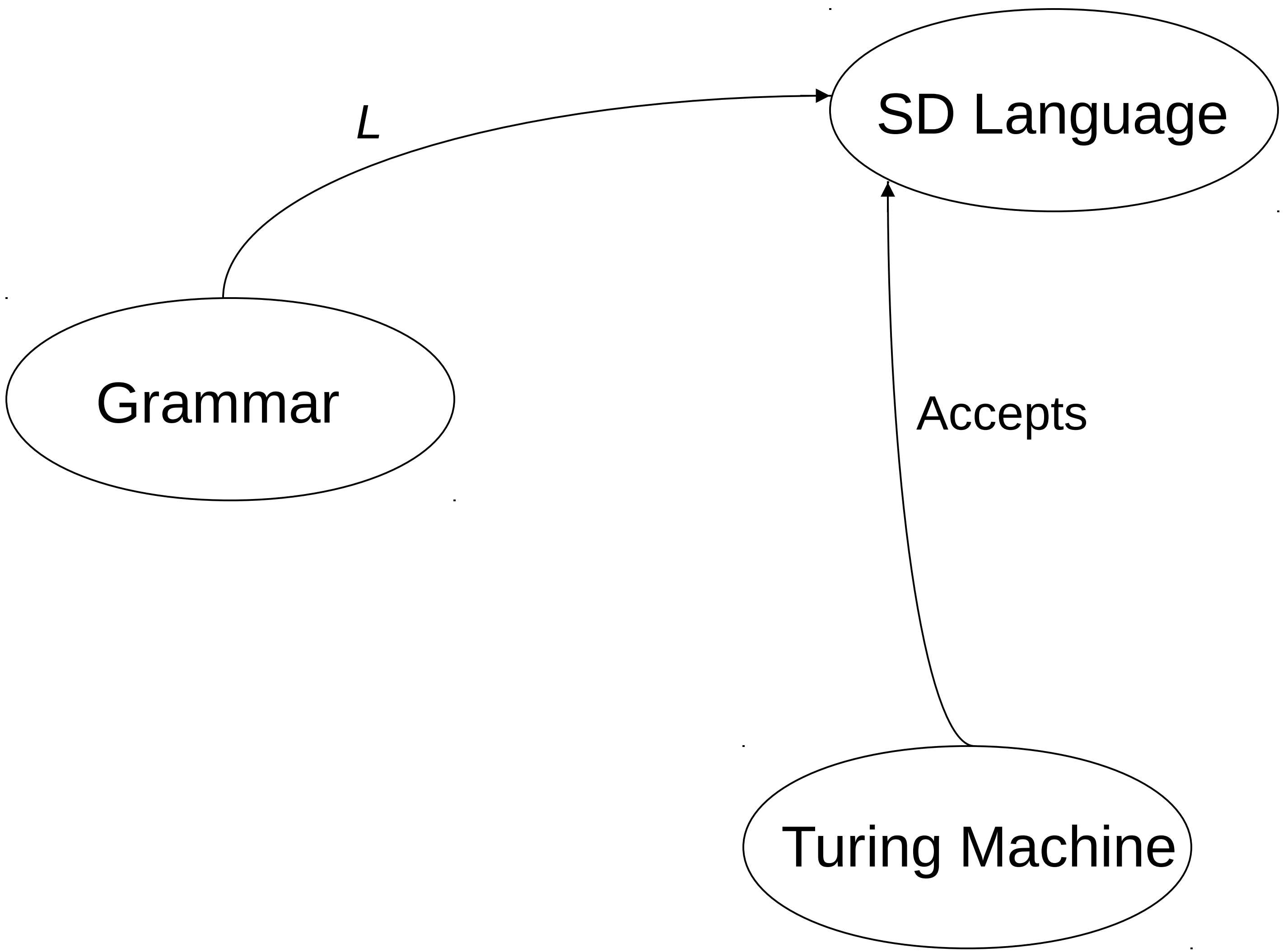
Unrestricted Grammars

Chapter 23

Grammars, SD Languages, and Turing Machines



Unrestricted Grammars

An ***unrestricted grammar*** G is a quadruple (V, Σ, R, S), where:

- V is an alphabet,
- Σ (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: $A^nB^nC^n = \{a^nb^nc^n, n \geq 0\}$.

$$S \rightarrow aBSc$$
$$S \rightarrow \varepsilon$$
$$Ba \rightarrow aB$$
$$Bc \rightarrow bc$$
$$Bb \rightarrow bb$$

Proof:

- Only strings in $A^nB^nC^n$:

- All strings in $A^nB^nC^n$:

Another Example

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

$S \rightarrow ABCS$
$S \rightarrow \varepsilon$
$AB \rightarrow BA$
$BA \rightarrow AB$
$BC \rightarrow CB$
$CB \rightarrow BC$
$AC \rightarrow CA$
$CA \rightarrow AC$
$A \rightarrow a$
$B \rightarrow b$
$C \rightarrow c$

$WW = \{ww : w \in \{a, b\}^*\}$

Idea:

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

aaabb*C*bbaaa#

2. Reverse the second half.

$WW = \{ww : w \in \{a, b\}^*\}$

$S \to T\#$	/* Generate the wall exactly once.
$T \to aTa$	/* Generate wCw^R.
$T \to bTb$	″
$T \to C$	″
$C \to CP$	/* Generate a pusher P
$Paa \to aPa$	/* Push one character to the right
$Pab \to bPa$	to get ready to jump.
$Pba \to aPb$	″
$Pbb \to bPb$	″
$Pa\# \to \#a$	/* Hop a character over the wall.
$Pb\# \to \#b$	″
$C\# \to \varepsilon$	

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 → TM): by construction of an NDTM.

If (TM → grammar): by construction of a grammar that mimics the behavior of a semideciding TM.

Grammar → Turing Machine

Given *G*, produce a Turing machine *M* that semidecides *L*(*G*).

M will be nondeterministic and will use two tapes:

	❑	a	b	a	b	❑	❑	
❑	1	0	0	0	0	0	0	❑
	a	*S*	*T*	a	a	b	❑	
	1	0	0	0	0	0	0	

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.

Turing Machine → Grammar

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

The first (generate) part of *G*:
Create all strings over Σ* of the form:

w = # ❑ ❑ q000 a_1 a_1 a_2 a_2 a_3 a_3 ❑ ❑ #

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

❑ ❑ a 1 b 2 c c b 4 q001 a 3

Examples of rules:

```
q100 b b       → b 2 q101
a a q011 b 4 → q011 a a b 4
```

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:

$\forall x$	x A $\rightarrow$ A x	/* Sweep A to the left.
$\forall x, y$	#A x y $\rightarrow$ x #A	/* Erase duplicates.
	#A# $\rightarrow$ ε	

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 $L(G) = \varnothing$?

Or, as languages:

- $L_a = \{\langle G, w\rangle : w \in L(G)\}$.
- $L_\varepsilon = \{\langle G\rangle : \varepsilon \in L(G)\}$.
- $L_= = \{\langle G_1, G_2\rangle : L(G_1) = L(G_2)\}$.
- $L_\varnothing = \{\langle G\rangle : L(G) = \varnothing\}$.

None of these questions is decidable.

$L_a = \{<G, w> : w \in L(G)\}$ is not in D.

Proof: Let R be a mapping reduction from:

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

But no machine to decide A can exist, so neither does *Oracle*.