TDDD14/TDDD85 Lecture 7: Context Free Grammars

Jonas Wallgren

Abstract

The second part (lectures 7-13) of the course deals with context free languages. This lecture introduces context free grammars.

1 Introduction

In lecture 6 it was proved (already hinted in lecture 1) that the language $\{0^n1^n|n \geqslant 0\}$ is not regular. In this part of the course we will provide notations, automata, and other algorithms to handle such languages. For regular languages we started with the automata and that was followed by the notation (regular expressions). In this case we begin with the notation (grammars) since it is well-known and is closely connected to programming languages. Automata will follow in lecture 9.

2 Context free grammars

In a formal definition of a programming language you could find someting like:

```
<expression> ::= <expression> * <expression>
               | <expression> + <expression>
               | <number>
<number> ::= <digit> <number>
           | <digit>
<digit> ::= 0|1|2|3|4|5|6|7|8|9
```

This is written in what is called BNF[1]. The first line means: What we call *expression* is composed of one such expression followed by a * followed by another expression. The second line means: Or it is composed of two expressions separated by a +. The third line means: Or, finally, it could be just a copy of what we call a *number*. The next two lines describe that a number is either a *digit* followed by another number or just a digit — i.e. a number is a sequence

[1]Sometimes this is read Backus-Naur Form. More often the Danish computer scientist Naur is ignored and it is read Backus Normal Form. It was invented together with the definition of the programming language Algol 60.

of digits. The final line describes what a digit could be. The ::= represents an arrow. It was invented in an era of poor characer sets. := was used for assignment in e.g. Algol, so they invented this one for grammar use.

This is a grammar — a set of rules describing the syntax, the constituents, of a complex construction. We will come to "context free" in a while.

There are two directions in reading a rule. Left-to-right, "is composed of", is the way to read if you want to split a program up in its smaller parts. Right-to-left, "builds", is the way to read if you want to see what makes up a whole program, how to construct it from its parts. Thus, this is the intended meaning of the ::= symbol.

Sometimes we discuss concrete examples like this one with expressions and numbers. Then we want readable names for all the parts, like expression. Such a name is pleced inside <> to show that it is not a part of the final string, like the digits or * and +. This is something that must be further described. In reading the rules from left to right it means that you cannot stop there, you must continue. They are called *non-terminals*. The digits and the operator symbols above will be found in the final string. There you cannot continue. They are called *terminals*

Sometimes we instead discuss grammar constructions in principle. Then we don't need any specific names for the non-terminals. Instead we use capital, latin letters without <>. Then it's also common to use a real arrow instead of ::=. Like

Example 1
$E \rightarrow E*E|E+E|N$
$N \rightarrow DN|D$
$D \rightarrow 0|1|2|3|4|5|6|7|8|9$

Yes, in this style it's most common to use $\rightarrow$, but $\leftarrow$ is used by some authors. Of course, real arrows could be used with the <> notation and ::= could be used without <>.

Yes, this grammar is of course inspired be the one above, that's why the names E, N, and D instead of e.g. A, B, and C.

To be precise: What we have seen this far is in some cases really are abbreviations of rules, to ease the writing and reading of them. It really should be

<expression> ::= <expression> * <expression>
<expression> ::= <expression> + <expression>
<expression> ::= <number>

and each such line is what will be called a (production) rule in more formal settings.

A part of the grammar above defines numbers, but the grammar in its whole defines expressions. To see every string in the language you should start looking at <expression>. It is a grammar for expressions. <expression> is called the start symbol.

So, we are ready for a definition:

Definition 1 *A grammar is a quadruple* $G=<N,\Sigma,P,S>$ *where*
N=*set of nonterminals*
Σ=*set of terminals (the alphabet)*
P=*set of production rules*$\subseteq N\times(N\cup\Sigma)^*$
S=*start symbol*$\in N$

Thus, P is a set of elements where each element has a left hand side non-terminal and a right hand side sequence of terminals and non-terminals

3 Derivation

If you begin with the start symbol and step by step in some way use the grammar rules and finally end up with a string of terminals you have performed a derivation.

In general discussions we will use small greek letters to stand for sequences of terminals and non-terminals, i.e. in this case

$$\alpha, \beta, \gamma \in (N \cup \Sigma)^*$$

Capital roman letters will stand for non-terminals.

So, if we in some way have reached $\alpha A\gamma$ and in the grammar there is a rule A$\rightarrow \beta$ then we can get $\alpha\beta\gamma$, i.e. the middle A has been replaced by the β from the rigt-hand-side of the grammar rule. It's written

$$\alpha A\gamma \Rightarrow \alpha\beta\gamma$$

This is one derivation step. More precisely it is one context-free derivation step. You don't bother about what α and γ are. You could always do the replacement of A with β. You can ignore the *context* of A.

If you make several derivation steps one after another you perform a derivation, denoted by $\stackrel{*}{\Rightarrow}$. E.g.:

$$\text{E}\Rightarrow\text{E+E}\Rightarrow\text{N+E}\Rightarrow\text{N+N}\Rightarrow\ \text{DN+N}\Rightarrow\text{1N+N}\Rightarrow\cdots\Rightarrow\text{123+456}$$

Thus

$$\text{E}\stackrel{*}{\Rightarrow}\text{123+456}$$

Now, we can define the language of a grammar:

Definition 2 $L(G)=\{w\in\Sigma^*|S\stackrel{*}{\Rightarrow}w\}$

i.e. the language of a given grammar is the set of all strings over the alphabet that can be derived from the start symbol.

A context-free language (CFL) is the language of a CFG.

Example 2 *Grammar G_1:*
N=$\{X\}$
Σ=$\{a,b\}$
S=X
P=$\{X \to aXb|\varepsilon\}$

$X \Rightarrow \epsilon$, $X \overset{*}{\Rightarrow} \varepsilon$
$X \Rightarrow aXb \Rightarrow ab$, $X \overset{*}{\Rightarrow} ab$
$X \Rightarrow aXb \Rightarrow aaXbb \Rightarrow aabb$, $X \overset{*}{\Rightarrow} aabb$

Thus $\{\varepsilon, ab, aabb\} \subseteq L(G_1)$

It's far from a proof but it clearly indicates that $L(G_1)=\{a^n b^n | n \geqslant 0\}$*, the languages earlier proven to be non-regular.*

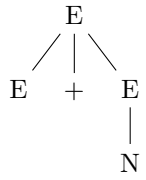
Example 3 *Grammar G_2:*
N=$\{X\}$
Σ=$\{a,b\}$
S=X
P=$\{X \to aXa|bXb|a|b|\varepsilon\}$

$X \Rightarrow aXa \Rightarrow abXba \Rightarrow abbXbba \Rightarrow abbbba$
$X \Rightarrow bXb \Rightarrow baXab \Rightarrow babab$

It seems like $L(G_2)=\{x \in \{a,b\}^* | x = x \text{ reversed}\}$*, i.e. palindromes over* Σ*.*

4 Derivation trees, parse trees

The idea is that the derivation E⇒E+E⇒ E+N⇒ $\cdots$ could be depicted in a tree like



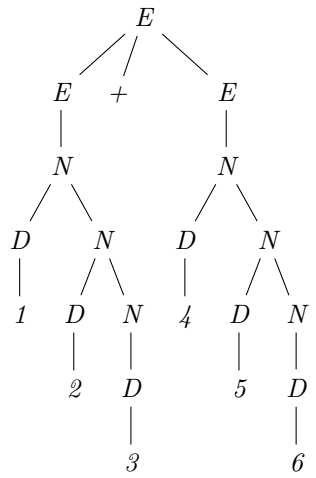
The start symbol E is found in the root. In the first derivation step that E is replaced by E+E. That is shown with the three children of the root. In the next step the rightmost E is replaced by N. In the tree that is shown by making N a child of the rightmost E.

Since such a tree shows a derivation it's called a derivation tree. In the way we constructed it in this example we read the grammar rules from left to right. When you use a grammar to parse a string, e.g. a computer program, you are given the leaves of the tree and the job is to build the whole tree up to its root.

Then it's natural to call it a parse tree. Independently of application the notions of inference tree and parse tree are used interchangeably.

Definition 3 *A derivation tree is a tree such that:*
The root of a derivation tree is S.
Each leaf of a derivation tree $\in \Sigma$.
Each inner node of a derivation tree $\in N$.
If the node A has the children p, q, r, . . . then there is a rule $A \to pqr \ldots \in P$.

Example 4 *The derivation tree for the string 123+456:*



5 Left and right derivations, ambiguity

In this section we will discuss the grammar

E→E*E|E+E|a|b|c

An example derivation:

E⇒E*E⇒E+E*E⇒a+E*E⇒ a+b*E⇒a+b*c (1)

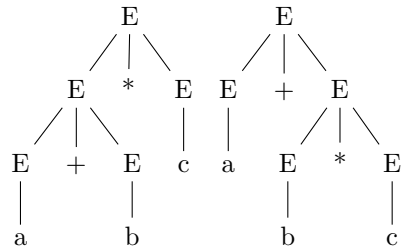
In this derivation in every step the leftmost non-terminal has been chosen. It is called a leftmost derivation. Symbol: $\Rightarrow_{lm}$

Another derivation:

E⇒E+E⇒E+E*E⇒E+E*c⇒ E+b*c⇒a+b*c (2)

In this derivation in every step the rightmost non-terminal has been chosen. It is called a rightmost derivation. Symbol: $\Rightarrow_{rm}$

The derivation trees corresponding to these two derivations are:



The left tree corresponds to derivation (1) and the right tree corresponds to derivation (2).

So, there are (at least) two different ways to derive the string a+b*c, and there are two different derivation trees for it.

A third derivation of the same string is

E⇒E+E⇒E+E*E⇒a+E*E⇒ a+b*E⇒a+b*c

This is another leftmost derivation but it corresponds to the right tree.

We want to be able to analyze every string in exactly one way, but here we found a problematic string. It's an ambiguity. A grammar is ambiguous if there exists a string that has more than one left-most derivation, more than one right-most derivation, more than one derivation tree (occurs simultaneously).

This grammar handles arithmetic expressions. Of course we want a+b*c to mean that the product is calculated first and then the sum. This corresponds to tree (2). Tree (1) corresponds to the sum beeing calculated first. How to solve that problem?

Here follows the standard solution — an unambiguous grammar for arithmetic expressions that respects the normal priorities and associativities. It also includes bracketed expressions and some more operators:

Example 5
$E \rightarrow E+T|E\text{-}T|T$
$T \rightarrow T*F|T/F|F$
$F \rightarrow (E)|a|b|c$

E means expression, T means term, F means factor.

6 An important relationship

One first step of building relations between different classes of languages is to see that all regular languages are context free, so the set of all regular languages is a subset of all context free languages. See tutorial problem 6.1–3.

7 More to think about

1. Do you think that your favourite programming language is context free? I.e. can a context free grammar describe the correct programs and nothing else?

2. Is there a context free grammar for $\{a^n b^n c^n | n \geqslant 0\}$?