Context-sensitive grammars: rules must be of the form

$$\alpha X \beta \rightarrow \alpha \gamma \beta$$

where \( X \) is a variable; RHS is at least as long as LHS, which makes it possible to check whether a given word belongs to the language...

Unrestricted grammar: Rules of the form

$$\alpha \rightarrow \beta$$

\( \alpha \) and \( \beta \) can be any string of symbols.

...Problem: given a word, how to check that it is derivable? There is no obvious limit on how long the intermediate words in the derivation may be. Unrestricted grammars are equivalent to Turing machines, let’s look at TMs first.

---

**Definition**

A TM has:
- input alphabet \( I \) (e.g. \( \{0, 1\} \))
- “blank symbol” not in \( I \), usually denoted \( \text{b} \).
- tape alphabet \( A \) containing \( I \) and \( \text{b} \), possibly also extra symbols such as \( \# \), \$ or \%
- finite set \( Q \) of states, where \( Q \) includes a subset \( T \) of accepting states
- Transition function \( Q \times A \rightarrow Q \times A \times \{L, R\} \)

---

**Turing Machines (TMs)**

equivalent to unrestricted grammars (in terms of expressive power)
Like a PDA, a TM has access to an infinite memory. But TM’s memory is a “tape” not a stack, and TM is free to scan the tape taking actions determined by its state and tape symbol(s) being scanned.
Not just a class of language acceptors, TMs are a standard representation of “effectively computable function”
You have to look rather hard to find a language not accepted by some TM.

---

**Initial configuration of Turing Machine**

INPUT in initial state \( i \)
TM’s computation ends if it encounters a combination of alphabet symbol and state for which no transition is defined, or if it tries to go off left-hand end of tape. When TM’s computation ends we say it halts. An input is accepted if a TM halts in an accepting state.

Some comments
Unlike DFAs or PDAs, no guarantee that TM will halt. It may “end up” stuck at a particular square, or endlessly moving right, reading blank symbols. Deemed to reject.

If a TM halts on some given input, we can regard the contents of the tape at the end of the computation as being an output. So a TM can be viewed as computing a function.

TMs can simulate PDAs (We’ll see how later on)

---

**Example: Palindromes over binary alphabet**

**States:** \{ \(i, p_0, p_1, q_0, q_1, r, t\) \}
- \(i\) – initial state
- \(p_0\) – look for 0 at RHS
- \(p_1\) – look for 1 at RHS
- \(q_0\) – found RHS; check it’s 0
- \(q_1\) – found RHS; check it’s 1
- \(r\) – return to beginning
- \(t\) – accepting

---

**Transitions**

\[
\begin{align*}
\delta(i, b) &= (t, b, R) \quad \text{accept if tape is blank}\ \\
\delta(i, 0) &= (p_0, b, R) \quad \text{delete 0; look for 0 at RHS} \\
\delta(i, 1) &= (p_1, b, R) \quad \text{delete 1; look for 0 at RHS} \\
\delta(p_0, 0) &= (p_0, 0, R) \quad \text{move right to RHS} \\
\delta(p_0, 1) &= (p_0, 1, R) \quad \text{“”””} \\
\delta(p_1, 0) &= (p_1, 0, R) \quad \text{“”””} \\
\delta(p_1, 1) &= (p_1, 1, R) \quad \text{“”””} \\
\delta(p_0, b) &= (q_0, b, L) \quad \text{found RHS; now check} \\
\delta(p_1, b) &= (q_1, b, L) \quad \text{whether 0 or 1} \\
\delta(q_0, 0) &= (r, b, L) \quad \text{check RHS is 0; delete it} \\
\delta(q_1, 1) &= (r, b, L) \quad \text{check RHS is 1 and delete}
\end{align*}
\]

---

**Transitions**

\[
\begin{align*}
\delta(q_0, b) &= (t, b, R) \quad \text{accept if all tape is blank} \\
\delta(q_1, b) &= (t, b, R) \quad \text{blank} \\
\delta(r, 0) &= (r, 0, L) \quad \text{return to LHS} \\
\delta(r, 1) &= (r, 1, L) \quad \text{“”””} \\
\delta(r, b) &= (i, b, R) \quad \text{found LHS; goto state 1}
\end{align*}
\]
Example

Find a TM that accepts words \(w^2w\) where \(w \in \{0, 1\}^*\)

\text{e.g.} \ 01201, 110021100

The language is not a CFL.

Input alphabet \(\{0, 1, 2\}\); we will use tape alphabet \(\{0, 1, 2, \#, \bar{1}\}\)

States:

\begin{itemize}
  \item \(i\) initial state
  \item \(t\) accepting state
  \item \(s_0, s'_0\) look for a corresp. 0
  \item \(s_1, s'_1\) look for a corresp. 1
  \item \(s_2\) check there are no more 0's and 1's to right
  \item \(r, r'\) look for leftmost 0/1 in first half of input
\end{itemize}

Design a TM that accepts

\(\{1, 11, 1101, 1101001, 11010010001, \ldots\}\)

General idea:

Let \(P_i\) be smallest prefix of input containing \(i\) 1's.

TM will check that \(P_i\) is good. Then move to right of \(P_i\).

If \(\bar{1}\), accept.

If 1, for \(i > 1\), reject.

If 0, check \(P_{i+1}\)

To check \(P_{i+1}\):

Check that next block of 0's is:

\begin{itemize}
  \item followed by 1
  \item contains one more 0 than previous block.
\end{itemize}

Transitions

\[
\begin{align*}
\delta(i, 0) &= (s_0, \#, R) \\
\delta(i, 1) &= (s_1, \#, R) \\
\delta(i, \bar{1}) &= (t, \#, R) \\
\delta(i, 2) &= (s_2, \#, R) \\
\delta(s_0, 0) &= (s_0, 0, R) \\
\delta(s_1, 0) &= (s_1, 0, R) \\
\delta(s_2, 0) &= (s_0, 2, R) \\
\delta(s_1, 1) &= (s_1, 1, R) \\
\delta(s_2, 1) &= (s_1, 2, R) \\
\delta(s_0', \#) &= (s_0', \#, R) \\
\delta&s_1', \#) &= (s_1', \#, R) \\
\delta(s_0', 0) &= (r, \#, L) \\
\delta(s_1', 0) &= (r, \#, R) \\
\delta(s_2', 0) &= (s_2, \#, R) \\
\delta(r, \#) &= (s_2, \#, R) \\
\delta(r, 2) &= (r, \#) = (r, \#, L) \\
\delta(r', 2, L) &= (r', 2, L) \\
\delta(r', 0, \#) &= (r', 0, L) \\
\delta(r', 0, \#) &= (r', 0, L) \\
\delta(r', 1) &= (r', 1, L) \\
\delta(r', \#) &= (r', \#, L) \\
\delta(r', \#) &= (i, \#, R)
\end{align*}
\]

Input alphabet \(\{0, 1\}\)

Tape alphabet \(\{0, 1, \#, \bar{1}\}\)

States: \(i\): initial state; \(i':\) second state

\(p_0\): just moved to right of prefix \(P_i\)

\(q_0\): found \(0\) to right of \(P_i\); replace with \# and check next block of 0's has equal length to previous

\(q_1, q'_1\): move to left, turn two 0's into #'

\(r_0, r_1\): move to right to repeat process of turning two 0's into #'

\(q_2, q'_2\): move to left, check we have no more 0's in either block (i.e. all 0's replaced with #')

\(s_0, s_1, s_2\): verified a new block of 0's (and replaced them with #') – now replace them (left to right) with 0's.

\(t\): accepting state
Transitions
\[ \delta(i, 1) = (i', 1, R) \quad i: \text{ initial state} \quad \delta(i', \bar{b}) = (\bar{t}, \bar{b}, R) \]
\[ \delta(p_0, \bar{b}) = (t, \bar{b}, R) \quad p_0: \text{ just moved right of } P_i \]
\[ \delta(p_0, 0) = (q_0, \#, R) \quad \delta(q_1, \#) = (q_1, \#, L) \]
\[ \delta(q_0, 0) = (q_1, \#, L) \quad \delta(q_1, 1) = (q_1', 1, L) \]
\[ \delta(q_0, \#) = (q_0, \#, R) \quad \delta(q_1', \#) = (q_1', \#, L) \]
\[ \delta(r_0, 1) = (q_0, 1, R) \quad \text{next, turn 2 more } 0\text{'s into } \#\text{'s} \]
\[ \delta(q_2, \#) = (q_2, \#, L) \]
\[ \delta(q_2, 1) = (q_2', 1, L) \]
\[ \delta(q_2', \#) = (q_2', \#, L) \]
\[ \delta(q_2', 1) = (s_0, 1, R) \]
\[ s_0, s_1, s_2: \text{ replace } \# \text{ with } 0 \]
\[ \delta(s_0, \#) = (s_0, 0, R) \quad \delta(s_1, 1) = (s_2, 1, R) \]
\[ \delta(s_0, 1) = (s_1, 1, R) \quad \delta(s_2, \#) = (s_2, 0, R) \]
\[ \delta(s_1, \#) = (s_1, 0, R) \quad \delta(s_2, 1) = (r_0, 1, R) \]

An accepting computation
exercise: write down accepting computation given input 1101001

Move to right, looking for another 0 in rightmost block being checked so far: \[ \delta(r_0, 1) = (r_1, 1, R) \quad \delta(r_1, \#) = (r_1, \#, R) \]
where the last one of the above 3 deals with finding a 1 when there are no zeroes left. When we find a 0 in rightmost block, get ready to look for another 0 in previous block as follows:
\[ \delta(r_1, 0) = (d, 0, R) \]
\[ \delta(d, \alpha) = (q_0, \alpha, L) \quad \text{for all symbols } \alpha \]
Return to left of place where \#'s are located, prior to replacing them with 0's:
\[ \delta(x_0, \#) = (x_0, \#, L) \]
\[ \delta(x_0, 1) = (x_1, 1, L) \]
\[ \delta(x_1, \#) = (x_1, \#, L) \]
\[ \delta(x_1, 1) = (s_0, 1, R) \]

Languages accepted by TMs

Languages accepted by TMs are called recursively enumerable languages.
Other models of computation characterise the r.e. languages (e.g. Markov algorithms, random access machines); also infinite memory models in conjunctions with various programming languages.
Also equivalent are non-deterministic TMs, and other variants of TM. So definition is very "robust".
To show acceptability by a TM, it is sufficient to show acceptability by any of these other models.
Languages accepted by TMs that are guaranteed to halt are called recursive languages. (not quite the same!)
More on Recursively Enumerable

Languages

- "recursive" refers to existence of a computational procedure...
- "enumerable" ...which generates members of a language (i.e. enumerates the words)
- very large class of languages (try to think of a language that you cannot define a TM for)

Church-Turing Thesis ("Church’s hypothesis" in Hopcroft and Ullman’s textbook) says that TMs represent the class of all functions which can be computed.

An alternative TM definition

"Extended TM" has transition function

\[ \delta : Q \times A \rightarrow Q \times A \times \{L, R, S\} \]

where \( S \) means stay at same tape square.

Claim: extended TMs accept the same languages as TMs. Anything accepted by a TM is accepted by an extended TM (since any standard TM is an extended TM). Need to show that any language accepted by an extended TM is accepted a standard TM.

Another (more interesting) variation

A TM with doubly infinite tape is one whose tape is infinite in both directions:

A TM in initial state \( i \) with input \( \cdots b b b 0 1 0 0 1 1 1 0 1 b b b b b b b b b b \cdots \)

Suppose language \( L \) is accepted by extended TM \( T \).
Given any transition in \( T \) \( \delta(q_1, a_1) = (q_2, a_2, S) \)
Replace it with \( \delta(q_1, a_1) = (q_{\text{new}}, a_2, R) \) and
for all \( a \in A \): \( \delta(q_{\text{new}}, a) = (q_2, a, L) \)
The new state \( q_{\text{new}} \) appears in all new transitions introduced above, but not in any other.
Do the same thing for all transitions with an \( S \). Resulting TM accepts the same language.

Claim: TMs with doubly infinite tape also accept the r.e. languages.
Converting “standard” TM into equivalent doubly infinite tape TM:
Use two new states that initially place a new character to left of input, then halt if that character is ever encountered by new machine.
New initial state \( i' \) has transitions
\[
\delta(i', 0) = (i'_0, \triangle, R) \quad \delta(i'_0, 0) = (i'_0, 0, R) \quad \delta(i'_1, 0) = (i'_0, 1, R)
\]
\[
\delta(i'_1, 1) = (i'_1, \triangle, R) \quad \delta(i'_0, 0) = (i'_0, 0, R) \quad \delta(i'_1, 1) = (i'_1, 1, R)
\]
Then, in states \( i'_0 \) and \( i'_1 \), on encountering \( b \), replace \( b \) with either 0 or 1, enter a new state that moves to left, finds \( \triangle \), move right and enter state 1.

Converting doubly infinite tape TM into standard TM:
If \( A \) is original tape alphabet, use new alphabet \( A \times A \). Treat the tape as doubly-infinite tape that is folded at the origin.
New blank symbol is \((\text{b}, \text{b})\).
Number of states must be doubled to keep track of which side of the doubly-infinite tape the original machine is on.

### TMs and grammars

Let \( M \) be a TM. Convert \( M \) into an unrestricted grammar \( G \) so that \( M \) accepts word \( w \) iff \( G \) can generate \( w \).
The above transformation would show that unrestricted grammars have at least as much expressive power as TMs.
\( M \) has input alphabet \( I \); let \( I \) be the alphabet of \( G \).
New symbol \( S \) as starting symbol of \( G \).

Start with a grammar that generates words of the form \( wDM_1wB \) where \( D \) and \( M_1 \) are variable symbols.
The following is a modification of the unrestricted grammar for generating words of the form \( ww \) for \( w \in \{a, b\}^* \). (we’ll use 0 and 1 instead of \( a \) and \( b \).)
\[
S \rightarrow XDMYT
\]
\[
T \rightarrow R
\]
\[
YR \rightarrow RY
\]
\[
0R \rightarrow R0
\]
\[
1R \rightarrow R1
\]
\[
XR \rightarrow 0XA \mid 1XB
\]
Further rules of $G$ will simulate TM computation.
$G$ gets variable $M_i$ for each state $i$ of $M$
For each symbol of the tape alphabet $A$, give $G$ a variable that represents that symbol. For $x \in A$ let $V_x$ be a variable that represents $x$.
$B$ denotes a blank tape symbol
$B$ denotes an infinite sequence of blanks.
Rules of $G$: a string derivable from $S$ should represent a state of the TM computation, i.e. a tape with a sequence of symbols and the TM located somewhere on that tape.

Example: suppose that $M$ got input $1110$ and reached a point later on with a tape $10001B1bb\ldots$ where $M$ is located at the 3rd tape square in state $q_2$, the following string should be derivable from $S$:

$$1110D10M_2001BB$$

note we can derive

$$1110D_M1110B$$

corresponding to starting configuration of $M$.

So far we have

$$S \Rightarrow^* wDM_1wB \text{ for all } w \in \{0,1\}^*$$

Next, interpret sequence of symbols as state of a Turing machine computation. $M_q$ represents $TM$ in state $q$ (where $1$ is the initial state), $B$ represents an infinite sequence of blanks.
The first $w$ is a copy of the input, separated from the computation using $D$. Introduce grammar rules to simulate transitions. We want

$$M \text{ accepts } w \iff DM_1wB \Rightarrow^* \epsilon$$

and furthermore, if $M$ does not accept $w$, $DM_1wB$ does not derive any word in $I^*$. 

Further rules of $G$ will simulate TM computation.
$G$ gets variable $M_i$ for each state $i$ of $M$
For each symbol of the tape alphabet $A$, give $G$ a variable that represents that symbol. For $x \in A$ let $V_x$ be a variable that represents $x$.
$B$ denotes a blank tape symbol
$B$ denotes an infinite sequence of blanks.
Rules of $G$: a string derivable from $S$ should represent a state of the TM computation, i.e. a tape with a sequence of symbols and the TM located somewhere on that tape.

Example: suppose that $M$ got input $1110$ and reached a point later on with a tape $10001B1bb\ldots$ where $M$ is located at the 3rd tape square in state $q_2$, the following string should be derivable from $S$:

$$1110D10M_2001BB$$

note we can derive

$$1110D_M1110B$$

corresponding to starting configuration of $M$. 

Further rules of $G$ will simulate TM computation.
$G$ gets variable $M_i$ for each state $i$ of $M$
For each symbol of the tape alphabet $A$, give $G$ a variable that represents that symbol. For $x \in A$ let $V_x$ be a variable that represents $x$.
$B$ denotes a blank tape symbol
$B$ denotes an infinite sequence of blanks.
Rules of $G$: a string derivable from $S$ should represent a state of the TM computation, i.e. a tape with a sequence of symbols and the TM located somewhere on that tape.
Transitions of $M$ become productions of $G$:

- **transition:** $\delta(Q_1, 0) = (Q_2, 1, R)$ becomes production:

$$M_10 \rightarrow 1M_2$$

- **transition:** $\delta(Q_1, 0) = (Q_2, 1, L)$ becomes production:

$$for\  all\ x \in A, \ xM_10 \rightarrow M_2x1$$

$\delta(Q_1, \bar{b}) = (Q_2, 0, R)$ becomes production:

$$M_1\bar{b} \rightarrow 0M_2$$

$$M_1b \rightarrow 0M_2b$$

---

Finally, we want to make the grammar leave behind the input word $w$ provided that $M$ accepts $w$.

Given accepting state $t$ and $a \in A$ for which there is no transition in $M$, add grammar rule

$$M_t a \rightarrow F'$$

$F'$ is a symbol that "mops up" other symbols – use rules of the form:

$$F'\alpha \rightarrow F' \quad \text{and} \quad \alpha F' \rightarrow F'$$

where $\alpha$ is any symbol other than $D$. One more rule:

$$DF' \rightarrow \epsilon$$

---

Example: starting configuration of $T$ will have $T$ in state $Q_1$ at LHS of tape. Suppose input is the word 10011.

Derivation:

---

Going the other way: how to convert a grammar into an "equivalent" Turing machine?

Easiest to use a nondeterministic TM.

General idea: for $n = 1, 2, 3 \ldots$ program the TM to generate all words derivable from $S$ using $n$ steps of derivation. If the TM finds word $w$, it halts and accepts. Otherwise, keep trying! The tape is infinite, so the machine can store the set of all words derivable after $n$ steps on the tape. Lots of implementation detail has been omitted.
It is undecidable to figure out whether a given word is derivable from a given unrestricted grammar. We have essentially reduced from the problem of recognising $L_{\text{accept}}$.

Recall “context-sensitive grammar” – the restriction is that RHS of a rule must be at least as long as LHS.

We can verify that context-sensitive grammars are more restricted than context-free grammars in terms of what languages they can represent, by showing that the problem of determining whether a word $w$ is derivable from a given CSG $G$ is decidable.

Deciding whether $w$ is derivable using (context-sensitive) $G$:
For $n = 1, 2, 3, \ldots$ let $S_n$ be the set of all words of length $\leq n$ that are derivable using $G$.

Finding $S_n$: any word $w$ in $S_n$ must have a derivation of length at most $|V \cup A|^n$ steps. A longer derivation is possible but would contain a repetition

$$S \Rightarrow^* w' \Rightarrow^* w' \Rightarrow^* w$$

since all intermediate words $w'$ have length at most $n$.

So, there’s a limit to how many steps of derivation we need to try out to see if a given word is derivable.

---

CSGs and automata

Context-sensitive grammars correspond to the class of “linearly bounded Turing machines” - this is a nondeterministic TM with a linear bound on the number of tape squares that may be used during computation.

For given input length $n$, if $kn$ is the upper bound on how many squares may be used, we have a bound on the total number of configurations reachable during computation. Hence the acceptance problem can be seen to be decidable.