Derivations in Rewriting Systems

Assume strings over some alphabet \( \Gamma \)

For grammars, \( \Gamma = V \cup \Sigma \)

For semi-Thue systems \( \Gamma = \Sigma \)

Assume rules \( \alpha_i \rightarrow \beta_i \) \( 1 \leq i \leq n \)

Simple application is, if \( \omega = \delta \alpha_i \beta_i \gamma \)

Then \( \omega \Rightarrow \omega' \) if \( \omega' = \delta \beta_i \gamma \)

This is a one-step derivation.

We extend to \( \Rightarrow, \Rightarrow*, \Rightarrow^* \)

\( \omega \Rightarrow \omega' \) IFF \( \omega = \omega' \)

\( \omega \Rightarrow \omega' \) IFF \( \exists \omega'', \omega \Rightarrow \omega'' \Rightarrow \omega' \) \( \Rightarrow^* \)

\( \omega \Rightarrow \omega' \) IFF \( \exists \omega'', \omega \Rightarrow \omega'' \Rightarrow \omega' \) \( \Rightarrow^* \)

\( \omega \Rightarrow^* \omega' \) IFF \( \omega \Rightarrow \omega' \) FOR SOME \( \Rightarrow^* \)

\( \omega \Rightarrow^* \omega' \) \( \Rightarrow^* \) IS THE REFLEXIVE, TRANSITIVE

CLOSURE OF \( \Rightarrow \)

\( \omega \Rightarrow \omega' \) IFF \( \omega \Rightarrow^* \omega' \) FOR SOME \( \Rightarrow^* \)

\( \Rightarrow \) IS THE TRANSITIVE CLOSURE

OF \( \Rightarrow \)

\( \Rightarrow \) \( \Rightarrow^* \) AND ITS VARIANTS DENOTE THE

OPERATION (OR RELATION) OF

DERIVATION
Derivations in Grammars

Let $G = (V, \Sigma, R, S)$ such that $S \notin V$

Let $\Rightarrow$ be the derivation operation (relation) associate with $R$ applied to strings over $V \cup \Sigma$

Note: Any string over $V \cup \Sigma$ is called a sentential form.

Any string over $\Sigma$ is a sentence.

We care mostly about sentences derivable from the trivial sentential form $S$

$$L(G) = \{ w \mid w \in \Sigma^* \text{ and } S \Rightarrow w \}$$

Can limit to $\Rightarrow$

Since $S$ is not a sentence.
DFA to RIGHTLINEAR

\( Q = (Q, \Sigma, S, q_0, F) \)
\( G = (Q, \Sigma, R, q_0) \)

\( R: \)
\( q \rightarrow a P \quad \text{WHENEVER} \quad s(q, a) = P \)
\( q \rightarrow \lambda \quad \text{WHENEVER} \quad q \notin F \)

Prove: \( s^*(q_0, w) \in F \iff q_0 \Rightarrow^* w \)

Lemma: Show \( s^*(q, w) = P \iff q \Rightarrow^* w_P \)

Use induction on \(|w|\)

Basis: \(|w| = 1 \quad s^*(q, \lambda) = q \)
\( q \Rightarrow^\lambda q \), i.e., \( q = \lambda q \)

IH: \(|w| = k \). Assume if \(|w| = k\) then
\( s^*(q, w) = P \iff q \Rightarrow^* w_P \)

IS: \(|w| = k+1\). Hence \( w = xa_x\), \(|x| = k \)

By IH \( s^*(q, x) = P \iff q \Rightarrow^* x_P \)

Let \( s^*(q, xa) = s(s^*(q, x_a)) = s(P, a) = t \)

By construction \( s(P, a) = t \iff P \rightarrow a \epsilon R \)

Thus, \( q \Rightarrow^* x_P \Rightarrow xa \).

And so \( q \Rightarrow^* w\).
Final Step: DFA to Right Linear

Lemma shows \( S^*(q,w) = p \iff q \Rightarrow w \in p \)

Clearly then \( S^*(q_0,w) = p \iff q_0 \Rightarrow w \in p \)

But then \( w \in L(A) \iff S^*(q_0,w) \in F \)

And so \( w \in L(A) \iff q_0 \Rightarrow w \in p \in F \)

1. If \( q_0 \Rightarrow w \in p \in F \)
2. If \( q_0 \Rightarrow w \in p \in F \)

But then

\( w \in L(A) \iff w \in L(G) \)
CONTEXT-FREE GRAMMARS

\[ G = (V, \Sigma, R, S) \]

\( V \): finite set of variables (non-terminals)
\( \Sigma \): finite set of terminals
\( S \): start symbol, \( \in \Sigma \)
\( R \): rules, each of form

\[ A \to \alpha \quad A \in V \quad \alpha \in (V \cup \Sigma)^* \]

EXAMPLE NON-REGULAR CFLs

\[ S \to aSB | \lambda \quad L = \{ a^n b^n \mid n \geq 0 \} \]

\[ S \to \lambda | a | b | aSa | bSb \]
\[ L = \{ \text{palindromes over } \{a, b\} \} \]

\[ S \to aSBs | bSaS | \lambda \]
\[ L = \{ \text{strings over } \{a, b\} \text{ with equal number of a's and b's} \} \]
Using Notation

This notation is useful for inductive proofs.

Consider \( G = (\Sigma, \Gamma, \delta, S, \rho) \) where
\[ R: S \Rightarrow \lambda | \alpha | \beta | \sigma | \lambda S \beta \lambda \]

Claim: \( L(G) = \{ w | \text{w is } \{a, b\}^* \text{ and w is a palindrome} \} \)

Lemma: \( S \Rightarrow \beta \), where \( \beta \) contains a non-terminal
iff \( \beta = \text{x } S \text{x } \rho \), where \( x \in \{a, b\}^* \). We attack
by considering \( S \Rightarrow \beta \), showing \( \beta = \text{x } S \text{x } \rho \)
for any and all \( x \in \{a, b\}^* \), \( |x| = \rho \)

Base: \( \rho = 0 \), \( S \Rightarrow S \) by defn. of \( \Rightarrow \). But \( S = \lambda S \lambda \)
and this is only string of form \( \text{x } S \text{x } \rho \), \( |x| = 0 \)

Induction Hypothesis: \( \rho = n \), assume \( S \Rightarrow \beta \) iff \( \beta = \text{x } S \text{x } \rho \)

Induction Step: \( \rho = n + 1 \), show \( S \Rightarrow \beta \) iff \( \beta = \text{x } S \text{x } \rho \), \( x \in \{a, b\}^* \), \( |x| = n + 1 \)

By defn. of \( \Rightarrow \), \( S \Rightarrow \beta \) iff \( S \Rightarrow \alpha \Rightarrow \beta \)

By rules in \( R \) and our constraint, that we retain
a variable (S) in the derivation,

\[ \alpha = \sigma | \sigma A | \sigma A \sigma | \lambda \sigma \]

By \( \text{IH} \), \( \beta = \text{x } S \text{x } \rho \) for any and all \( x \in \{a, b\}^* \), \( |x| = n \)

Combining, we get \( S \Rightarrow \text{x } S \text{x } \sigma \) or \( S \Rightarrow \text{x } S \text{x } \lambda \)

\( x \Rightarrow \lambda x \) gives us all and only those strings
of length \( n+1 \) in \( \{a, b\}^* \). This proves IS
and hence the original hypothesis.

Theorem: All strings in \( \{ w | \text{w is } \{a, b\}^* \text{ and w is a palindrome} \} \)
are of form \( \text{x } S \text{x } \rho \), \( x \in \{a, b\}^* \) or \( x \Lambda x \rho \)

Applying \( S \Rightarrow \gamma \) or \( S \Rightarrow \sigma \) or \( S \Rightarrow \lambda \) gets
all and only these forms and the application
of such rule is the only means of
deriving a sentence in \( L(G) \).
A PRACTICAL GRAMMAR
(Sort of)

\[ G = (\{E, a, +, -, *, /, (, )\}, R, E) \]

\[ R : E \to E + E \mid E - E \mid E \cdot E \mid E / E \mid (E) \]

A DERIVATION

\[ E \Rightarrow E + E \Rightarrow E \cdot E \cdot E \]
\[ \Rightarrow a \cdot E + E \Rightarrow a \cdot a + E \]
\[ \Rightarrow a \cdot a + a \]

TREE VERSION

```
  E
 /\  
E+ E
 /\  
 E* E
 /\  
 a  a  a
```

FRONTIER IS

\[ a \cdot a + a \]

OOPS: Can get by

```
  E
 /\  
E *  
 /\  
 a   a
```

SAME FRONTIER

\[ E \Rightarrow E * E \Rightarrow a \cdot E \Rightarrow a \cdot E + E \]
\[ \Rightarrow a \cdot a + E \Rightarrow a \cdot a + a \]
Ambiguity

Grammar is ambiguous if there is a string in language such that

W can be derived from S by two distinct leftmost deriv.
(always rewrite leftmost non-terminal before others)

\[ S \xrightarrow{LM} w + S \xrightarrow{LM} w \] where intermediates differ

W can be derived from S by two distinct rightmost derivations

\[ S \xrightarrow{RM} w + S \xrightarrow{RM} w \] where paths differ

W has two distinct parse trees topologically different with same frontier

A language L is inherently ambiguous if all grammars for L are ambiguous
**Arithmetic Language is not Ambiguous**

**Alternative Grammar Rules**

\[ R: E \rightarrow E + T \mid E - T \mid T \]
\[ T \rightarrow T \times F \mid T / F \mid F \]
\[ F \rightarrow (E) \mid a \]

**Lowest, Left to Right**

\[ a \times a + a \] can only be gotten by **Leftmost**

\[ E \Rightarrow E + T \Rightarrow T + T \Rightarrow T \times F + T \]
\[ \Rightarrow F \times F + T \Rightarrow a \times F + T \Rightarrow a \times a + T \]
\[ \Rightarrow a \times a + F \Rightarrow a \times a + a \]

**Highest, Left to Right**

\[ E \Rightarrow E + T \Rightarrow E + F \Rightarrow E + a \Rightarrow T + a \]
\[ \Rightarrow T \times F + a \Rightarrow T \times a + a \]
\[ \Rightarrow F \times a + a \Rightarrow a \times a + a \]

**Tree**

```
     E
    /  \
   /    \
  E     T
 /      /  \
/        F   F
 T        /   /  \
 /     a    a    a
 F
    /  \
   a   a
```
An Inherently Ambiguous Language

\[ L = \{ a^i b^j c^k \mid i = j \text{ or } j = k \} \]

\[
S \rightarrow A <bc> | <ab> c \\
A \rightarrow a A | \lambda \\
C \rightarrow c C | \lambda \\
<bc> \rightarrow b <bc> c | \lambda \\
<ab> \rightarrow a <ab> b | \lambda
\]

can get \( L = \{ a^i b^j c^k \mid i = j \text{ or } j = k \} \) on two paths and there is no way to avoid this
Some Easy CFL Closures

$G_1 = (V_1, \Sigma, R_1, S_1) \quad G_2 = (V_2, \Sigma, R_2, S_2) \quad V_1 \cap V_2 = \emptyset$

Union

$G = (V_1 \cup V_2 \cup \{s\}, \Sigma, R, S)$

$R = R_1 \cup R_2 \cup \{s \rightarrow s_1s_2 \mid s_1 \in \text{Lang}(G_1) \land s_2 \in \text{Lang}(G_2)\}$

or \( s \rightarrow S_1S_2 \).

Concatenation

$R = R_1 \cup R_2 \cup \{s \rightarrow S_1S_2\}$

\[
R = R_1 \cup \{s \rightarrow S_1S_1 \mid \lambda \}
\]

We will see closure under

Intersection with regular
Substitution/Homomorphism

But lack of closure under

Intersection with CFL

Complement
An Interesting CFL

We will prove that \(\{ww \mid w \in \{a, b\}^*\}\) is not a CFL.

The complement of above has two parts

(a) Odd length strings over \(\{a, b\}\)

Clearly regular

\[ S \rightarrow aT \mid bT \]

\[ T \rightarrow 2 \mid aS \mid bS \]

(b) \(\{xy \mid x, y \in \{a, b\}^+ \text{ and } |x| = |y| \text{ and } x \neq y\}\)

Looking at (b) we need one transcrition error from \(x\) to \(y\)

It is a \(\exists\) rather than a \(\forall\)

As is \(ww\)
VIEWING THE STRINGS
IN \( S \times y \mid xy \in \Sigma^* 0^* \), \(|x| = |y|, x \neq y\)

**View 1**

\[
x_1 a x_2 y_1 : b y_2 \quad \text{or} \quad x_1 b x_2 y_1 : a y_2
\]

\(|x_1| = |y_1|, \quad |x_2| = |y_2|

**View 2**

\[
x_1 a y_1 x_2 b y_2 \quad \text{or} \quad x_1 b y_1 x_2 a y_2
\]

\(\text{mid} \quad \text{mid}\)

\[
S \rightarrow AB \mid BA
\]

\[
A \rightarrow CAC \mid a
\]

\[
B \rightarrow CBC \mid b
\]

\[
C \rightarrow a \mid b
\]
Bottom up vs Top Down Parsing

Bottom up uses input to drive process
It is driven by shift/reduce
Shift is push character on a stack
Reduce is replace "handle" of top of stack
with variable A where A \rightarrow "handle"
This is a reduce.

Bottom up hates right recursion
But loves left recursion

Example: \[ E \rightarrow E + T \quad T \rightarrow F \ldots \quad F \rightarrow a \]

\[ a + a \]

Shift a
Reduce a to F
Reduce F to T
Reduce T to E
Shift +
Shift a
Reduce a to F
Reduce F to T
Reduce E + T to E
Top Down vs Bottom Up Parsing

Top Down is predictive and commonly implemented using recursive descent.

If have \( E \rightarrow E + T | E - T | E \)

Which RHS do we use?

Let's say input is \( a + a \), the plus could help us predict

\[
\begin{array}{c}
E \\
/ \big/ \\
E + T
\end{array}
\]

The problem is when see \( E \) again, we would deterministically make same prediction and get infinite descent on \( E \).

Top Down hates left recursion but loves right recursion.
Back to Arithmetic Expressions

E → E + T | E - T | T
T → T * F | T / F | F
F → (E) | α

Is left recursive, this is disastrous for top down

Consider any non-terminal A

Where \( A \rightarrow A_1 A_2 \cdots A_r \beta_1 \cdots \beta_j \)

Not left rec. in A

Can see we get

\[ A \rightarrow (\beta_1 + \beta_2 + \cdots + \beta_j)(\alpha_1 + \alpha_2 + \cdots + \alpha_r)^* \]

Can redo as

\[ A \rightarrow \beta_1 A' | \cdots | \beta_j A' \]

\[ A' \rightarrow \alpha_1 A' | \cdots | \alpha_r A' | \lambda \]

E → TE'
E' → +TE' | -TE' | X

T → FT'
T' → *FT' | /FT' | X

F → (E) | α