{a, b\}\}^*\}

\[
S \rightarrow T# \\
T \rightarrow aTa \\
T \rightarrow bTb \\
T \rightarrow C \\
C \rightarrow CP \\
P_{aa} \rightarrow aPa \\
P_{ab} \rightarrow bPa \\
P_{ba} \rightarrow aPb \\
P_{bb} \rightarrow bPb \\
P_{a#} \rightarrow #a \\
P_{b#} \rightarrow #b \\
C# \rightarrow \varepsilon
\]

/* Generate the wall exactly once. */
/* Generate \(wCw^R\). */

"""""""""""""""""""""

/* Generate a pusher \(P\) */
/* Push one character to the right to get ready to jump. */

"""""""""""""""""""""

/* Hop a character over the wall. */
"""""""""""""""""""""
A Strong Procedural Feel

Unrestricted grammars have a procedural feel that is absent from most restricted grammars.

Derivations often proceed in phases. We make sure that the phases work properly by using nonterminals as flags that we’re in a particular phase.

It’s very common to have three main phases:
- Generate the right number of the various symbols.
- Move them around to get them in the right order.
- Clean up and get rid of nonterminals.

No surprise: unrestricted grammars are general computing devices.
Equivalence of Unrestricted Grammars and Turing Machines

**Theorem:** A language is generated by an unrestricted grammar if and only if it is in SD.

**Proof:**

*Only if (grammar $\rightarrow$ TM):* by construction of an NDTM.

*If (TM $\rightarrow$ grammar):* by construction of a grammar that mimics the behavior of a semideciding TM.
Given $G$, produce a Turing machine $M$ that semidecides $L(G)$.

$M$ will be nondeterministic and will use two tapes:

For each nondeterministic “incarnation”:
- Tape 1 holds the input.
- Tape 2 holds the current state of a proposed derivation.

At each step, $M$ nondeterministically chooses a rule to try to apply and a position on tape 2 to start looking for the left hand side of the rule. Or it chooses to check whether tape 2 equals tape 1. If any such machine succeeds, we accept. Otherwise, we keep looking.
Proof that Turing Machine $\rightarrow$ Grammar

Build $G$ to simulate the forward operation of a TM $M$:

The first (generate) part of $G$:
Create all strings over $\Sigma^*$ of the form:

$$w = \# q q q q 0 0 0 1 1 a a 2 2 a a 3 3 q q q q \#$$

The second (test) part of $G$ simulates the execution of $M$ on a particular string $w$. An example of a partially derived string:

$$\# q q q q 1 1 b b 2 2 c c 4 4 q 0 0 1 1 a a 3 3 \#$$

Examples of rules:

- $q 1 0 0 0 b b a a \rightarrow b 2 2 q 1 0 1 1 a a b b 4 4 q 0 1 1 1 a a 3 3$
The Last Step

The third (cleanup) part of G erases the junk if M ever reaches any of its accepting states, all of which will be encoded as A.

Rules:

\[ \forall x \quad x \ A \rightarrow A \ x \quad /* \text{Sweep A to the left.} \]
\[ \forall x, y \quad \#A \ x \ y \rightarrow x \ \#A \quad /* \text{Erase duplicates.} \]
\[ \#A\# \rightarrow \varepsilon \]
An Unrestricted Grammar of Tabla Drumming

contexts that generate $t\ddot{i}$:

\[
dha V \ dha \rightarrow \ dha \ t\ddot{i} \ dha \\
dha V V \rightarrow \ dha \ t\ddot{i} V \\
\text{etc.}
\]

contexts that generate $dha$:

\[
kt V_1 \ tr \rightarrow \ kt \ dha \ tr \\
kt V_1 V_1 \rightarrow \ kt \ dha \ V_1 \\
\text{etc.}
\]

contexts that generate $-\!-$:

\[
dha V_1 \ tr \rightarrow \ dha \ -\ tr \\
dha V_1 \ dha \rightarrow \ dha \ -\ dha \\
\text{etc.}
\]
We say that $G$ computes $f$ iff:

\[ \forall w, \nu \in \Sigma^* (S w S \Rightarrow_{G}^* \nu \iff \nu = f(w)) \]

Example: $f(x) = \text{succ}(x)$

\[
\begin{align*}
S \perp S & \Rightarrow_{G}^* 11 \\
S \perp 1 S & \Rightarrow_{G}^* 11 \\
S \perp 1 1 S & \Rightarrow_{G}^* 11 1
\end{align*}
\]
Computing with Grammars

We say that $G$ computes $f$ iff:

$$\forall w, \ v \in \Sigma^* \ (SwS \Rightarrow_{G^*} v \iff v = f(w))$$

Example: $f(x) = succ(x)$

<table>
<thead>
<tr>
<th>Rule</th>
<th>Derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow 1S$</td>
<td>$S \rightarrow 1$</td>
</tr>
<tr>
<td>$SS \rightarrow 1$</td>
<td>$SS \rightarrow 1$</td>
</tr>
</tbody>
</table>

Rules: $S \rightarrow 1S$ or $1S \rightarrow 11$
A function $f$ is called *grammatically computable* iff there is a grammar $G$ that computes it.

**Theorem:** A function $f$ is computable iff it is grammatically computable.

In other words, iff a Turing machine can do it, so can a grammar.

**Proof:** uses two constructions, analogous to the ones used to prove that a language can be defined with an unrestricted grammar iff it is in SD.
The $value_n$ Functions

For any $k$, $value_k(n)$ returns the nonnegative integer that is encoded, base $k$, by the string $n$.

Example:

$$value_2(101) = 5.$$
Example of Computing with a Grammar

\[ f(n) = m, \text{ where } value_1(m) = 2 \cdot value_1(n). \]

\[ G = (\{S, 1\}, \{1\}, R, S), \text{ where } R = \]

Examples:

Input: \[ S111S \quad SS \]

Output:
Example of Computing with a Grammar

\[ f(n) = m, \text{ where } value_1(m) = 2 \cdot value_1(n). \]

\[ G = (\{S, 1\}, \{1\}, R, S), \text{ where } R = \]

Examples:

Input: \( S111S \) \( SS \)

Output: \( 111111 \) \( \epsilon \)
Example of Computing with a Grammar

\[ f(n) = m, \text{ where } \text{value}_1(m) = 2 \cdot \text{value}_1(n). \]

\[ G = (\{S, 1\}, \{1\}, R, S), \text{ where } R = \]

\[ S_1 \rightarrow 11S \]
\[ SS \rightarrow \epsilon \]

Examples:

Input: \[ S111S \quad SS \]

Output: \[ 111111 \quad \epsilon \]
Another Example

\[ f(x) = x \] with extra blanks squeezed out.

Examples:

Input: \( S\text{-}a\text{-}b\text{-}a\text{-}a\text{-}b\text{-}b\text{-}S \)

Output: \( a\text{-}a\text{-}b\text{-}a\text{-}a\text{-}b \)

Input: \( S\text{-}a\text{-}a\text{-}b\text{S} \)

Output: \( a\text{b} \)
Another Example

\( f(x) = x \) with extra blanks squeezed out.

\[ G = (\{S, T, a, b, \Box\}, \{a, b, \Box\}, R, S), \text{ where } R = \]

\[
SS \rightarrow \varepsilon \quad /* \text{In case } x \text{ has no nonblank characters.} \\
S\Box \rightarrow S \quad /* \text{Get rid of leading } \Box \text{'s.} \\
Sa \rightarrow aT \quad /* \text{All characters to the left of } T \text{ will be correct.} \\
b \rightarrow bT \\
Ta \rightarrow aT \quad /* \text{Sweep } T \text{ across } a \text{'s and } b \text{'s.} \\
b \rightarrow bT \\
\Box\Box \rightarrow \Box \Box \quad /* \text{Squeeze repeated } \Box \text{'s.} \\
\Box a \rightarrow \Box a T \quad /* \text{Once there is a single } \Box, \text{ sweep } T \text{ past it and the first letter} \\
\Box b \rightarrow \Box b T \quad /* \text{After it.} \\
\Box S \rightarrow \varepsilon \quad /* \text{The } \Box\Box \text{ rule will get rid of all but possibly one } \Box \text{ at the end} \\
\text{of the string.} \\
TS \rightarrow \varepsilon \quad /* \text{If there were no trailing } \Box \text{'s, this rule finishes up.} \]
Decision Problems for Unrestricted Grammars

- Given a grammar $G$ and a string $w$, is $w \in L(G)$?
- Given a grammar $G$, is $\varepsilon \in L(G)$?
- Given two grammars $G_1$ and $G_2$, is $L(G_1) = L(G_2)$?
- Given a grammar $G$, is $L(G) = \emptyset$?

Or, as languages:

- $L_a = \{<G, w> : w \in L(G)\}$.
- $L_b = \{<G> : \varepsilon \in L(G)\}$.
- $L_c = \{<G_1, G_2> : L(G_1) = L(G_2)\}$.
- $L_d = \{<G> : L(G) = \emptyset\}$.

None of these questions is decidable.
Proof: Let \( R \) be a mapping reduction from:

\[
A = \{<M, w>: \text{Turing machine } M \text{ accepts } w\} \text{ to } L_a:\
\]

\[
R(<M, w>) =
\]

1. From \( M \), construct the description \(<G#>\) of a grammar \( G#\) such that \( L(G#) = L(M)\).
2. Return \(<G#>, w>\).

If \( Oracle \) decides \( L_a \), then \( C = Oracle(R(<M, w>)) \) decides \( A \). We have already defined an algorithm that implements \( R \). \( C \) is correct:

- If \(<M, w> \in A: M(w) \text{ halts and accepts. } w \in L(M). \text{ So } w \in L(G#). Oracle(<G#, w>) \text{ accepts.}\)
- If \(<M, w> \notin A: M(w) \text{ does not accept. } w \notin L(M). \text{ So } w \notin L(G#). Oracle(<G#, w>) \text{ rejects.}\)

But no machine to decide \( A \) can exist, so neither does \( Oracle \).
The Word Problem

Given two strings, $w$ and $v$, and a rewrite system $T$, determine whether $v$ can be derived from $w$ using $T$.

A key application: Logical reasoning
Semi-Thue Systems

A semi-Thue system $T$ is a pair $(\Sigma, R)$, where:

- $\Sigma$ is an alphabet,
- $R$ (the set of rules) is a subset of $\Sigma^+ \times \Sigma^*$.

Note that:
- There is no unique start symbol.
- There is no distinction between terminal and nonterminal symbols.

**Theorem:** The word problem for semi-Thue systems is undecidable.