Context-free grammars and languages

Motivation

- It turns out that many important languages, e.g. programming languages, cannot be described by finite automata/regular expressions.
- To describe such languages, we shall introduce **context-free grammars** (CFG's).

Idea behind CFG's

To generate strings by beginning with a **start symbol** S and then apply rules that describe how certain symbols may be replaced with other strings.

Context-free grammar: definition

Definition. A **context-free grammar** $G = (N, \Sigma, P, S)$ consists of

- a finite set N of **non-terminal symbols**;
- a finite set Σ of **terminal symbols** not in N;
- a finite set P of **production rules** of the form

$$u \longrightarrow v$$

  where u is in N and v is a string in $(\Sigma \cup N)^*$;
- a **start symbol** S in N.

Example

The grammar G with non-terminal symbols $N = \{S\}$, terminal symbols $\Sigma = \{a, b\}$, and productions

$$S \to aSb$$
$$S \to ba.$$

Following a common practice, we use capital letters for non-terminal symbols and small letters for terminal symbols.

Example: integer expressions

A simple form of integer expressions.

- The non-terminal symbols are

symbol	intended meaning
E	expressions, e.g. $(x + y) * 3$
I	identifiers, e.g. x, y
C	constants (here: natural numbers).

- The alphabet Σ of terminal symbols is $\{x, y, 0, 1, \ldots, 9, *, +, (,)\}$.

Productions for integer expressions

$$
\begin{array}{lll}
E \longrightarrow C & C \longrightarrow 0 & C \longrightarrow C0 \\
E \longrightarrow I & C \longrightarrow 1 & C \longrightarrow C1 \\
E \longrightarrow E + E & \vdots & \vdots \\
E \longrightarrow E * E & C \longrightarrow 9 & C \longrightarrow C9 \\
E \longrightarrow (E) & &
\end{array}
$$

$$
\begin{array}{l}
I \longrightarrow x \\
I \longrightarrow y
\end{array}
$$

Language of a formal grammar

Definition. The language of a formal grammar $G = (N, \Sigma, P, S)$, denoted as $L(G)$, is defined as all those strings over Σ that can be generated by beginning with the start symbol S and then applying the productions P.

Parse trees

- Every string w in a context-free language has a **parse tree**.
- The root of a parse tree is the start symbol S.
- The leaves of a parse tree are the terminal symbols that make up the string w.
- The branches of the parse tree describe how the productions are applied.

Exercise

Consider the grammar

$$S \rightarrow A1B$$

$$A \rightarrow 0A \mid \epsilon$$

$$B \rightarrow 0B \mid 1B \mid \epsilon.$$

Give parse trees for the strings

1. 00101
2. 1001
3. 00011

Exercise

For the CFG G defined by the productions

$$S \rightarrow aS \mid Sb \mid a \mid b,$$

prove by induction on the size of the parse tree that the no string in language $L(G)$ has ba as a substring.

Exercises

Give a context-free grammar for the language of all palindromes over the alphabet $\{a, b, c\}$. (A **palindrome** is a word that is equal to the reversed version of itself, e.g. “abba”, “bab”.)

Parsing

- Finding a parse tree for a string is called **parsing**. Software tools that do this are called **parsers**.
- A compiler must parse every program before producing the executable code.
- There are tools called “parser generators” that turn CFG’s into parsers. One of them is the famous YACC (“Yet Another Compiler Compiler”).
- Parsing is a science unto itself, and described in detail in courses about compilers.

Ambiguity

- A context-free grammar with more than one parse tree for some expression is called **ambiguous**.
- Ambiguity is dangerous, because it can affect the meaning of expressions; e.g. in the expression $b * a + b$, it matters whether $*$ has precedence over $+$ or vice versa.
- To address this problem, parser generators (like YACC) allow the language designer to specify the operator precedence.

Exercise

Consider the grammar

$$S \to aS \mid aSbS \mid \epsilon.$$

Show that this grammar is ambiguous.

Exercise

Consider the grammar

$$S \rightarrow A1B$$
$$A \rightarrow 0A \mid \epsilon$$
$$B \rightarrow 0B \mid 1B \mid \epsilon.$$

Show that this grammar is unambiguous.

Compact notation

More compact notation for productions:

$$E \rightarrow I \mid C \mid E + E \mid E * E \mid (E)$$

$$C \rightarrow 0 \mid C0 \mid \ldots \mid 9 \mid C9$$

$$I \rightarrow x \mid y$$

Regular grammars

Next, we shall certain CFG's called **regular grammars** and show that they define the same languages as finite automata and regular expressions.

Regular grammars: definition

Definition. A CFG is called **regular** if every production has one of the three forms below

$$A \longrightarrow aB$$
$$A \longrightarrow \varepsilon$$

where A and B are terminal symbols and a is a non-terminal symbol.

Non-example

The grammar below is not regular because of the b on the right of S.

$$S \longrightarrow aSb$$
$$S \longrightarrow \epsilon.$$

From NFA to regular grammar

For an NFA over alphabet Σ, we construct a regular grammar G over alphabet Σ as follows:

1. The non-terminal symbols are the states of the NFA.
2. The start symbol is the initial state of the NFA.
3. For every transition $q \xrightarrow{a} q'$ in the NFA, introduce a production $q \rightarrow aq'$.
4. For every final state q, add a production $q \rightarrow \varepsilon$.

Exercise

Turn some of the NFA's in this lecture into regular grammars.

From reg. grammar to reg. expression

We I shall explain this in a number of steps.

First, we write the grammar in its compact notation, e.g.

$$X \to aY \mid bZ \mid \epsilon \tag{1}$$

$$Y \to cX \mid dZ \tag{2}$$

$$Z \to eZ \mid fY \tag{3}$$.

From reg. grammar to reg. expression

Next, we replace $\rightarrow$ by $=$ and $|$ by $+$, to make the grammar look like an equation system:

$$X = aY + bZ + \epsilon \quad (1)$$

$$Y = cX + dZ \quad (2)$$

$$Z = eZ + fY \quad (3).$$

The trick is now to get a solution for the start symbol X **that does not rely on** Y **and** Z.

From reg. grammar to reg. expression

$$X = aY + bZ + \epsilon \tag{1}$$
$$Y = cX + dZ \tag{2}$$
$$Z = eZ + fY \tag{3}.$$

We shall reduce the three equations above to two equations as follows:

1. Find a solution for Z in terms of only X and Y.
2. Replace that solution for Z in equations (1) and (2). (That yields equations for X and Y that don't rely on Z anymore.)

From reg. grammar to reg. expression

How do we find a solution for

$$Z = eZ + fY \tag{3}$$

that doesn't rely on Z anymore? The idea is

> "Z can produce an e and become Z again for a number of times, but finally Z must become fY."

Formally, the solution of Equation (3) is

$$Z = e^* fY.$$

From reg. grammar to reg. expression

Generally, the solution of any equation of the form

$$A = cA + B$$

is

$$A = c^* B.$$

From reg. grammar to reg. expression

Next, we replace the solution

$$Z = e^* f Y$$

in equations (1) and (2)

$$X = aY + bZ + \epsilon \qquad (1)$$

$$Y = cX + dZ \qquad (2).$$

This yields

$$X = aY + be^* fY + \epsilon \qquad (1')$$

$$Y = cX + de^* fY \qquad (2').$$

From reg. grammar to reg. expression

Simplifying (1') by using the **distributivity law** yields

$$X = (a + be^*f)Y + \epsilon \qquad (1')$$

Now instead of three equations over X, Y, Z, we have only two equations over X, Y:

$$X = (a + be^*f)Y + \epsilon \qquad (1')$$
$$Y = cX + de^*fY \qquad (2').$$

From reg. grammar to reg. expression

$$X = (a + be^* f)Y + \epsilon \qquad (1')$$
$$Y = cX + de^* fY \qquad (2').$$

Next, we repeat our game of eliminating non-terminals and find a solution for Y:

$$Y = (de^* f)^* cX.$$

From reg. grammar to reg. expression

Using

$$Y = (de^*f)^*cX.$$

in (1’) yields.

$$X = (a + be^*f)(de^*f)^*cX + \epsilon \qquad (1'')$$

The solution of (1”) is

$$X = ((a + be^*f)(de^*f)^*c)^*\epsilon = ((a + be^*f)(de^*f)^*c)^*$$

So we found a regular expression for the language generated by X.

Exercise

Consider the NFA given by the transition table below.

	0	1
$\rightarrow X$	$\{X\}$	$\{X, Y\}$
$*Y$	$\{Y\}$	$\{Y\}$

1. Give a regular grammar for it.
2. Calculate a regular expression that describes the language.

Exercise

Repeat the exercise for the NFA below.

	0	1
$\rightarrow *X$	$\{X\}$	$\{Y\}$
Y	$\{Y\}$	$\{Z\}$
Z	$\{Z\}$	$\{X\}$

Also, describe the accepted language in English.

The big picture: final version

See lecture for diagram.

Definition. Languages that are definable by regular expressions or regular grammars or finite automata) are called **regular languages**.

A non-regular language

Proposition. The language

$$L = \{a^n b^n \mid n \geq 1\}$$

(for which we gave a CFG earlier) is not regular.

Proof. By contradiction. Suppose that L is regular. Then there is a DFA A that accepts L. Let n be the number of states of A. After reading the string a^n, the DFA must be in some state that it already reached for a^m for some $m < n$. So if A accepts $a^n b^n$, then it also accepts $a^m b^n$. But this contradicts the definition

Significance of the example

- That L is not regular has great practical significance:
- Recall that a string $a^n b^n$ can be seen as n opening brackets followed by n closing brackets.
- Realistic programming languages contain balanced brackets.
- So it can be shown that realistic programming languages are are not regular!