CS112: Theory of Computation (LFA)

Lecture8: Context-free Grammars

Dumitru Bogdan

Faculty of Computer Science
University of Bucharest

April 6, 2022

Table of contents

Section 1

Previously on CS112

Nonregular Languages

- Fine automata proved to be quite powerful for such simple model
- However, they are limited in the sense that **there are languages not recognized by any finite automaton**
- For example $B = \{0^n 1^n | n \geq 0\}$. Any attempt to find a DFA that recognize B will fail
- The DFA must remember all number of 0 seen so far and the number is not finite and we cannot do that having finite number of states
- We have studied a method for proving that languages such as B are **not regular**

Pumping Lemma for Regular Languages

Theorem

If A is a regular language, then there is a number p (the pumping length) where if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. *for each* $i \geq 0$, $xy^iz \in A$
2. $|y| > 0$
3. $|xy| < p$

Pumping Lemma for Regular Languages

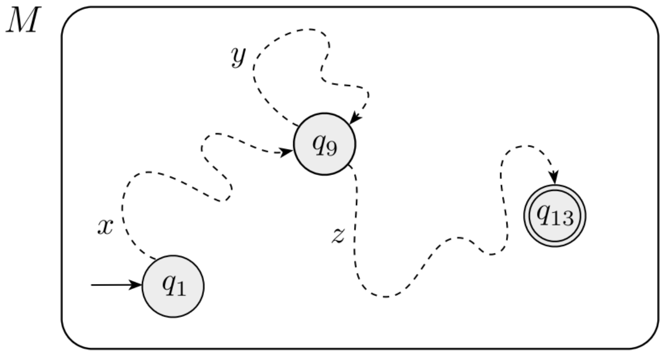


Figure: Example showing how the strings x, y, and z affect M

Section 2

Context setup

Context setup

Corresponding to Sipser 2.1

Context setup

- Until now we introduced two different, though equivalent, methods of describing languages: finite automata and regular expressions
- We showed that many languages can be described in this way but that some simple languages (e.g., $\{0^n1^n | n \geq 1\}$ cannot
- Now we move to a more powerful method of describing languages: **context-free grammars**. Such grammars can describe certain features that have a recursive structure, which makes them useful in a variety of applications
- Context-free grammars were first used in the study of human languages. One way of understanding the relationship of terms such as noun, verb, and preposition and their respective phrases leads to a natural recursion because noun phrases may appear inside verb phrases and vice versa. Context-free grammars help us organize and understand these relationships

Context setup

- An important application of context-free grammars occurs in the specification and compilation of programming languages. A grammar for a programming language often appears as a reference for people trying to learn the language syntax
- Designers of compilers and interpreters for programming languages often start by obtaining a grammar for the language. Most compilers and interpreters contain a component called a **parser** that extracts the meaning of a program prior to generating the compiled code or performing the interpreted execution
- A number of methodologies facilitate the construction of a parser once a context-free grammar is available. Some tools even automatically generate the parser from the grammar

Context setup

- The collection of languages associated with context-free grammars are called the **context-free languages**. They include all the regular languages and many additional languages. In next lectures, we give a formal definition of context-free grammars and study the properties of context-free languages
- We also introduce pushdown automata, a class of machines recognizing the context-free languages. Pushdown automata are useful because they allow us to gain additional insight into the power of context-free grammars

Section 3

Context-free Grammars

Intuition

- A grammar consists of a collection of **substitution rules**, also called **productions**
- Each rule appears as a line in the grammar, comprising a symbol and a string separated by an arrow
- The symbol is called a **variable**. The string consists of variables and other symbols called **terminals**
- The variable symbols often are represented by capital letters
- The terminals are analogous to the input alphabet and often are represented by lowercase letters, numbers, or special symbols
- One variable is designated as the start variable
- It usually occurs on the left-hand side of the topmost rule

Example

For example, grammar G_1 contains three rules. G_1's variables are A and B, where A is the start variable. Its terminals are 0, 1, and $\#$

$$A \rightarrow 0A1$$
$$A \rightarrow B$$
$$B \rightarrow \#$$

Intuition

You use a grammar to describe a language by generating each string of that language in the following manner:

1. Write down the start variable. It is the variable on the left-hand side of the top rule, unless specified otherwise
2. Find a variable that is written down and a rule that starts with that variable. Replace the written down variable with the right-hand side of that rule
3. Repeat step 2 until no variables remain

Example

For example, grammar G_1 generates the string 000#111. The sequence of substitutions to obtain a string is called a **derivation**. A derivation of string 000#111 in grammar G_1 is:

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$$

Example

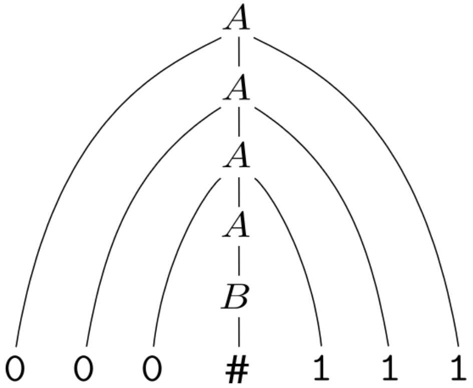


Figure: Parse tree for 000#111 in grammar G_1

Intuition

- All strings generated in this way constitute the language of the grammar
- We write $L(G_1)$ for the language of grammar G_1
- Some experimentation with the grammar G_1 shows us that $L(G_1) = \{0^n\#1^n | n \geq 0\}$
- Any language that can be generated by some context-free grammar is called a **context-free language** (CFL)
- For convenience when presenting a context-free grammar, we abbreviate several rules with the same left-hand variable, such as $A \rightarrow 0A1$ and $A \rightarrow B$, into a single line $A \rightarrow 0A1|B$

Example

Another example of a context-free grammar, called G_2, which describes a fragment of the English language:

$$
\begin{aligned}
<SENTENCE> &\rightarrow <NOUN-PHRASE><VERB-PHRASE>\\
<NOUN-PHRASE> &\rightarrow <CMPLX-NOUN>|<CMPLX-NOUN><PREP-PHRASE>\\
<VERB-PHRASE> &\rightarrow <CMPLX-VERB>|<CMPLX-VERB><PREP-PHRASE>\\
<PREP-PHRASE> &\rightarrow <PREP><CMPLX-NOUN>\\
<CMPLX-NOUN> &\rightarrow <ARTICLE><NOUN>\\
<CMPLX-VERB> &\rightarrow <VERB>|<VERB><NOUN-PHRASE>\\
<ARTICLE> &\rightarrow a|the\\
<NOUN> &\rightarrow boy|girl|flower\\
<VERB> &\rightarrow touches|likes|sees\\
<PREP> &\rightarrow with
\end{aligned}
$$

Example

Grammar G_2 has

- 10 variables (the capitalized grammatical terms written inside brackets)
- 27 terminals (the standard English alphabet plus a space character)
- 18 rules

Strings from $L(G_2$ include:

- a boy sees
- the boy sees a flower
- a girl with a flower likes the boy

Example

The derivation for "a boy sees" is:

$$\begin{aligned}
<SENTENCE> &\Rightarrow <NOUN-PHRASE><VERB-PHRASE>\\
&\Rightarrow <CMPLX-NOUN><VERB-PHRASE>\\
&\Rightarrow <ARTICLE><NOUN><VERB-PHRASE>\\
&\Rightarrow \text{a}<NOUN><VERB-PHRASE>\\
&\Rightarrow \text{a boy}<VERB-PHRASE>\\
&\Rightarrow \text{a boy}<CMPLX-VERB>\\
&\Rightarrow \text{a boy}<VERB>\\
&\Rightarrow \text{a boy sees}
\end{aligned}$$

Formal definition

Definition

A **context-free grammar** is a 4-tuple (V, Σ, R, S) where:

- V is a finite set called the **variables**
- Σ is a finite set, disjoint from V called the **terminals**
- R is a finite set of rules, with each rule being a variable and a string of variables and terminals
- $S \in V$ is the start variable

Formal definition

If u, v, and w are strings of variables and terminals, and $A \rightarrow w$ is a rule of the grammar, we say that uAv **yields** uwv, written $uAv \Rightarrow uwv$

We say that u **derives** v, written $u \overset{*}{\Rightarrow} v$, if $u = v$ or if a sequence $u_1, u_2, \ldots, u_k$ exists for $k \geq 0$ and

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v$$

The **language of the grammar** is $\{w \in \Sigma^* | S \overset{*}{\Rightarrow} w\}$

Example

In grammar G_1 we have $V = \{A, B\}$, $\Sigma = \{0, 1, \#\}$, $S = A$ and R is the collection of the rules

$$A \rightarrow 0A1$$
$$A \rightarrow B$$
$$B \rightarrow \#$$

In case of grammaer G_2 we have:

$$V = \{< SENTENCE >, < NOUN - PHRASE >, < VERB - PHRASE >$$
$$< PREP - PHRASE >, < CMPLX - NOUN >, < CMPLX - VERB >$$
$$< ARTICLE >, < NOUN >, < VERB >, < PREP >\}$$

and $\Sigma = \{a, b, c, \ldots, z\}$ Also we need the blank symbol, placed invisibly after each word

Specification rules

- Often we specify a grammar by writing down only its rules
- We can identify the variables as the symbols that appear on the left-hand side of the rules and the terminals as the remaining symbols
- By convention, the start variable is the variable on the left-hand side of the first rule

Example

Consider grammar $G_3 = (\{S\}, \{a, b\}, R, S)$. The set of rules R is:

$$S \to aSb|SS|\epsilon$$

- This grammar generates strings such as *abab*, *aaabbb* and *aababb*
- You can see more easily what this language is if you think of a as a left parenthesis "(" and b as a right parenthesis ")"
- Viewed in this way, $L(G_3)$ is the language of all strings of properly nested parentheses
- Observe that the right-hand side of a rule may be the empty string ϵ

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

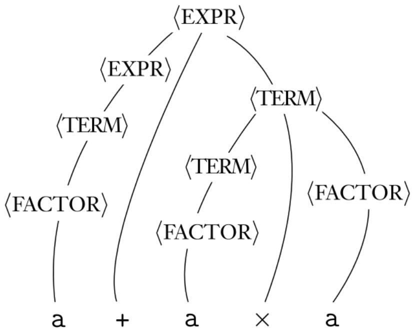
Example

Consider grammar $G_4 = (\{S\}, \{a, b\}, R, < EXPR >)$.
V is $\{< EXPR >, < TERM >, < FACTOR >\}$ and Σ is $\{a, +, x, (,)\}$
The set of rules R is:

$$
\begin{aligned}
<\!EXPR\!> &\rightarrow <\!EXPR\!> + <\!TERM\!> | <\!TERM\!> \\
<\!TERM\!> &\rightarrow <\!TERM\!> x <\!FACTOR\!> | <\!FACTOR\!> \\
<\!FACTOR\!> &\rightarrow (<\!EXPR\!>) | a
\end{aligned}
$$

The two strings $a + axa$ and $(a + a)xa$ can be generated with grammar G_4

Example



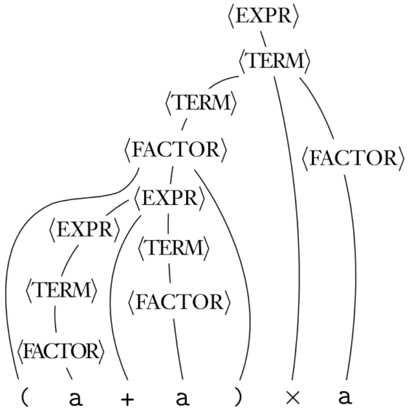


Figure: Parse trees for the strings $a + axa$ and $(a + a)xa$

Example

- Grammar G_4 describes a fragment of a programming language concerned with arithmetic expressions
- Observe how the parse trees "group" the operations. The tree for $a + axa$ groups the x operator and its operands (the second two a's) as one operand of the $+$ operator
- In the tree for $(a + a)xa$, the grouping is reversed. These groupings fit the standard precedence of multiplication before addition and the use of parentheses to override the standard precedence
- Grammar G4 is designed to capture these precedence relations

Section 4

Designing Context-free grammars

General principles

The design of context-free grammars requires creativity

Context-free grammars are even trickier to construct than finite automata because we are more accustomed to programming a machine for specific tasks than we are to describing languages with grammars

General principles

The following techniques are helpful, singly or in combination, when you're faced with the problem of constructing a *CFG*

- Many *CFL*s are the union of simpler *CFL*s. If you must construct a *CFG* for a *CFL* that you can break into simpler pieces, do so and then construct individual grammars for each piece
- Solving several simpler problems is often easier than solving one complicated problem
- These individual grammars can be easily merged into a grammar for the original language by combining their rules and then adding the new rule $S \rightarrow S_1 | S_2 | \ldots | S_k$

Example

To get a grammar for the language $\{0^n1^n | n \geq 0\} \cup \{1^n0^n | n \geq 0\}$

$$S \rightarrow 0S_11|\epsilon$$

for the language $\{0^n1^n | n \geq 0\}$ and the grammar

$$S \rightarrow 1S_10|\epsilon$$

for the language $\{1^n0^n | n \geq 0\}$. Next we add the rule $S \rightarrow S_1|S_2$ and we get the grammar:

$$\begin{aligned} S &\rightarrow S_1|S_2 \\ S_1 &\rightarrow 0S_11|\epsilon \\ S_2 &\rightarrow 1S_10|\epsilon \end{aligned}$$

General principles

Constructing a *CFG* for a language that happens to be regular is easy if you can first construct a *DFA* for that language. You can convert any *DFA* into an equivalent *CFG* as follows:

- Make a variable R_i for each state q_i of the *DFA*. Add the rule $R_i \rightarrow aR_j$ to the *CFG* if $\delta(q_i, a) = q_j$ is a transition in the *DFA*
- Add the rule $R_i \rightarrow \epsilon$ if q_i is an accept state of the *DFA*
- Make R_0 the start variable of the grammar, where q_0 is the start state of the machine
- *Verify on your own that the resulting CFG generates the same language that the DFA recognizes*

General principles

Certain context-free languages contain strings with two substrings that are "linked" in the sense that a machine for such a language would need to remember an unbounded amount of information about one of the substrings to verify that it corresponds properly to the other substring as in $\{0^n 1^n | n \geq 0\}$

You can construct a *CFG* to handle this situation by using a rule of the form $R \rightarrow uRv$, which generates strings wherein the portion containing the u's corresponds to the portion containing the vs.

General principles

In more complex languages, the strings may contain certain structures that appear recursively as part of other (or the same) structures.

That situation occurs in the grammar that generates arithmetic expressions Any time the symbol a appears, an entire parenthesized expression might appear recursively instead.

To achieve this effect, place the variable symbol generating the structure in the location of the rules corresponding to where that structure may recursively appear.