Formal Languages

Context free languages provide a convenient notation for recursive description of languages.

The original goal of CFL was to formalize the structure of natural languages. This goal is still elusive, but CFGs are now the universally accepted formalism for definition of (the syntax of) *programming* languages.

Writing parsers has become an almost fully automated process thanks to this theory.

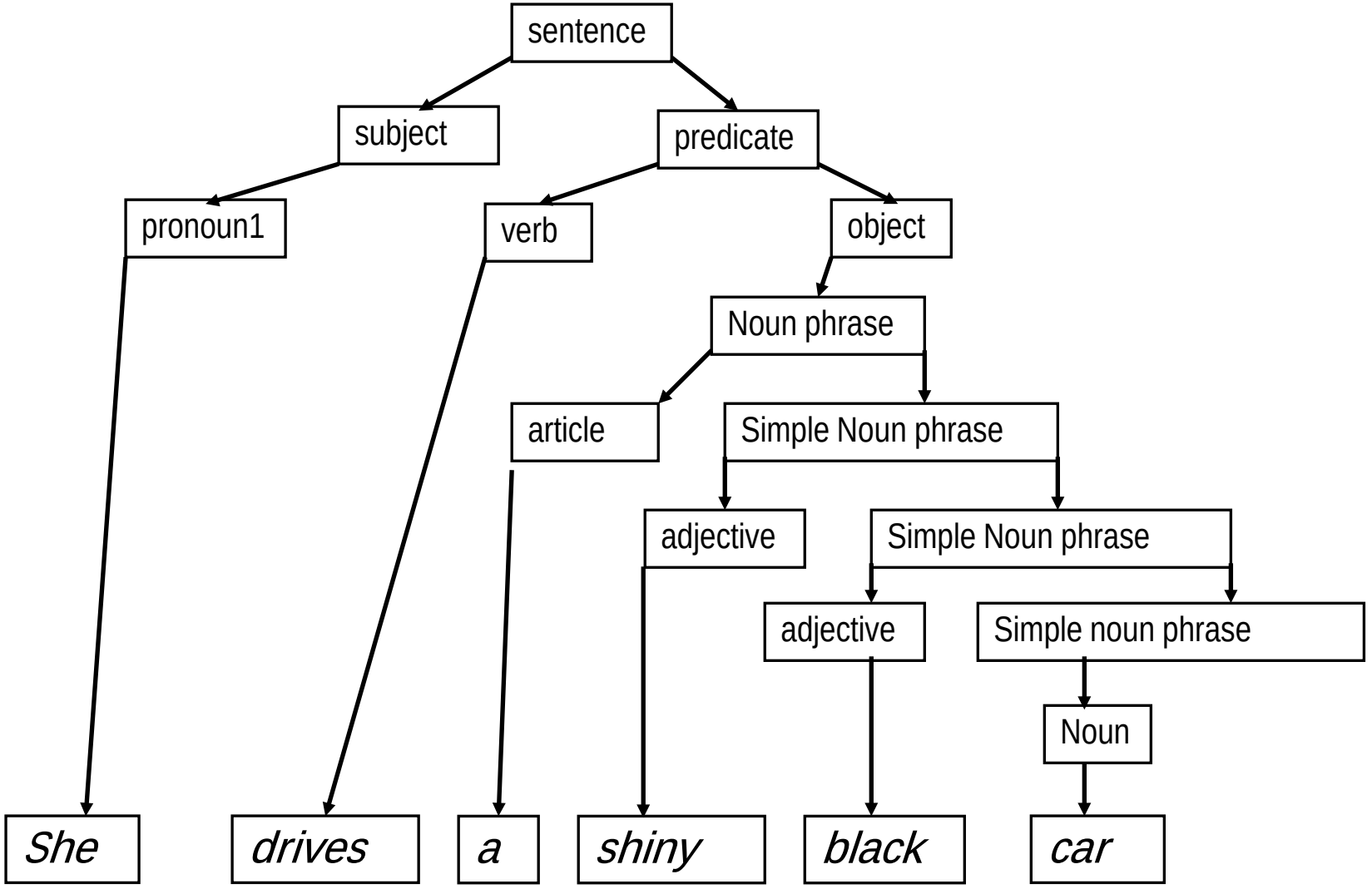
A Simple Grammar for English

Example taken from Floyd & Beigel.

<Sentence>	→	<Subject> <Predicate>
<Subject>	→	<Pronoun1> | <Pronoun2>
<Pronoun1>	→	**I** | **we** | **you** | **he** | **she** | **it** | **they**
<Noun Phrase>	→	<Simple Noun Phrase> | <Article> <Noun Phrase>
<Article>	→	**a** | **an** | **the**
<Predicate>	→	<Noun> | <Adjective> <Simple Noun Phrase>
<Simple Noun Phrase>	→	<Verb> | <Verb> <Object>
<Object>	→	<Pronoun2> | <Noun Phrase>
<Pronoun2>	→	**me** | **us** | **you** | **him** | **her** | **it** | **them**
<Noun>	→	. . .
<Verb>	→	. . .

Example

Derive the sentence "*She drives a shiny black car*" from these rules.



<sentence> ⇒
<subject> <predicate> ⇒
<pronoun> <predicate> ⇒
She <predicate> ⇒
She <verb> <object> ⇒
She drives <object> ⇒
She drives <simple noun phrase> ⇒
She drives <article> <noun phrase> ⇒
She drives a <noun phrase> ⇒
She drives a <adjective> <noun phrase> ⇒
She drives a shiny <noun phrase> ⇒
She drives a shiny <adjective> <simple noun phrase> ⇒
She drives a shiny black <simple noun phrase> ⇒
She drives a shiny black <noun> ⇒
She drives a shiny black car

Definition of Context-Free-Grammars

A CFG is a quadruple G= (V,T,P,S), where

- V is a finite set of *variables* (*nonterminals*, *syntactic categories*)
- T is a finite set of *terminals*
- P is a finite set of *productions* -- rules of the form $X \longrightarrow a$, where $X \in V$ and $a \in (V \cup T)^*$
- S, the *start symbol*, is an element of V

Vertical bar (|), as used in the examples on the previous slides, is used to denote a set of several productions (with the same *lhs*).

Example

<Expression>	→	<Term> | <Expression> **+** <Term>
<Term>	→	<Factor> | <Term> ***** <Factor>
<Factor>	→	<Identifer> | **(** <Expression> **)**
<Identifier>	→	**x** | **y** | **z** | ...

V = {<Expression>, <Term>, <Factor>, <Identifier>}

T = {+, *, (,), x, y, z, ...}

P = {

<Expression> → <Term>
<Expression> → <Expression> **+** <Term>
<Term> → <Factor>
<Term> → <Term> ***** <Factor>
<Factor> → <Identifer>
<Factor> → **(** <Expression> **)**
<Identifier> → **x**
<Identifier> → **y**
<Identifier> → **z**
<Identifier> → ...

}

S = <Expression>

Exercise in deriving Expressions

<Expression>	→	<Term> | <Expression> **+** <Term>
<Term>	→	<Factor> | <Term> ***** <Factor>
<Factor>	→	<Identifer> | **(** <Expression> **)**
<Identifier>	→	**x** | **y** | **z** | ...

In class exercise: Derive

- x + (y * 3)
- x + z * w + q

Notational Conventions

a, b, c, … (lower case, beginning of alphabet) are concrete terminals;

u, v, w, x, y, z (lower case, end of alphabet) are for strings of terminals

$\alpha, \beta, \gamma, \ldots$ (Greek letters) are for strings over $(T \cup V)$ (*sentential forms*)

A, B, C, … (capitals, beginning of alphabet) are for variables (for non-terminals).

X, Y, Z are for variables standing for terminals.

Short-hand

Note. We often abbreviate a context free grammar, such as:

$G_2 =$ (V={S},
T={(,)},
P={ $S \to \varepsilon$, $S \to SS$, $S \to (S)$ },
S=S}

By giving just its productions

$S \to \varepsilon$ | SS |(S)

And by using the following conventions.

1) The start symbol is the lhs of the first production.
2) Multiple production for the same lhs non-terminal can be grouped together by using vertical bar (|)
3) Non-terminals are capitalized.
4) Terminal-symbols are lower case or non-alphabetic.

Derivations

The single-step derivation relation $\Rightarrow$ on $(V \cup T)^*$ is defined by:

$\alpha \Rightarrow \beta$ iff β is obtained from α by replacing an occurrence of the lhs of a production with its rhs. That is, $\alpha'A\alpha'' \Rightarrow \alpha'\gamma\alpha''$ is true iff $A \rightarrow \gamma$ is a production.

We write $\alpha \Rightarrow^* \beta$ when β can be obtained from α through a sequence of several (possibly zero) derivation steps.

The *language of the CFG* , G, is the set

$L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$ (where S is the start symbol of G)

Context-free languages are languages of the form L(G)

Example 1

The familiar non-regular language

$$L = \{ a^k b^k \mid k \geq 0 \}$$

is context-free.

The grammar G_1 for it is given by T={a,b}, V={S}, and productions:

1. $S \rightarrow \varepsilon$
2. $S \rightarrow a\ S\ b$

Here is a derivation showing $a^3b^3 \in L(G)$:

$S \Rightarrow_2 aSb \Rightarrow_2 aaSbb \Rightarrow_2 aaaSbbb \Rightarrow_1 aaabbb$

(Note: we sometimes label the arrow with a subscript which tells the production used to enable the transformation)

Example 1 continued

Note, however, that the fact $L=L(G_1)$ is not totally obvious. We need to prove set inclusion both ways.

To prove $L \subseteq L(G_1)$ we must show that there exists a derivation for every string $a^k b^k$; this is done by induction on k.

For the converse, $L(G_1) \subseteq L$, we need to show that if $S \Rightarrow^* w$ and $w \in T^*$, then $w \in L$. This is done by induction on the length of derivation of w.

Example 2

The language of balanced parentheses is context-free. It is generated by the following grammar:

$G_2 = (V=\{S\},$

$T=\{(,)\},$

$P=\{ S \rightarrow \varepsilon \mid SS \mid (S)\},$

$S=S\}$

Example 3

Consider the grammar:

$S \rightarrow AS \mid \varepsilon$

$A \rightarrow 0A1 \mid A1 \mid 01$

The derivation:

$S \Rightarrow AS \Rightarrow A1S \Rightarrow 011S \Rightarrow 011AS \Rightarrow$
$0110A1S \Rightarrow 0110011S \Rightarrow 0110011$

shows that $0110011 \in L(G_3)$.

Example 3 notes

The language $L(G_3)$ consists of strings $w \in \{0,1\}^*$ such that:

P(w): Either $w=\varepsilon$, or w begins with 0, and every block of 0's in w is followed by at least as many 1's

Again, the proof that G_3 generates all and only strings that satisfy P(w) is not obvious. It requires a two-part inductive proof.

Leftmost and Rightmost Derivations

The same string w usually has many possible derivations $S \equiv a_0 \Rightarrow a_1 \Rightarrow a_2 \Rightarrow \ldots \Rightarrow a_n \equiv w$

We call a derivation *leftmost* if in every step $a_i \Rightarrow a_{i+1}$, it is the first (leftmost) variable in $a_i\$$ that is being replaced with the rhs of a production. Similarly, in a *rightmost* derivation, it is always the last variable that gets replaced.

The above derivation of the string 0110011 in the grammar G_3 is leftmost. Here is a rightmost derivation of the same string:

$S \Rightarrow A\underline{S} \Rightarrow AA\underline{S} \Rightarrow A\underline{A} \Rightarrow A0\underline{A}1 \Rightarrow \underline{A}0011 \Rightarrow \underline{A}10011 \Rightarrow 0110011$

$S \rightarrow AS \mid \varepsilon$

$A \rightarrow 0A1 \mid A1 \mid 01$

Facts

Every Regular Language is also a Context Free Language

How might we prove this?

- Choose one of the many specifications for regular languages
- Show that every instance of that kind of specification has a total mapping into a Context Free Grammar

What is an appropriate choice?

In Class Exercise

Map the Regular Expressions into a Context Free language.

1. $L(\emptyset)=\emptyset$ and $L(\varepsilon)=\{\text{“”}\}$
2. $L(a)=\{\text{‘a’}\}$
3. Inductive cases
   1. $L(E^*) = (L(E))^*$
   2. $L(EF) = L(E)\ L(F)$ recall implicit use of dot $L(E) \cdot L(F)$
   3. $L(E+F) = L(E) \cup L(F)$

Designing CFGs

Break the language into simpler (disjoint) parts with Grammars A B C . Then put them together $S \rightarrow Start_A \mid Start_B \mid Start_C$

If a Language fragment is Regular construct a DFA or RE, use these to guide you.

Infinite languages use rules like

$R \rightarrow a\ R$

$R \rightarrow R\ b$

Languages with linked parts use rules like

$R \rightarrow B\ x\ B$

$R \rightarrow x\ R\ x$

Find CFG for these languages

$\{a^n\, b\, a^n \mid n \in Nat\}$

$\{ w \mid w \in \{a,b\}^*,$ and w is a palindrome of even length$\}$

$\{a^n\, b^k \mid n,k \in Nat,\ n \leq k\}$

$\{a^n\, b^k \mid n,k \in Nat,\ n \geq k\}$

$\{ w \mid w \in \{a,b\}^*,$ w has equal number of a's and b's $\}$

Ambiguity

A grammar is ambiguous if one of its strings has 2 or more leftmost (or rightmost) derivations.

- Consider the grammar
  1. E -> E + E
  2. E -> E * E
  3. E -> x | y
- And the string: x + x * y

$E \Rightarrow_1 E + E \Rightarrow_3 x + E \Rightarrow_2 x + E * E \Rightarrow_3 x + x * E \Rightarrow_3 x + x * y$

$E \Rightarrow_2 E * E \Rightarrow_1 E + E * E \Rightarrow_3 x + E * E \Rightarrow_3 x + x * E \Rightarrow_3 x + x * y$

Note we use the productions in a different order.

Ambiguity

Note that ambiguity is a property of the grammar

It is not a property of the set of strings derived by that grammar.

The same set of strings, can be derived by two different grammars, one which is ambiguous and one that is not!

Common Grammars with ambiguity

Expression grammars with infix operators with different precedence levels.

Nested if-then-else statements

```
st  ->  if exp  then st  else st
        |  if exp then st
        | id := exp
```

if x=2 then if x=3 then y:=2 else y := 4

if x=2 then (if x=3 then y:=2) else y := 4
if x=2 then (if x=3 then y:=2 else y := 4)

Removing ambiguity.

Adding levels to a grammar

```
E -> E + E | E * E | id | ( E )
```

Transform to an equivalent grammar

```
E  ->  E + T | T
T  ->  T * F | F
F  -> id | ( E )
```

Levels make formal the notion of precedence.
Operators that bind “tightly” are on the lowest levels

Length of Derivations

Consider:

S -> A B

A -> 1

B -> B B | ε

What is L(G)?

How many ways can you derive 1?

If you know S =>* α =>* 1 is there a bound on $|\alpha|$?

Can we do better? Can we restrict grammars to those where the size of the derivation is bounded by some function of the size of the string?

Chomsky Normal Form

There are many CFG's for any given CFL. When reasoning about CFL's, it often helps to assume that a grammar for it has some particularly simple form.

Here are some ideas how CFG's can be simplified.

Useless Symbols

A *useful* symbol (terminal or variable) X must be

1. *generating*: $X \Rightarrow^* w$ for some $w \in T^*$ (I.e. w is all terminal symbols)
2. *reachable from S*: $S \Rightarrow^* \alpha X \beta$ for some $\alpha, \beta \in (V \cup T)^*$

An algorithm for elimination of useless symbols first eliminates non-generating ones, then eliminates those not reachable from S.

The order is important, because, for example, when $S \Rightarrow^* \alpha X \beta$ and α contains a non-generating symbol, then X is reachable, but will become unreachable after elimination of non-generators.

Algorithm: Part 1

We describe the algorithm on an example grammar:

$S \rightarrow AB \mid C$
$A \rightarrow 0B \mid C$
$B \rightarrow 1 \mid A0$
$C \rightarrow AC \mid C1$

1. Elimination of non-generators

0 and 1 are in. (because 0 and 1 are terminal)
$B \rightarrow 1$, says B is in.
$A \rightarrow 0B$, says A is in.
$S \rightarrow AB$, says S is in.
Nothing more can be added.

Thus, C can be eliminated, along with any productions containing it. The result is this grammar:

$S \rightarrow AB$
$A \rightarrow 0B$
$B \rightarrow 1 \mid A0$

Algorithm: Part 2

2. Elimination of non-reachables

S is in. (since it is the start symbol)

A and B are in.

0 and 1 are in.

Nothing more can be added.

There is nothing left to eliminate.

$S \rightarrow AB$
$A \rightarrow 0B$
$B \rightarrow 1 \mid A0$

In this case, the end result is the same grammar we used as input to this part of algorithm.

Λ-Productions

A variable A is *nullable* if $A \Rightarrow^* \Lambda$. We can modify a given grammar G and obtain a grammar G' in which there are no nullable variables and which satisfies $L(G') = L(G) - \{\Lambda\}$.

Find nullable symbols iteratively, using these facts:

1. If $A \rightarrow \Lambda$ is a production, then A is nullable.
2. If $A \rightarrow B_1B_2 \ldots B_k$ is a production and $B_1, B_2, \ldots, B_k$ are all nullable, then A is nullable.

Once nullable symbols are known, we get G' as follows:

1. For every production $A \to \alpha$, add new productions $A \to \alpha'$, where α' is obtained by deleting some (or all) nullable symbols from α.
2. Remove all productions $A \to \Lambda$

Example. If G contains a production $A \to BC$ and both B and C are nullable, then we add

$A \to B \mid C$

to G'.

Unit Productions

These are of the form $A \to B$, where A,B are variables.

Assuming the grammar has no Λ–productions, we can eliminate unit productions as follows.

1. Find all pairs of variables such that $A \Rightarrow^* B$. (This happens iff B can be obtained from A by a chain of unit productions.)
2. Add new production $A \to \alpha$ whenever $A \Rightarrow^* B \Rightarrow \alpha$.
3. Remove all unit productions.

Chomsky Normal Form defined

A grammar is in *Chomsky normal form* (CNF) if it has no useless symbols and all its productions have one of these two forms:

1. $A \rightarrow BC$, where B,C are variables
2. $A \rightarrow a$, where a is a terminal

Theorem. For every CFG G, there exists a CFG G' in CNF such that L(G')=L(G) - {ε}

The first three steps of getting G' are elimination of Λ-productions, elimination of unit productions, and elimination of useless symbols (in that order). There remain two steps:

1. Arrange that all productions are of the form A $\rightarrow \alpha$, where α is a terminal, or contains only variables.
2. Break up every production A $\rightarrow \alpha$ with $|\alpha| > 2$ into productions whose rhs has length two.

For the first part, introduce a new variable C for each terminal c that occurs in the rhs of some production, add the production $C \to c$ (unless such a production already exists), and replace c with C in all other productions.

For example, the production $A \to 0B1$ would be replaced with $A_0 \to 0$, $A_1 \to 1$, $A \to A_0BA_1$.

An example explains the second part. The production $A \to BCDE$ is replaced by three others,

1. $A \to BA_1$,
2. $A_1 \to CA_2$,
3. $A_2 \to DE$,

using two new variables A_1, A_2.

Example

To bring the grammar: $S \to SS \mid (S) \mid \Lambda$

into CNF, we first eliminate the only Λ-production and get

$S \to SS \mid (S) \mid ()$

There are no unit productions and no useless symbols. We need to introduce new variables for both terminals, so we get the grammar

$S \to SS \mid LSR \mid LR$

$L \to ($

$R \to)$

Finally, we need to take care of the (only) long production $S \to LSR$, and the result is

$S \to SS \mid LA \mid LR$

$L \to ($

$R \to)$

$A \to SR$