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
  $$(V^* \setminus (V \cdot \Sigma) \setminus V^*) \times V^*.$$
- $S$ (the start symbol) is an element of $V - \Sigma$.

We define derivations just as we did for context-free grammars.

The language generated by $G$ is

$$\{w \in \Sigma^* : S \Rightarrow^* G w\}$$

There is no notion of a derivation tree or rightmost/leftmost derivation for unrestricted grammars.

Unrestricted Grammars

Example: $L = a^n b^n c^n$, $n > 0$
- $S \rightarrow aBSc$
- $S \rightarrow aBc$
- $Ba \rightarrow aB$
- $Bc \rightarrow bc$
- $Bb \rightarrow bb$

Another Example

$L = \{w \in \{a, b, c\}^* : \text{number of a's, b's and c's is the same}\}$
- $S \rightarrow ABCS$
- $S \rightarrow ABC$
- $AB \rightarrow BA$
- $BC \rightarrow CB$
- $AC \rightarrow CA$
- $BA \rightarrow AB$
- $CA \rightarrow AC$
- $CB \rightarrow BC$
- $A \rightarrow a$
- $B \rightarrow b$
- $C \rightarrow c$
A Strong Procedural Feel

Unrestricted grammars have a procedural feel that is absent from 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 two main phases:
- Generate the right number of the various symbols.
- Move them around to get them in the right order.

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 recursively enumerable (i.e., it is semidecided by some Turing machine M).

**Proof:**
- Only if (grammar → TM): by construction of a nondeterministic Turing machine.
- If (TM → grammar): by construction of a grammar that mimics backward computations of M.

**Proof that Grammar → Turing Machine**

Given a grammar 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 → Grammar

Suppose that M semidecides a language L (it halts when fed strings in L and loops otherwise). Then we can build M' that halts in the configuration (h, 0).

We will define G so that it simulates M' backwards.
We will represent the configuration (q, 0aw) as
>uaqw<

M'
goesto

Then, if w ∈ L, we require that our grammar produce a derivation of the form
S →_G_ >h< (produces final state of M')
⇒_G_* >abq< (some intermediate state of M')
⇒_G_* >sw< (the initial state of M')
⇒_G_ w< (via a special rule to clean up >s)
⇒_G_ w (via a special rule to clean up <)

The Rules of G

S → >h< (the halting configuration)
> → ε (clean-up rules to be applied at the end)
< → ε

Rules that correspond to δ:

If δ(q, a) = (p, b) :
bp → aq

If δ(q, a) = (p, −) :
abp → aqb ∀b ∈ Σ
aqp< → aq<

If δ(q, a) = (p, ←), a ≠ 0
pa → aq

If δ(q, 0) = (p, ←)
pqb → qpb ∀b ∈ Σ
p< → q<
A REALLY Simple Example

\[ M' = (K, \{a\}, \delta, s, \{h\}) \]

\[
\delta = (\{(s, \Box), (q, \rightarrow)\}, 1 \\
(\{(q, a), (q, \rightarrow)\}, 2 \\
(\{(q, \Box), (t, \leftarrow)\}, 3 \\
(\{(t, a), (p, \Box)\}, 4 \\
(\{(t, \Box), (h, \Box)\}, 5 \\
(\{(p, \Box), (t, \leftarrow)\}) 6 \\
\]

\[ L = a^* \]

\[
S \rightarrow >\Box h < \\
>\Box s \rightarrow \varepsilon \\
< \rightarrow \varepsilon \\
\]

(1) \[ \Box q \rightarrow \Box s\Box \] \\
[ \Box aq \rightarrow \Box sa \] \\
[ \Box q< \rightarrow >\Box s< \]

(2) \[ a\Box q \rightarrow aq\Box \] \\
[ aaq \rightarrow aqa \] \\
[ a\Box q< \rightarrow aq< \]

Working It Out

\[
S \rightarrow >\Box h < \\
>\Box s \rightarrow \varepsilon \\
< \rightarrow \varepsilon \\
\]

(1) \[ \Box q \rightarrow \Box s\Box \] \\
[ \Box aq \rightarrow \Box sa \] \\
[ \Box q< \rightarrow >\Box s< \]

(2) \[ a\Box q \rightarrow aq\Box \] \\
[ aaq \rightarrow aqa \] \\
[ a\Box q< \rightarrow aq< \]

<table>
<thead>
<tr>
<th>Production</th>
<th>Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>S \rightarrow &gt;\Box h &lt;</td>
<td>1</td>
</tr>
<tr>
<td>&gt;\Box s \rightarrow \varepsilon</td>
<td>2</td>
</tr>
<tr>
<td>&lt; \rightarrow \varepsilon</td>
<td>3</td>
</tr>
<tr>
<td>(1) \Box q \rightarrow \Box s\Box</td>
<td>4</td>
</tr>
<tr>
<td>\Box aq \rightarrow \Box sa</td>
<td>5</td>
</tr>
<tr>
<td>\Box q&lt; \rightarrow &gt;\Box s&lt;</td>
<td>6</td>
</tr>
<tr>
<td>(2) a\Box q \rightarrow aq\Box</td>
<td>7</td>
</tr>
<tr>
<td>aaq \rightarrow aqa</td>
<td>8</td>
</tr>
<tr>
<td>a\Box q&lt; \rightarrow aq&lt;</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Production</th>
<th>Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;\Box saa&lt;</td>
<td>1</td>
</tr>
<tr>
<td>&gt;\Box aqa&lt;</td>
<td>2</td>
</tr>
<tr>
<td>&gt;\Box aaq&lt;</td>
<td>2</td>
</tr>
<tr>
<td>&gt;\Box aaq&lt;</td>
<td>2</td>
</tr>
<tr>
<td>&gt;\Box aat&lt;</td>
<td>4</td>
</tr>
<tr>
<td>&gt;\Box a\Box q&lt;</td>
<td>5</td>
</tr>
<tr>
<td>&gt;\Box q&lt;</td>
<td>6</td>
</tr>
<tr>
<td>&gt;\Box t&lt;</td>
<td>5</td>
</tr>
<tr>
<td>&gt;\Box h&lt;</td>
<td>5</td>
</tr>
</tbody>
</table>

Lecture Notes 25 Grammars and Turing Machines 4
An Alternative Proof

An alternative is to build a grammar $G$ that simulates the forward operation of a Turing machine $M$. It uses alternating symbols to represent two interleaved tapes. One tape remembers the starting string, the other “working” tape simulates the run of the machine.

The first (generate) part of $G$:
Creates all strings over $\Sigma^*$ of the form
\[ w = \text{# } \text{# } Q_s a_1 a_1 a_2 a_2 a_3 a_3 \ldots \]

The second (test) part of $G$ simulates the execution of $M$ on a particular string $w$. An example of a partially derived string:
\[ \text{# } \text{# } Q_s a_1 b_2 c b_4 Q_3 a_3 \]

Examples of rules:
- $b b Q_4 \rightarrow b 4 Q_4$ (rewrite $b$ as $4$)
- $b 4 Q_3 \rightarrow Q_3 b 4$ (move left)

The third (cleanup) part of $G$ erases the junk if $M$ ever reaches $h$.

Example rule:
- $\# h a_1 \rightarrow a_1 \# h$ (sweep $\# h$ to the right erasing the working “tape”)

Computing with Grammars

We say that $G$ computes $f$ if, for all $w, v \in \Sigma^*$,
\[ SwS \Rightarrow_{G^*} v \iff v = f(w) \]
Example:
- $S1S \Rightarrow_{G^*} 11$
- $S11S \Rightarrow_{G^*} 111$ \hspace{1cm} $f(x) = \text{succ}(x)$

A function $f$ is called grammatically computable iff there is a grammar $G$ that computes it.

**Theorem:** A function $f$ is recursive iff it is grammatically computable.
In other words, if a Turing machine can do it, so can a grammar.

**Example of Computing with a Grammar**

$f(x) = 2x$, where $x$ is an integer represented in unary

$G = (\{S, 1\}, \{1\}, R, S)$, where $R =$
- $S1 \rightarrow 11S$
- $SS \rightarrow \varepsilon$

Example:

Input: $S111S$

Output:
More on Functions: Why Have We Been Using Recursive as a Synonym for Computable?

Primitive Recursive Functions

Define a set of basic functions:
- \( \text{zero}_k(n_1, n_2, \ldots n_k) = 0 \)
- \( \text{id}_{j,k}(n_1, n_2, \ldots n_k) = n_j \)
- \( \text{successor}(n) = n + 1 \)

Combining functions:
- Composition of \( g \) with \( h_1, h_2, \ldots h_k \) is
  \[ g(h_1(\quad), h_2(\quad), \ldots h_k(\quad)) \]
- Primitive recursion of \( f \) in terms of \( g \) and \( h \):
  \[ f(n_1, n_2, \ldots n_k, 0) = g(n_1, n_2, \ldots n_k) \]
  \[ f(n_1, n_2, \ldots n_k, m+1) = h(n_1, n_2, \ldots n_k, m, f(n_1, n_2, \ldots n_k, m)) \]

Example:
- \( \text{plus}(n, 0) = n \)
- \( \text{plus}(n, m+1) = \text{succ(plus}(n, m)) \)

Primitive Recursive Functions and Computability

Trivially true: all primitive recursive functions are Turing computable.

What about the other way: Not all Turing computable functions are primitive recursive.

Proof:
Lexicographically enumerate the unary primitive recursive functions, \( f_0, f_1, f_2, f_3, \ldots \).
Define \( g(n) = f_n(n) + 1 \).
\( G \) is clearly computable, but it is not on the list. Suppose it were \( f_m \) for some \( m \). Then
\( f_m(m) = f_m(m) + 1 \), which is absurd.

Suppose \( g \) is \( f_3 \). Then \( g(3) = 27 + 1 = 28 \). Contradiction.

Functions that Aren’t Primitive Recursive

Example: Ackermann’s function:
- \( A(0, y) = y + 1 \)
- \( A(x + 1, 0) = A(x, 1) \)
- \( A(x + 1, y + 1) = A(x, A(x + 1, y)) \)

Thus writing digits at the speed of light on all protons and neutrons in the universe (all lined up) starting at the big bang would have produced \( 10^{127} \) digits.
Recursive Functions

A function is \textbf{\(\mu\)-recursive} if it can be obtained from the basic functions using the operations of:
- Composition,
- Recursive definition, and
- Minimalization of minimalizable functions:

The \textbf{minimalization} of \(g\) (of \(k + 1\) arguments) is a function \(f\) of \(k\) arguments defined as:
\[
f(n_1,n_2,\ldots,n_k) = \begin{cases} 
\text{the least } m \text{ such at } g(n_1,n_2,\ldots,n_k,m)=1, & \text{if such an } m \text{ exists,} \\
0 & \text{otherwise}
\end{cases}
\]

A function \(g\) is \textbf{minimalizable} iff for every \(n_1,n_2,\ldots,n_k\), there is an \(m\) such that \(g(n_1,n_2,\ldots,n_k,m)=1\).

\textbf{Theorem}: A function is \(\mu\)-recursive iff it is recursive (i.e., computable by a Turing machine).

Partial Recursive Functions

Consider the following function \(f\):
\[
f(n) = \begin{cases} 
1 & \text{if } \text{TM}(n) \text{ halts on a blank tape} \\
0 & \text{otherwise}
\end{cases}
\]

The domain of \(f\) is the natural numbers. Is \(f\) recursive?

\textbf{Theorem}: There are uncountably many partially recursive functions (but only countably many Turing machines).

Functions and Machines

![Diagram of Functions and Machines](image-url)
Is There Anything In Between CFGs and Unrestricted Grammars?

Answer: yes, various things have been proposed.

**Context-Sensitive Grammars and Languages:**

A grammar $G$ is context sensitive if all productions are of the form

$$x \rightarrow y$$

and $|x| \leq |y|$.

In other words, there are no length-reducing rules.

A language is context sensitive if there exists a context-sensitive grammar for it.

Examples:

- $L = \{a^n b^n c^n, n > 0\}$
- $L = \{w \in \{a, b, c\}^*: \text{number of } a's, b's \text{ and } c's \text{ is the same}\}$
Context-Sensitive Languages are Recursive

The basic idea: To decide if a string w is in L, start generating strings systematically, shortest first. If you generate w, accept. If you get to strings that are longer than w, reject.

Linear Bounded Automata

A linear bounded automaton is a nondeterministic Turing machine the length of whose tape is bounded by some fixed constant k times the length of the input.

Example: \[ L = \{a^n b^n c^n : n \geq 0\} \]

\[
\begin{array}{l}
\emptyset \xrightarrow{aabbcc} \\
\Rightarrow a' \xrightarrow{a} a' \xrightarrow{b} b' \xrightarrow{c} c' \xrightarrow{L_2}
\end{array}
\]

Context-Sensitive Languages and Linear Bounded Automata

Theorem: The set of context-sensitive languages is exactly the set of languages that can be accepted by linear bounded automata.

Proof: (sketch) We can construct a linear-bound automaton B for any context-sensitive language L defined by some grammar G. We build a machine B with a two track tape. On input w, B keeps w on the first tape. On the second tape, it nondeterministically constructs all derivations of G. The key is that as soon as any derivation becomes longer than \(|w|\) we stop, since we know it can never get any shorter and thus match w. There is also a proof that from any lba we can construct a context-sensitive grammar, analogous to the one we used for Turing machines and unrestricted grammars.

Theorem: There exist recursive languages that are not context sensitive.
Languages and Machines

- Recursively Enumerable Languages
- Recursive Languages
- Context-Sensitive Languages
- Context-Free Languages
- Deterministic Context-Free Languages
- Regular Languages
  - FSMs
  - DPDAs
  - NDPDAs
- Linear Bounded Automata
- Turing Machines
The Chomsky Hierarchy

Type 0 | Type 1 | Type 2

Regular (Type 3) Languages
FSMs
PDAs
Linear Bounded Automata
Turing Machines

Recursively Enumerable Languages
Context-Sensitive Languages
Context-Free Languages
Regular Languages
FSMs
PDAs
Linear Bounded Automata
Turing Machines