Lecture 20: Context-free grammars

8 April 2010

This lecture introduces context-free grammars, covering section 2.1 from Sipser.

1 Context-free grammars

1.1 Introduction

Regular languages are efficient but very limited in power. For example, not powerful enough to represent the overall structure of a `C` program.

As another example, consider the following language

$$L = \{\text{all strings formed by properly nested parenthesis}\}.$$

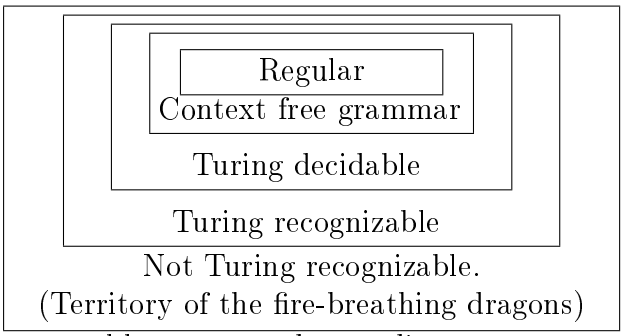
Here, the string (()()) is in L. ())(is not.

Lemma 1.1 *The language L is not regular.*

Proof: Assume for the sake of contradiction that L is regular. Then consider $L' = L \cap {}'({}'^*)^*$. Since L is regular and regular languages are closed under intersection, L' must be regular. But L is just $\left\{ ({}'^n)^n \;\middle|\; n \geq 0 \right\}$. We can map this, with a homomorphism, to $\mathtt{0}^n\mathtt{1}^n$, which is not regular (as we seen before). A contradiction. ■

Our purpose is to come up with a way to describe the above language L in a compact way. It turns out that context-free grammars are one possible way to capture such languages.

Here is a diagram demonstrating the classes of languages we will encounter in this class. Currently, we only saw the weakest class – regular language. Next, we will see context free grammars.



A compiler or a natural language understanding program, use these languages as follows:

- It uses regular languages to convert character strings to tokens (e.g. words, variables names, function names).

- It uses context-free languages to parse token sequences into functions, programs, sentences.

Just as for regular languages, context-free languages have a procedural and a declarative representation, which we will show to be equivalent.

procedural	declarative
NFAs/DFAs	regular expressions
pushdown automata (PDAs)	context-free grammar

1.2 Deriving the context-free grammars by example

So, consider our old arch-nemesis, the language

$$L = \left\{ \mathtt{a}^n \mathtt{b}^n \;\middle|\; n \geq 0 \right\}.$$

we would like to come up with a recursive definition for a word in the language.

So, let S denote any word we can generate in the language, then a word w in this language can be generated as

$$w = \mathtt{a}^n \mathtt{b}^n = \mathtt{a} \underbrace{\boxed{\mathtt{a}^{n-1}\mathtt{b}^{n-1}}}_{=w'} \mathtt{b},$$

where $w' \in L$. Thus, we have a compact recursive way to generate L. It is the language containing the empty word, and one can generate a new word, by taking a word w' already in the language and padding it with a before, and b after it. Thus, generating the new word $\mathtt{a}w'\mathtt{b}$. This suggests a random procedure S to generate such a word. It either return without generating anything, or it prints a a, generates a word recursively by calling S, and then it outputs a b. Naturally, the procedure has to somehow guess which of the two options to perform. We demonstrate this idea in the C program on the right, where S uses randomization to decide which action to take. As such, running this program would generate a random word in this language.

```
#include  <stdlib.h>
#include  <stdio.h>
#include  <time.h>
int  guess()
{ return  random() % 16; }
void  S() {
    if  ( guess() == 0 ) return;
    else {
        printf( "a" );
        S();
        printf( "b" );
    }
}
int  main() {
    srand( time( 0 ) );
    S();
}
```

The way to write this recursive generation algorithm using context free grammar is by specifying

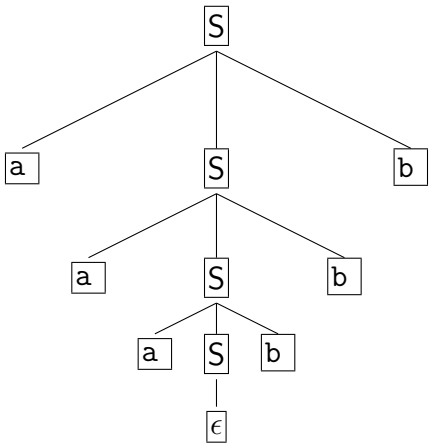
$$\mathsf{S} \rightarrow \epsilon \;\mid\; \mathsf{aSb}. \tag{1}$$

Thus, CFG can be taught of as a way to specify languages by a recursive means: We can build sole basic words, and then we can build up together more complicated words by recursively building fragments of words and concatenating them together.

For example, we can derive the word aaabbb from the grammar of Eq. (1), as follows:

$$\mathsf{S} \rightarrow \mathtt{aSb} \rightarrow \mathtt{aaSbb} \rightarrow \mathtt{aaaSbbb} \rightarrow \mathtt{aaa}\epsilon\mathtt{bbb} = \mathtt{aaabbb}.$$

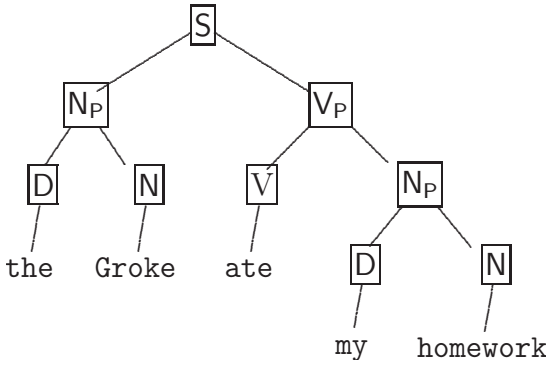
Alternatively, we can think about the recursion tree used by our program to generate this string.



This tree is known as the ***parse tree*** of the grammar of Eq. (1) for the word aaabbb.

1.2.1 Deriving the context-free grammars by constructing sentence structure

A context-free grammar defines the syntax of a program or sentence. The structure is easiest to see in a parse tree.



The interior nodes of the tree contain "variables", e.g. $\mathsf{N_P}$, D. The leaves of the tree contain "terminals". The "yield" of the tree is the string of terminals at the bottom. In this case, "the Groke at my homework." [1]

The grammar for this has several components:

(i) A start symbol: S.

(ii) A finite set of variables: $\{\mathsf{S}, \mathsf{N_P}, \mathsf{D}, \mathsf{N}, \mathsf{V}, \mathsf{V_P}\}$

(iii) A finite set of terminals = $\{\mathtt{the}, \mathtt{Groke}, \mathtt{ate}, \mathtt{my}, \ldots\}$

[1] Groke – Also known in Norway as Hufsa, in Estonia as Urr and in Mexico as La Coca is a fictional character in the Moomin world created by Tove Jansson.

(iv) A finite set of rules.

Example of how rules look like

(i) $\mathsf{S} \to \mathsf{N_P}\ \mathsf{V_P}$

(ii) $\mathsf{N_P} \to \mathsf{D}\ \mathsf{N}$

(iii) $\mathsf{V_P} \to \mathsf{V}\ \mathsf{N_P}$

(iv) $\mathsf{N} \to |\ \texttt{Groke} \mid$ homework $\mid$ lunch $\ldots$

(v) $\mathsf{D} \to \texttt{the} \mid \texttt{my} \ldots$

(vi) $\mathsf{V} \to \texttt{ate} \mid \texttt{corrected} \mid$ washed $\ldots$

If projection is working, show a sample computer-language grammar from the net. (See pointers on web page.)

1.2.2 Synthetic examples

In practical applications, the terminals are often whole words, as in the example above. In synthetic examples (and often in the homework problems), the terminals will be single letters.

Consider $L = \left\{ \texttt{0}^n\texttt{1}^n \;\middle|\; n \geq 0 \right\}$. We can capture this language with a grammar that has start symbol S and rule

$$\mathsf{S} \to \texttt{0}\mathsf{S}\texttt{1} \mid \epsilon\ .$$

For example, we can derive the string `000111` as follows:

$$\mathsf{S} \to 0\mathsf{S}1 \to 00S11 \to 000S111 \to 000\epsilon111 = 000111.$$

Or, consider the language of palindromes $L = \left\{ w \in \{a,b\}^* \;\middle|\; w = w^R \right\}$. Here is a grammar with start symbol P. for this language

$$\mathsf{P} \to \texttt{a}\mathsf{P}\texttt{a} \mid \texttt{b}\mathsf{P}\texttt{b} \mid \epsilon \mid \texttt{a} \mid \texttt{b}.$$

A possible derivation of the string `abbba` is

$$\mathsf{P} \to \texttt{a}\mathsf{P}\texttt{a} \to \texttt{ab}\mathsf{P}\texttt{ba} \to \texttt{abbba}.$$

2 Derivations

Consider our Groke example again. It has only one parse tree, but multiple derivations: After we apply the first rule, we have two variables in our string. So we have two choices about which to expand first:

$$S \to \mathsf{N_P}\ \mathsf{V_P} \to \ldots$$

If we expand the leftmost variable first, we get this derivation:

$$S \to \mathsf{N_P}\ \mathsf{V_P} \to \mathsf{D}\ N\ \mathsf{V_P} \to \textit{the}\ N\ \mathsf{V_P} \to \textit{the Groke}\ \mathsf{V_P} \to \textit{the Groke}\ V\ \mathsf{N_P} \to \dots$$

If we expand the rightmost variable first, we get this derivation:

$$\begin{aligned} \mathsf{S} &\to \mathsf{N_P}\ \mathsf{V_P} \to \mathsf{N_P}\ V\ \mathsf{N_P} \to \mathsf{N_P}\ V\ \mathsf{D}\ N \to \mathsf{N_P}\ V\ \mathsf{D}\ \texttt{homework} \\ &\to \mathsf{N_P}\ V\ \texttt{my homework} \dots \end{aligned}$$

The first is called the ***leftmost derivation***. The second is called the ***rightmost derivation***. There are also many other possible derivations. Each parse tree has many derivations, but exactly one rightmost derivation, and exactly one leftmost derivation.

2.1 Formal definition of context-free grammar

Definition 2.1 (CFG) A ***context-free grammar*** (CFG) is a 4-tuple $\mathcal{G} = (V, \Sigma, R, \mathsf{S})$, where

(i) $\mathsf{S} \in V$ is the ***start variable***,

(ii) Σ is the alphabet (as such, we refer to $c \in \Sigma$ as a character or ***terminal***),

(iii) V is a finite set of ***variables***, and

(iv) R is a finite set of rules, each is of the form $\mathsf{B} \to w$ where $\mathsf{B} \in V$ and $w \in (V \cup \Sigma)^*$ is a word made out of variables and terminals..

Definition 2.2 (CFG yields.) Suppose x, y, and w are strings in $(V \cup \Sigma)^*$ and B is a variable. Then $x\mathsf{B}y$ ***yields*** xwy, written as

$$x\mathsf{B}y \Rightarrow xwy,$$

if there is a rule in R of the form $\mathsf{B} \to w$.

Notice that $x \Rightarrow x$, for any x and any set of rules.

Definition 2.3 (CFG derives.) If x and y in $(V \cup \Sigma)^*$, then w ***derives*** x, written as

$$w \stackrel{*}{\Rightarrow} x$$

if you can get from w to x in zero or more yields steps.

That is, there is a sequence of strings $y_1, y_2, \dots y_k$ in $(V \cup \Sigma)^*$ such that

$$w = y_1 \Rightarrow y_2 \Rightarrow \dots \Rightarrow y_k = x.$$

Definition 2.4 If $\mathcal{G} = (V, \Sigma, R, S)$ is a grammar, then $L(\mathcal{G})$ (the ***language*** of $\mathcal{G}$) is the set

$$L(\mathcal{G}) = \left\{ w \in \Sigma^* \;\middle|\; S \stackrel{*}{\Rightarrow} w \right\}..$$

That is, $L(G)$ is all the strings containing only terminals which can be derived from the start symbol of $\mathcal{G}$.

2.2 Ambiguity

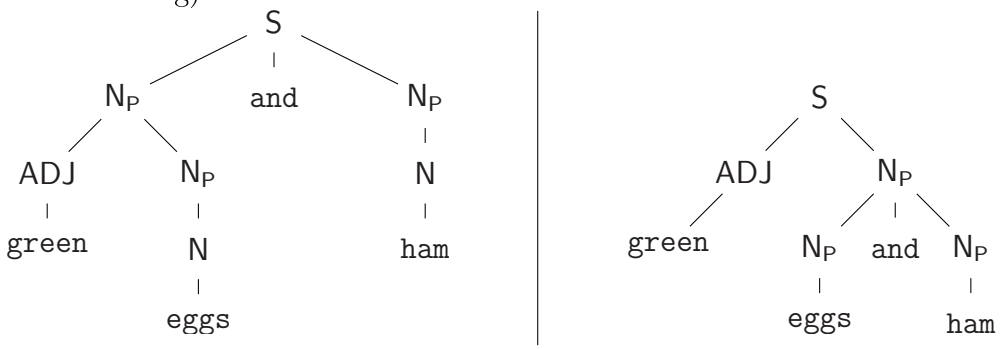
Consider the following grammar $\mathcal{G} = (V, \Sigma, R, \mathsf{S})$. Here

$$V = \{\mathsf{S}, N, \mathsf{N_P}, \mathsf{ADJ}\} \qquad \text{and} \qquad \Sigma = \{\texttt{and}, \texttt{eggs}, \texttt{ham}, \texttt{pencilgreen}, \texttt{cold}, \texttt{tasty}, \ldots\}\,.$$

The set R contains the following rules:

- $\mathsf{N_P} \to \mathsf{N_P}$ `and` $\mathsf{N_P}$
- $\mathsf{N_P} \to \mathsf{ADJ}\ \mathsf{N_P}$
- $\mathsf{N_P} \to \mathsf{N}$
- $\mathsf{N} \to$ `eggs` | `ham` | `pencil` | ...
- $\mathsf{ADJ} \to$ `green` | `cold` | `tasty` | ...
- ...

Here are two possible parse trees for the string `green eggs and ham` (ignore the spacing for the time being).



The two parse trees group the words differently, creating a different meaning. In the first case, only the eggs are green. In the second, both the eggs and the ham are green.

A string w is ***ambiguous*** with respect to a grammar $\mathcal{G}$ if w has more than one possible parse tree using the rules in $\mathcal{G}$.

Most grammars for practical applications are ambiguous. This is a source of real practical issues, because the end users of parsers (e.g. the compiler) need to be clear on which meaning is intended.

2.2.1 Removing ambiguity

There are several ways to remove ambiguity:

(A) Fix grammar so it is not ambiguous. (Not always possible or reasonable or possible.)

(B) Add grouping/precedence rules.

(C) Use semantics: choose parse that makes the most sense.

Grouping/precedence rules are the most common approach in programming language applications. E.g. "else" goes with the closest "if", * binds more tightly than +.

Invoking semantics is more common in natural language applications. For example, "The policeman killed the burgler with the knife." Did the burgler have the knife or the policeman? The previous context from the news story or the mystery novel may have made this clear.

E.g. perhaps we have already been told that the burgler had a knife and the policeman had a gun.

Fixing the grammar is less often useful in practice, but neat when you can do it. Here's an ambiguous grammar with start symbol E. N stands for "number" and E stands for "expression".

$$\mathsf{E} \to \mathsf{E} \times \mathsf{E} \mid \mathsf{E}{+}\mathsf{E} \mid \mathsf{N}$$

$$\mathsf{N} \to \mathtt{0}\mathsf{N} \mid \mathtt{1}\mathsf{N} \mid \mathtt{0} \mid \mathtt{1}$$

An expression like $\mathtt{0110} \times \mathtt{110} + \mathtt{01111}$ has two parse trees and, therefore, we do not know which operation to do first when we evaluate it.

We can remove this ambiguity as follows, by rewriting the grammar as

$$\mathsf{E} \to \mathsf{E} + \mathsf{T} \mid \mathsf{T}$$

$$\mathsf{T} \to \mathsf{N} \times \mathsf{T} \mid \mathsf{N}$$

$$\mathsf{N} \to \mathtt{0}\mathsf{N} \mid \mathtt{1}\mathsf{N} \mid \mathtt{0} \mid \mathtt{1}$$

Now, the expression $\mathtt{0110} \times \mathtt{110} + \mathtt{01111}$ must be parsed with the $+$ as the topmost operation.