Language of a Grammar

If G is a grammar we write

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

Definition: *A language L is* context-free *iff there is a grammar G such that L = L(G)*

start symbol corresponds to start state

variable symbols correspond to states

terminal symbols T correspond to the alphabet Σ

Context-Free Languages and Regular Languages

Theorem: *If L is regular then L is context-free.*

Proof: We know $L = L(A)$ for a DFA A. From A we can build a CFG G such that $L(A) = L(G)$

The variables are A, B, C, with start symbol A, the terminal token are $0, 1$ and the productions are

$$A \to 1A \mid 0B \qquad B \to 0B \mid 1C \qquad C \to \epsilon \mid 0B \mid 1C$$

Context-Free Languages and Regular Languages

Let L_X be the language generated by the grammar with X as a start symbol we prove (mutual induction!) that $w \in L_X$ iff $\hat{\delta}(X, w) = C$ by induction on $|w|$

Such a CFG is called *right regular*

It would be possible also to define L by a *left regular* language with start state C

$$A \rightarrow \epsilon \mid C1 \mid A1 \qquad B \rightarrow A0 \mid C0 \mid B0 \qquad C \rightarrow B1$$

The intuition here is that L_X represents the path from A to X

Example of a derivation

Given the grammar for english above we can generate (*leftmost* derivation)

SENTENCE $\Rightarrow$ SUBJECT VERB OBJECT

$\Rightarrow$ ARTICLE NOUN VERB OBJECT $\Rightarrow$ the NOUN VERB OBJECT

$\Rightarrow$ the NOUN VERB OBJECT $\Rightarrow$ the cat VERB OBJECT

$\Rightarrow$ the cat caught OBJECT $\Rightarrow$ the cat caught ARTICLE NOUN

$\Rightarrow$ the cat caught a NOUN $\Rightarrow$ the cat caught a dog

Derivation Tree

Notice that the following generation is possible (*rightmost* derivation)

SENTENCE $\Rightarrow$ SUBJECT VERB OBJECT

$\Rightarrow$ SUBJECT VERB ARTICLE NOUN

$\Rightarrow$ SUBJECT VERB ARTICLE dog

$\Rightarrow$ SUBJECT VERB a dog $\Rightarrow$ SUBJECT caught a dog

$\Rightarrow$ ARTICLE NOUN caught a dog $\Rightarrow$ ARTICLE cat caught a dog

$\Rightarrow$ the cat caught a dog

Derivation Tree

Both generation corresponds to the same *derivation tree* or *parse tree* which reflects the *internal structure* of the sentence

Number of left derivations of one word

= number of right derivations

= number of parse trees

A grammar for arithmetical expressions

$S \to (S) \mid S + S \mid S \times S \mid I$

$I \to 1 \mid 2 \mid 3$

The terminal symbols are $\{(,), +, \times, 1, 2, 3\}$

The variable symbols are S and I

Ambiguity

Definition: *A grammar G is* ambiguous *iff there is some word in L(G) which has two distinct derivation trees*

Intuitively, there are two possible meaning of this word

Example: the previous grammar for arithmetical expression is ambigiuous since the word $2 + 1 \times 3$ has two possible parse trees

Ambiguity

An example of ambiguity in programming language is **else** with the following production

$C \rightarrow \textbf{if}\ b\ \textbf{then}\ C\ \textbf{else}\ C$

$C \rightarrow \textbf{if}\ b\ \textbf{then}\ C$

$C \rightarrow s$

Ambiguity

A word like

$\textbf{if } b \textbf{ then if } b \textbf{ then } s \textbf{ else } s$

can be interpreted as

$\textbf{if } b \textbf{ then } (\textbf{if } b \textbf{ then } s \textbf{ else } s)$

or

$\textbf{if } b \textbf{ then } (\textbf{if } b \textbf{ then } s) \textbf{ else } s$

Context-Free Languages and Inductive Definitions

Each CFG can be seen as an inductive definition

For instance the grammar for arithmetical expression can be seen as the following inductive definition

- $1, 2, 3$ are arithmetical expressions
- if w is an arithmetical expression then so is (w)
- if w_1, w_2 are arithmetical expressions then so are $w_1 + w_2$ and $w_1 \times w_2$

A natural way to do proofs on context-free languages is to follow this inductive structure

Context-Free Languages and Regular Languages

The following language $L = \{a^n b^n \mid n \geqslant 1\}$ is context-free

We know that it is *not* regular

Proposition: *The following grammar G generates L*

$$S \rightarrow ab \mid aSb$$

Context-Free Languages and Regular Languages

We prove $w \in L(G)$ implies $w \in L$ by induction on the derivation of $w \in L(G)$

- $ab \in L(G)$
- if $w \in L(G)$ then $awb \in L(G)$

We can prove also $w \in L(G)$ implies $w \in L$ by induction on the length of a derivation $S \Rightarrow^* w$

We prove $a^n b^n \in L(G)$ by induction on n

Abstract Syntax

The parse tree has often too much information w.r.t. the internal structure of a document. This structure is best reflected by an *abstract syntax tree*. We give only an example here.

Here is BNF for arithmetic expression

$$E \to E + E \mid E \times E \mid (E) \mid I \qquad I \to 1 \mid 2 \mid 3$$

Parse tree for $2 + (1 \times 3)$

Abstract Syntax

This can be compared with the abstract syntax tree for the expression $2+(1\times 3)$

Concrete syntax describes the way documents are written while *abstract* syntax describes the pure structure of a document.

Abstract Syntax

In Haskell, use of data types for abstract syntax

```
data Exp = Plus Exp Exp | Times Exp Exp | Num Atom

data Atom = One | Two | Three

ex = Plus Two (Times One Three)
```

Ambiguity

Definition: *A grammar G is* ambiguous *iff there is some word in L(G) which has two distinct derivation trees*

Intuitively, there are two possible meaning of this word

Example: the previous grammar for arithmetical expression is ambigiuous since the word $2 + 1 \times 3$ has two possible parse trees

Ambiguity

Let Σ be $\{0, 1\}$.

The following grammar of parenthesis expressions is *ambiguous*

$E \to \epsilon \mid EE \mid 0E1$

A simple example

$\Sigma = \{0, 1\}$

$L = \{uu^R \mid u \in \Sigma^*\}$

This language is *not* regular: using the pumping lemma on $0^k 1 0^k$

$L = L(G)$ for the grammar

$S \rightarrow \epsilon \mid 0S0 \mid 1S1$

We prove that if $S \Rightarrow^* v$ then $v \in L$ by induction on the *length of* $S \Rightarrow^* v$

We prove $uu^R \in L(G)$ if $u \in \Sigma^*$ by induction on the length of u

A simple example

Theorem: *The grammar for S is not ambiguous*

Proof: By induction on $|v|$ we prove that there is at most one production $S \Rightarrow^* v$

Polish notation

The following is a grammar for arithmetical expressions

$E \rightarrow *EE \mid +EE \mid I, \quad I \rightarrow a \mid b$

Theorem: *This grammar is* not *ambiguous*

We show by induction on $|u|$ the following.

Lemma: *for any k there is at most one leftmost derivation of $E^k \Rightarrow^* u$*

Polish notation

The proof is by induction on $|u|$. If $|u| = n+1$ with $n \geqslant 1$ there are three cases

(1) $u = +v$ then the derivation has to be of the form

$$E^k \Rightarrow +EEE^{k-1} \Rightarrow^* +v$$

for a derivation $E^{k+1} \Rightarrow^* v$ and we conclude by induction hypothesis

(2) $u = *v$ then the derivation has to be of the form

$$E^k \Rightarrow *EEE^{k-1} \Rightarrow^* *v$$

for a derivation $E^{k+1} \Rightarrow^* v$ and we conclude by induction hypothesis

Polish notation

(3) $u = iv$ with $i = a$ or $i = b$, then the derivation has to be of the form

$$E^k \Rightarrow iE^{k-1} \Rightarrow^* iv$$

for a derivation $E^{k-1} \Rightarrow^* v$ and we conclude by induction hypothesis

Polish notation

It follows from this result that we have the following property.

Corollary: *If* $*u_1u_2 = *v_1v_2 \in L(E)$ *then* $u_1 = v_1$ *and* $u_2 = v_2$. *Similarly if* $+u_1u_2 = +v_1v_2 \in L(E)$ *then* $u_1 = v_1$ *and* $u_2 = v_2$.

but the result says also that if $u \in L(E)$ then there is a *unique* parse tree for u.

Ambiguity

Now, a more complicated example. Let Σ be $\{0,1\}$.

The following grammar of parenthesis expressions is *ambiguous*

$E \rightarrow \epsilon \mid EE \mid 0E1$

We replace this by the following *equivalent* grammar

$S \rightarrow 0S1S \mid \epsilon$

Lemma: *L(S) = L(E)*

Theorem: *The grammar for S is* not *ambiguous*

Ambiguity

Lemma: $L(S)L(S) \subseteq L(S)$

Proof: we prove that if $u \in L(S)$ then $uL(S) \subseteq L(S)$ by induction on $|u|$

If $u = \epsilon$ then $uL(S) = L(S)$

If $|u| = n+1$ then $u = 0v1w$ with $v, w \in L(S)$ and $|v|, |w| \leqslant n$. By induction hypothesis, we have $wL(S) \subseteq L(S)$ and so

$$uL(S) = 0v1wL(S) \subseteq 0v1L(S) \subseteq L(S)$$

since $v \in L(S)$ and $0L(S)1L(S) \subseteq L(S)$ Q.E.D.

Ambiguity

We can also do an induction on the length of a derivation $S \Rightarrow^* u$

Using this lemma, we can show $L(E) \subseteq L(S)$

Ambiguity

Lemma: $L(E) \subseteq L(S)$

Proof: We prove that if $u \in L(E)$ then $u \in L(S)$ by induction on the length of a derivation $E \Rightarrow^* u$

If $E \Rightarrow \epsilon = u$ then $u \in L(S)$

If $E \Rightarrow EE \Rightarrow^* vw = u$ then by induction $v, w \in L(S)$ and by the previous Lemma we have $u \in L(S)$

If $E \Rightarrow 0E1 \Rightarrow^* 0v1 = u$ then by induction $v \in L(S)$ and so $u = 0v1\epsilon \in L(S)$. Q.E.D.

Ambiguity

The proof that the grammar for S is not ambiguous is difficult

One first tries to show that there is at most one left-most derivation

$S \Rightarrow^*_{lm} u$

for any string $u \in \Sigma^*$

If u is not ϵ we have that u should be $0u_1$ and then the derivation should be

$S \Rightarrow 0S1S \Rightarrow 0u_1$

with $S1S \Rightarrow u_1$

Ambiguity

This suggests the following statement $\psi(u)$ to be proved by induction on the length of u

For any k there exists *at most* one leftmost derivation $S(1S)^k \Rightarrow^* u$

We can then prove $\psi(u)$ by induction on $|u|$

If $u = \epsilon$ we should have $k = 0$ and the derivation has to be $S \Rightarrow \epsilon$

Ambiguity

If $\psi(v)$ holds for $|v| = n$ and $|u| = n + 1$ then $u = 0v$ or $u = 1v$ with $|v| = n$. We have two cases

(1) $u = 1v$ and $S(1S)^k \Rightarrow^* 1v$, the derivation has the form

$$S(1S)^k \Rightarrow \epsilon(1S)^k \Rightarrow^* 1v$$

for a derivation $S(1S)^{k-1} \Rightarrow^* v$ and we conclude by induction hypothesis

(2) $u = 0v$ and $S(1S)^k \Rightarrow^* 0v$, the derivation has the form

$$S(1S)^k \Rightarrow 0S1S(1S)^k \Rightarrow^* 0v$$

for a derivation $S(1S)^{k+1} \Rightarrow^* v$ and we conclude by induction hypothesis

Inherent Ambiguity

There exists a context-free language L such that for any grammar G if $L = L(G)$ then G is ambiguous

$L = \{a^n b^n c^m d^m \mid n, m \geqslant 1\} \cup \{a^n b^m c^m d^n \mid n, m \geqslant 1\}$

L is context-free

$S \rightarrow AB \mid C \qquad A \rightarrow aAb \mid ab$

$B \rightarrow cBd \mid cd \qquad C \rightarrow aCd \mid aDd \qquad D \rightarrow bDc \mid bc$

Eliminating ϵ- and unit productions

Definition: *A* unit *production is a production of the form* $A \to B$ *with* A, B *non terminal symbols.*

This is similar to ϵ-transitions in a ϵ-NFA

Definition: *A* ϵ*-production is a production of the form* $A \to \epsilon$

Theorem: For any CFG G there exists a CFG G' with no ϵ- or unit productions such that $L(G') = L(G) - \{\epsilon\}$

Elimination of unit productions

Let P_1 be a system of productions such that if $A \to B$ and $B \to \beta$ are in P_1 then so is $A \to \beta$ and $G_1 = (V, T, P_1, S)$.

Let P_2 the set of non unit productions of P_1 and $G_2 = (V, T, P_2, S)$

Theorem: $L(G_1) = L(G_2)$

Elimination of unit productions

Proof: If $u \in L(G_1)$ and $S \Rightarrow^* u$ is a derivation of *minimal* length then this derivation is in G_2. Otherwise it has the shape

$$S \Rightarrow^* \alpha_1 A \alpha_2 \Rightarrow \alpha_1 B \alpha_2 \Rightarrow^n \beta_1 B \beta_2 \Rightarrow \beta_1 \beta \beta_2 \Rightarrow^* u$$

and we have a shorter derivation

$$S \Rightarrow^* \alpha_1 A \alpha_2 \Rightarrow^n \beta_1 A \beta_2 \Rightarrow \beta_1 \beta \beta_2 \Rightarrow^* u$$

contradiction.

Elimination of unit productions

$S \to CBh \mid D$

$A \to aaC$

$B \to Sf \mid ggg$

$C \to cA \mid d \mid C$

$D \to E \mid SABC$

$E \to be$

Elimination of unit productions

We eliminate unit productions

$S \to SABC \mid be \mid CBh$

$A \to aaC$

$B \to Sf \mid ggg$

$C \to cA \mid d$

Elimination of ϵ-productions

If $G = (V, T, P, S)$ build the new system P_1 closing P by adding rules

If $A \rightarrow \alpha B \beta$ and $B \rightarrow \epsilon$ then $A \rightarrow \alpha\beta$

We have $L(G_1) = L(G)$. Let P_2 the system obtained from P_1 by taking away all ϵ-productions

Theorem: $L(G_2) = L(G) - \{\epsilon\}$

Proof: We clearly have $L(G_2) \subseteq L(G_1)$. We prove that if $S \Rightarrow^* u,\ u \in T^*$ and $u \neq \epsilon$ is a production of *minimal* length then it does not use any ϵ-production, so it is a derivation in G_2. Q.E.D.

Eliminating ϵ- and unit productions

Starting from $G = (V, T, P, S)$ we build a larger set P_1 of productions containing P and closed under the two rules

1. if $A \rightarrow w_1 B w_2$ and $B \rightarrow \epsilon$ are in P_1 then $A \rightarrow w_1 w_2$ is in P_1

2. if $A \rightarrow B$ and $B \rightarrow w$ are in P_1 then so is $A \rightarrow w$

We add only productions whose right-handside is a subtring of an old right-handside, so this process stops.

It can be shown that if $L(V, T, P_1, S) = L(G)$ and that if P' is the set of productions in P_1 that are not ϵ- neither unit production then $L(V, T, P', S) = L(G) - \{\epsilon\}$

Eliminating ϵ- and unit productions

Example: If we start from the grammar

$$S \rightarrow \; aSb \mid SS \mid \epsilon$$

we get first the new productions

$$S \rightarrow \; ab \mid S \mid S$$

and if we eliminate the ϵ- and unit productions we get

$$S \rightarrow \; aSb \mid SS \mid ab$$

Eliminating ϵ- and unit productions

Example: If we start from the grammar

$$S \rightarrow \ AB \qquad A \rightarrow aAA \mid \epsilon \qquad B \rightarrow bBB \mid \epsilon$$

we get first the new productions

$$S \rightarrow \ A \mid B \qquad A \rightarrow aA \mid a \qquad B \rightarrow bB \mid b$$

and if we eliminate the ϵ- and unit productions we get

$$S \rightarrow \ AB \mid aAA \mid aA \mid a \mid bB \mid b \qquad A \rightarrow aAA \mid aA \mid a \quad B \rightarrow bBB \mid bB \mid b$$

Eliminating Useless Symbols

A symbol X is *useful* if there is some derivation $S \Rightarrow^* \alpha X \beta \Rightarrow^* w$ where w is in T^*

X can be in V or T

X is *useless* iff it is not useful

X is *generating* iff $X \Rightarrow^* w$ for some w in T^*

X is *reacheable* iff $S \Rightarrow^* \alpha X \beta$ for some α, β

Reachable Symbols

By analogy with accessible states, we can define *accessible* or *reachable* symbols. We give an inductive definition

- **BASIS:** The start symbol S is reachable
- **INDUCTION:** If A is reachable and $A \to w$ is a production, then all symbols occuring in w are reachable.

Reachable Symbols

Example: Consider the following CFG

$$S \rightarrow\ aB \mid BC \qquad A \rightarrow\ aA \mid c \mid aDb$$

$$B \rightarrow DB \mid C \qquad C \rightarrow b \quad D \rightarrow B$$

Then s is accessible, hence also B and C, and hence D is accessible.

But A is *not* accessible.

We can take away A from this grammar and we get the same language

$$S \rightarrow\ aB \mid BC \quad B \rightarrow DB \mid C \qquad C \rightarrow b \quad D \rightarrow B$$

Generating Symbols

We define when an element of $V \cup T$ (terminal or non terminal symbols) is generating by an *inductive definition*

- **BASIS:** all elements of T are generating
- **INDUCTION:** if there is a production $X \to w$ where all symbols occuring in w are generating then X is generating

This gives exactly the generating variables

Generating Symbols

Example: We consider

$$S \rightarrow \; aS \mid W \mid U \qquad W \rightarrow aW$$
$$U \rightarrow a \quad V \rightarrow aa$$

Then U, V are generating because $U \rightarrow a \quad V \rightarrow aa$

Hence S is generating because $S \rightarrow U$

W is not generating, we have only $W \rightarrow aW$ for production for W

Eliminating Useless Symbols

To eliminate useless symbols in a grammar G, first eliminate all nongenerating symbols we get an equivalent grammar G_1 and then eliminate all symbols in G_1 that are non reachable.

We get a grammar G_2 that is equivalent to G_1 and to G

We have to do this in this order

Examples: For the grammar

$$S \rightarrow AB \mid a \qquad A \rightarrow b$$

B is not generating, we get the grammar

$$S \to a \qquad A \to b$$

and then A is not reachable we get the grammar

$$S \to a$$

Elimination of useless variables

$$S \to gAe \mid aYB \mid CY, \qquad A \to bBY \mid ooC$$

$$B \to dd \mid D, \qquad C \to jVB \mid gi$$

$$D \to n, \qquad U \to kW$$

$$V \to baXXX \mid oV, \qquad W \to c$$

$$X \to fV, \qquad Y \to Yhm$$

Elimination of useless variables

Simplified grammar

$S \to gAe$

$A \to ooC$

$C \to gi$

Linear production systems

Several algorithms we have seen are instances of graph searching algorithm/derivability in linear production systems

Linear Production systems

For testing for accessibility, for the grammar

$S \to aB \mid BC, \qquad A \to aA \mid c \mid aDb$

$B \to DB \mid C, \qquad C \to b \mid B$

we associate the production system

$\to S, \quad S \to B, \quad S \to C$

$A \to A, \quad A \to D, \quad B \to B$

$B \to D, \quad B \to C, \quad C \to B$

and we can produce S, B, D, C

Linear Production systems

A lot of problems in elementary logic are of this form

$A \rightarrow B, \quad B \rightarrow C, \quad A, C \rightarrow D$

What can we deduce from A?

Linear Production systems

For computing *generating* symbols we have a more general form of production system

For instance for the grammar

$$A \to ABC, \quad A \to C, \quad B \to Ca, \quad C \to a$$

we can associate the following production system

$$A, B, C \to A, \quad C \to A, \quad C \to B, \quad \to C$$

and we can produce C, B, A. There is an algorithm for this kind of problem in 7.4.3

Chomsky Normal Form

Definition: A CFG is in *Chomsky Normal Form* (CNF) iff all productions are of the form $A \to BC$ or $A \to a$

Theorem: *For any CFG G there is a CFG G' in Chomsky Normal Form such that $L(G') = L(G) - \{\epsilon\}$*

Chomsky Normal Form

We can assume that G has no ϵ- or unit productions. For each terminal a we introduce a new nonterminal A_a with the production

$$A_a \to a$$

We can then assume that all productions are of the form $A \to a$ or $A \to B_1 B_2 \ldots B_k$ with $k \geqslant 2$

If $k > 2$ we introduce C with productions $A \to B_1 C$ and $C \to B_2 \ldots B_k$ until we have only right-hand sides of length $\leqslant 2$

Chomsky Normal Form

Example: For the grammar

$$S \rightarrow\ aSb \mid SS \mid ab$$

we get first

$$S \rightarrow\ ASB \mid SS \mid AB \quad A \rightarrow a \quad B \rightarrow b$$

and then

$$S \rightarrow\ AC \mid SS \mid AB \quad A \rightarrow a \quad B \rightarrow b \quad C \rightarrow SB$$

which is in Chomsky Normal Form

The Chomsky Hierarchy

Noam Chomsky 1956

Four types of grammars

Type 0: no restrictions

Type 1: Context-sensitive, rules $\alpha A\beta \rightarrow \alpha\gamma\beta$

Type 2: Context-free or context-insensitive

Type 3: Regular, rules of the form $A \rightarrow Ba$ or $A \rightarrow aB$ or $A \rightarrow \epsilon$

Type 3 $\subseteq$ Type 2 $\subseteq$ Type 1 $\subseteq$ Type 0

Grammars for programming languages are usually Type 2

Context-Free Languages and Regular Languages

Theorem: *If L is regular then L is context-free.*

Proof: We know $L = L(A)$ for a DFA A. We have $A = (Q, \Sigma, \delta, q_0, F)$. We define a CFG $G = (Q, \Sigma, P, q_0)$ where P is the set of productions $q \rightarrow aq'$ if $\delta(q, a) = q'$ and $q \rightarrow \epsilon$ if $q \in F$. We have then $q \Rightarrow^* uq'$ iff $\hat{\delta}(q, u) = q'$ and $q \Rightarrow^* u$ iff $\hat{\delta}(q, u) \in F$. In particular $u \in L(G)$ iff $u \in L(A)$.

A grammar where all productions are of the form $A \rightarrow aB$ or $A \rightarrow \epsilon$ is called *left regular*

Pumping Lemma for Left Regular Languages

Let $G = (V, T, P, S)$ be a left regular language, and let N be $|V|$.

If $a_1 \ldots a_r$ is a string of length $\geq N$ any derivation

$$S \Rightarrow a_1 B_1 \Rightarrow a_1 a_2 B_2 \Rightarrow \cdots \Rightarrow a_1 \ldots a_i A$$

$$\Rightarrow \cdots \Rightarrow a_1 \ldots a_j A \Rightarrow \cdots \Rightarrow a_1 \ldots a_n$$

has length n and there is at least one variable A which is used twice (pigeon-hole principle)

If $x = a_1 \ldots a_i$ and $y = a_{i+1} \ldots a_j$ and $z = a_{j+1} \ldots a_n$ we have $|xy| \leqslant N$ and $xy^k z \in L(G)$ for all k

Pumping Lemma for Context-Free Languages

Let L be a context-free language

Theorem: *There exists* N *such that if* $z \in L$ *and* $N \leqslant |z|$ *then one can write* $z = uvwxy$ *such that*

$$z = uvwxy, \quad |vx| > 0, \quad |vwx| \leqslant N, \quad uv^k wx^k y \in L \text{ for all } k$$

Pumping Lemma for Context-Free Languages

Theorem: *The language* $\{a^k b^k c^k \mid k > 0\}$ *is* not *context-free*

Proof: Assume L to be context-free. Then we have N as stated in the Pumping Lemma. Consider $z = a^N b^N c^N$. We have $N \leqslant |z|$ so we can write $z = uvwxy$ such that

$$z = uvwxy, \quad |vx| > 0, \quad |vwx| \leqslant N, \quad uv^k wx^k y \in L \text{ for all } k$$

Since $|vwx| \leqslant N$ there is one letter $d \in \{a, b, c\}$ that occurs not in vwx, and since $|vx| > 0$ there is another letter $e \neq d$ that occurs in vx. Then e has more occurence than d in uv^2wx^2y, and this contradicts $uv^2wx^2y \in L$. Q.E.D.

Proof of the CFL Pumping Lemma

We can assume that the language is presented by a grammar in Chomsky Normal Form, working with $L - \{\epsilon\}$

The crucial remark is that a binary tree with height $p + 1$ has at most 2^p leaves

The *height* of a binary tree is the number of nodes from the root to the longest path

Proof of the CFL Pumping Lemma

Example: the Chomsky grammar

$$S \to AC \mid AB, \quad A \to a, \quad B \to b, \quad C \to SB$$

consider a parse tree for a^4b^4 corresponding to the derivation

$$S \Rightarrow AC \Rightarrow aC \Rightarrow aSB \Rightarrow aACB \Rightarrow aaCB \Rightarrow aaSBB$$

$$\Rightarrow a^2ACBB \Rightarrow a^3CBB \Rightarrow a^3SBBB \Rightarrow a^3ABBBB \Rightarrow a^4BBBB$$
$$\Rightarrow a^4BBBB \Rightarrow a^4bBBB \Rightarrow a^4b^2BB \Rightarrow a^4b^3B \Rightarrow a^4b^4$$

The symbol S appears twice on a path $u = aa,\ v = a,\ w = ab,\ x = b,\ y = bb$

Non closure under intersection

$T = \{a, b, c\}$

$L_1 = \{a^k b^k c^m \mid k, m > 0\}$

$L_2 = \{a^m b^k c^k \mid k, m > 0\}$

L_1 and L_2 are CFL, but the intersection

$L_1 \cap L_2 = \{a^k b^k c^k \mid k > 0\}$

is *not* CF

Non closure under intersection

However one can show (we will not do the proof in this course, but you should know the result)

Theorem: *If* $L_1 \subseteq \Sigma^*$ *is context-free and* $L_2 \subseteq \Sigma^*$ *is* regular *then* $L_1 \cap L_2$ *is context-free*

Application: The following language, for $\Sigma = \{0, 1\}$

$$L = \{uu \mid u \in \Sigma^*\}$$

is *not* context-free, by considering the intersection with $L(0^*1^*0^*1^*)$

One can show that the *complement* of L *is* context-free!

Closure under union

If $L_1 = L(G_1)$ and $L_2 = L(G_2)$ with disjoint set of variables V_1 and V_2, and same alphabet T, we can define

$G = (V_1 \cup V_2 \cup \{S\}, T, P_1 \cup P_2 \cup \{S \rightarrow S_1 \mid S_2\}, S)$

It is then direct to show that $L(G) = L(G_1) \cup L(G_2)$ since a derivation has the form

$$S \Rightarrow S_1 \Rightarrow^* u$$

or

$$S \Rightarrow S_2 \Rightarrow^* u$$

Non-Closure Under Complement

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

So CFL cannot be closed under complement in general. Otherwise they would be closed under intersection.

Closure Under Concatenation

If $L_1 = L(G_1)$ and $L_2 = L(G_2)$ with disjoint set of variables V_1 and V_2, and same alphabet T, we can define

$$G = (V_1 \cup V_2 \cup \{S\}, T, P_1 \cup P_2 \cup \{S \rightarrow S_1 S_2\}, S)$$

It is then direct to show that $L(G) = L(G_1)L(G_2)$ since a derivation has the form

$$S \Rightarrow S_1 S_2 \Rightarrow^* u_1 u_2$$

with

$$S_1 \Rightarrow^* u_1, \qquad S_2 \Rightarrow^* u_2$$

$LL(1)$ **parsing**

A grammar is $LL(1)$ if in a sequence of leftmost production we can decide what is the production looking only at the first symbol of the string to be parsed

For instance $S \rightarrow +SS \mid a \mid b$ is $LL(1)$

Any regular grammar $S \rightarrow aA \quad , \; A \rightarrow bA \mid \epsilon$ is $LL(1)$ iff it corresponds to a *deterministic* FA

There are algorithms to decide if a grammar is $LL(1)$ (not done in this course)

Any $LL(1)$ grammar is unambiguous (because by definition there is a at most one left most derivation for any string)

Grammar transformations

The grammar

$S \rightarrow AB, \quad A \rightarrow aA \mid a, \quad b \rightarrow bB \mid c$

is equivalent to the grammar

$S \rightarrow aAB, \quad A \rightarrow aA \mid \epsilon, \quad b \rightarrow bB \mid c$

Grammar transformations

The grammar

$S \to Bb \qquad B \to Sa \mid a$

which is not $LL(1)$ is equivalent to the grammar

$S \to abT \quad T \to abT \mid \epsilon$

which is $LL(1)$

Grammar transformations

The grammar

$A \rightarrow Aa \mid b$

is equivalent to the grammar

$A \rightarrow bB, \quad B \rightarrow aB \mid \epsilon$

In general however there is *no* algorithm to decide $L(G_1) = L(G_2)$

For regular expression, we have an algorithm to decide $L(E_1) = L(E_2)$

The CYK Algorithm

We present now an algorithm to decide if $w \in L(G)$, assuming G to be in Chomsky Normal Form.

This is an example of the technique of *dynamic programming*

Let n be $|w|$. The natural algorithm (trying all productions of length $< 2n$) may be exponential. This technique gives a $O(n^3)$ algorithm!!

dynamic programming

$fib\ 0 = fib\ 1 = 1$

$fib\ (n+2) = fib\ n + fib\ (n+1)$

$fib\ 5?$ calls $fib\ 4,\ fib\ 3$ and $fib\ 4$ calls $fib\ 3$

So in a top-down computation there is duplication of works (if one does not use memoization)

dynamic programming

For a bottom-up computation

$fib\ 2 = 2,\ fib\ 3 = 3,\ fib\ 4 = 5,\ fib\ 5 = 8$

What is going on in the CYK algorithm or Earley algorithm is similar

$S \rightarrow AB \mid BC, \quad A \rightarrow BA \mid a, \quad B \rightarrow CC \mid b, \quad C \rightarrow AB \mid a$

$bab \in L(G)??$ and $aba \in L(G)?$

dynamic programming

The idea is to represent bab as the collection of the facts $b(0,1),\ a(1,2),\ b(2,3)$

We compute then the facts $X(i,k)$ for $i < k$ by induction on $k-i$

Only one rule:

If we have a production $C \rightarrow AB$ and A in $X(i,j)$ and B in $X(j,k)$ then C is in $X(i,k)$

The CYK Algorithm

The algorithm is best understood in term of production systems

Example: the grammar

$$S \to AB \mid BA \mid SS \mid AC \mid BD$$

$$A \to a, \quad B \to b, \quad C \to SB, \quad D \to SA$$

becomes the production system

The CYK Algorithm

$$A(x,y), B(y,z) \rightarrow S(x,z), \qquad B(x,y), A(y,z) \rightarrow S(x,z)$$
$$S(x,y), S(y,z) \rightarrow S(x,z), \qquad A(x,y), C(y,z) \rightarrow S(x,z)$$
$$B(x,y), D(y,z) \rightarrow S(x,z), \qquad S(x,y), B(y,z) \rightarrow C(x,z)$$
$$S(x,y), A(y,z) \rightarrow D(x,z), \qquad a(x,y) \rightarrow A(x,y), \quad b(x,y) \rightarrow B(x,y)$$

The CYK Algorithm

The problem if one can one derive $S \Rightarrow^* aabbab$ is transformed to the problem: can one produce $S(0, 6)$ in this production system given the facts

$$a(0, 1), a(1, 2), b(2, 3), b(3, 4), a(4, 5), b(5, 6)$$

The CYK Algorithm

For this we apply a forward chaining/bottom up sequence of productions

$$A(0,1), A(1,2), B(2,3), B(3,4), A(4,5), B(5,6)$$

$$S(1,3), S(3,5), S(4,6)$$
$$S(1,5), C(1,4), C(3,6)$$
$$S(0,4), \ldots$$
$$S(0,6)$$

The CYK Algorithm

For instance the fact that $C(3,6)$ is produced corresponds to the derivation

$$C \Rightarrow SB \Rightarrow BAB \Rightarrow bAB \Rightarrow baB \Rightarrow bab$$

In this way, we get a solution in $O(n^3)$!

Forward-chaining inference

This idea works actually for any grammar. For instance

$$S \to SS \mid aSb \mid \epsilon$$

is represented by the production system

$$\to S(x,x), \qquad S(x,y), S(y,z) \to S(x,z)$$

$$a(x,y), S(y,z), b(z,t) \to S(x,t)$$

and the problem to decide $S \Rightarrow^* aabb$ is replaced by the problem to derive $S(0,4)$ from the facts

$$a(0,1), a(1,2), b(2,3), b(3,4)$$

Forward-chaining inference

This is the main idea behind *Earley algorithm*

Mainly used for parsing in computational linguistics

Earley parsers are interesting because they can parse all context-free languages

Complement of a CLF

We have seen that CLF are not closed under intersection, are closed under union

It follows that they are not closed under complement

Here is an explicit example: one can show that the complement of

$\{a^n b^n c^n \mid n \geqslant 0\}$

is a CFL

Undecidable Problems

We have given algorithm to decide $L(G) \neq \emptyset$ and $w \in L(G)$. What is surprising is that it can be *shown* that there are no algorithms for the following problems

Given G_1 and G_2 do we have $L(G_1) \subseteq L(G_2)$? Do we have $L(G_1) = L(G_2)$? Given G and R regular expression, do we have $L(G) = L(R)$? $L(R) \subseteq L(G)$? Do we have $L(G) = T^*$ where T is the alphabet of G? (Compare to the case of regular languages)

Given G is G ambiguous??

Undecidable Problems

One reduces these problems to the Post Correspondance Problem

Given $u_1, \ldots, u_n$ and $v_1, \ldots, v_n$ in $\{0,1\}^*$ is it possible to find $i_1, \ldots, i_k$ such that

$u_{i_1} \ldots u_{i_k} = v_{i_1} \ldots v_{i_k}$

Example: $1,\ 10,\ 011$ and $101,\ 00,\ 11$

Challenge example: $001,\ 01,\ 01,\ 10$ and $0,\ 011,\ 101,\ 001$

Haskell Program

```
isPrefix [] ys = True
isPrefix (x:xs) (y:ys) = x == y && isPrefix xs ys
isPrefix xs ys = False

isComp (xs,ys) = isPrefix xs ys || isPrefix ys xs

exists p [] = False
exists p (x:xs) = p x || exists p xs

exhibit p (x:xs) = if p x then x else exhibit p xs
```

Haskell Program

```
addNum k [] = []
addNum k (x:xs) = (k,x):(addNum (k+1) xs)

nextStep xs ys =
 concat (map (\ (n,(s,t)) ->
              map (\ (ns,(u,v)) -> (ns++[n],(u ++ s,v ++ t)))
                   ys)
              xs)
```

Haskell Program

```
mainLoop xs ys =
 let
  bs = filter (isComp . snd) ys
  prop (_,(u,v)) = u == v
 in
  if exists prop bs then exhibit prop bs
   else if bs == [] then error"NO SOLUTION"
          else mainLoop xs (nextStep xs bs)
```

Haskell Program

```
post xs =
 let
  as = addNum 1 xs
 in mainLoop as (map (\ (n,z) -> ([n],z)) as)

xs1 = [("1","101"),("10","00"),("011","11")]

xs2 = [("001","0"),("01","011"),("01","101"),("10","001")]
```

Haskell Program

```
Main> post xs1
([1,3,2,3],("101110011","101110011"))

Main> post xs2

ERROR - Garbage collection fails to reclaim sufficient space
[2,2,2,3,2,2,2,3,3,4,4,6,8,8,15,
 21,15,17,18,24,15,12,12,18,18,24,24,45,
 63,66,84,91,140,182,201,346,418,324,330,321,423,459,780
```

Post Correspondance Problem and CFL

To the sequence $u_1, \ldots, u_n$ we associate the following grammar G_A

The alphabet is $\{0, 1, a_1, \ldots, a_n\}$

The productions are

$A \rightarrow u_1 a_1 \mid \ \ldots \ \mid u_n a_n \mid u_1 A a_1 \mid \ \ldots \ \mid u_n A a_n$

This grammar is non ambiguous

Post Correspondance Problem and CFL

To the sequence $v_1, \ldots, v_n$ we associate the following grammar G_B

The alphabet is the same $\{0, 1, a_1, \ldots, a_n\}$

The productions are

$B \rightarrow v_1 a_1 \mid \ \ldots \ \mid v_n a_n \mid v_1 B a_1 \mid \ \ldots \ \mid v_n B a_n$

This grammar is non ambiguous

Post Correspondance Problem and CFL

Theorem: *We have* $L(G_A) \cap L(G_B) \neq \emptyset$ *iff the Post Correspondance Problem for* $u_1, \ \ldots, \ u_n$ *and* $v_1, \ \ldots, \ v_n$ *has a solution*

Post Correspondance Problem and CFL

Finally we have the grammar G with productions

$S \rightarrow A \mid B$

Theorem: *The grammar G is ambiguous iff the Post Correspondance Problem for $u_1, \ \ldots, \ u_n$ and $v_1, \ \ldots, \ v_n$ has a solution*

Post Correspondance Problem and CFL

The complement of $L(G_A)$ is CF

We see this on one example $u_1 = 0,\ u_2 = 10$

The complement of $L(G_B)$ is CF

Hence we have a grammar G_C for the union of the complement of $L(G_A)$ and the complement of $L(G_B)$

Post Correspondance Problem and CFL

Theorem: *We have* $L(G_C) = T^*$ *iff* $L(G_A) \cap L(G_B) = \emptyset$

Hence the problems

$L(E) = L(G)$

$L(E) \subseteq L(G)$

are in general undecidable