Context-Free Languages

This material is described in Chapter 10 of the textbook.

We have observed that regular languages have nice properties, however, the specification capabilities of regular expressions and state diagrams are limited. Next we consider a rewriting mechanism, or grammar, that overcomes some of the limitations. Context-free grammars are more powerful than state diagrams, that is, they define a larger class of languages.

Context-free grammars are the most widely used specification tool for the syntactic structure of programming languages. The appendix of the textbook contains a grammar specification for a subset of the C language that covers features that are needed for the program verification part of the course.

In the programming language community grammars were originally specified using the Backus-Naur formalism (BNF):

- the nonterminals are indicated with angle brackets
- the left and right sides of rules (productions) are separated by `::=`
- `|` indicates alternative definitions

Example. `<expr> ::= <expr> + <expr> | <expr>` × `<expr> | (<expr>) | a`

Note that the string `a + a` × `a` has two essentially different derivations: the derivation can begin either with the "plus" rule or the "product" rule. Which derivation should we use?

Notational convention for grammars: In the following we will use → instead of `::=` for the productions (or rewriting rules). We normally denote nonterminals/variables by upper case letters and terminal symbols by lower case letters or digits.

Example. Design a context-free grammar for the language

$$\{0^n 1^n \mid n \geq 0\} \cup \{1^n 0^n \mid n \geq 0\}$$

Next we will go through a more formal definition of a context-free grammar. Also we define formally the derivations of the grammar and the set of terminal strings (language) generated by the grammar.

Definition. A context-free grammar (CFG) is a 4-tuple (V, Σ, P, S) where

1. V is a finite set of nonterminals (also called variables)
2. Σ is a finite set of terminals; Σ and V are disjoint
3. P is a finite set of productions, the left side of each production is a nonterminal and the right side is a string of nonterminals and terminals
4. $S \in V$ is the start nonterminal

The definition means that the set of productions P is a finite subset of $V \times (\Sigma \cup V)^*$. The productions are applied only to variables, that is, the left side of a production is always a variable. The right side of a production is a string of variables and terminal symbols.

A derivation begins with the start variable S. The productions of the grammar are applied to any variable occurring in the "current string" (a.k.a. sentential form) and the process is continued as long a we get a string consisting only of terminal symbols. The language generated by the grammar consists of all terminal strings that are obtained in this way. Next we define formally the notion of *derivation.*

Let w_1 and w_2 be strings over $\Sigma \cup V$ (sets of terminals and nonterminals). We say that w_2 is immediately derivable from w_1 if w_2 can be obtained from w_1 by replacing one occurrence of a nonterminal by a string that appears on the right side of a production for that nonterminal.

Using notation the single step derivation is defined as follows:

- $w_1 = xNy$, $N \in V$, $x, y \in (V \cup \Sigma^*)$,
- $w_2 = xzy$ where $N \to z$ is a production of P.

If w_2 is obtained from w_1 in a single derivation step, we write $w_1 \Rightarrow w_2$.

The language generated by the grammar is the set of terminal strings that can be derived from the start nonterminal, that is,

$$\{w \in \Sigma^* \mid S \Rightarrow^* w\}$$

The relation $\Rightarrow^*$ is the *reflexive-transitive closure* of the single-step derivation relation $\Rightarrow$. That is, $u \Rightarrow^* v$ if $u = v$ or there exists a sequence of strings $u_1, \ldots, u_k$, $k \geq 0$, such that

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

Example. Consider the grammar

$$G = (\{S\}, \{a, b\}, P, S)$$

where the productions of P are

$$S \to aSb \mid SS \mid \varepsilon$$

What is the language generated by this grammar? How can you describe the set of terminal strings generated by this grammar.

Regular grammars. A context-free grammar where all productions are of the forms

$B \to bC$

$B \to b \mid \varepsilon$

where B, C are nonterminals and b is a terminal, is called a regular grammar. It is not difficult to show that regular grammars generate exactly the regular languages. (Time permitting this can be discussed in class.) This means that every regular language is generated by some context-free grammar. We have already seen that there exist context-free languages that are not regular.

Parse Trees and Ambiguity

A context-free derivation can apply a production to any variable in a sentential form. This means that normally there are multiple ways to derive a given terminal string – exceptions can be, for example, right-linear grammars where a sentential form includes only one variable.

To begin with a very simple example, consider a grammar with productions:

$$S \to AB, \quad A \to a \quad B \to b$$

where S is the start variable, A, B are variables and a, b are terminals. Now the terminal string has two different derivations

$$S \Rightarrow AB \Rightarrow aB \Rightarrow ab \quad \text{and} \quad S \Rightarrow AB \Rightarrow Ab \Rightarrow ab.$$

However, the derivations differ only in the order of production applications and both derivations have the *same parse tree.*

The notion of *ambiguity* refers to a situation where a given terminal string has more than one parse tree. Since compilers do program translation based on the parse tree it is desirable that each terminal string can be parsed in only one way - such grammars are called *unambiguous.*

Consider our earlier example with a simplified expression grammar[1]:

`<expr>` $\to$ `<expr> + <expr> | <expr>` $\times$ `<expr> | (<expr>) | a`

[1]For simplicity we use only one terminal value "a" for an expression. A more realistic grammar would have additional rules that can generate e.g. any integer value.

Since expressions would be evaluated based on the parse tree, ambiguity is an undesirable feature. For example, the expression `a + a × a` can be derived by first using the production `<expr>` → `<expr> + <expr>` or `<expr>` → `<expr> × <expr>` and these derivations yield distinct parse trees.

Here if the expression is evaluated according to the parse tree, the two parse trees may produce different values! If we consider how the expression `a + a × a` is evaluated and the usual precedence rules for addition and multiplication, the parse tree where we apply first the production `<expr>` → `<expr> + <expr>` yields the correct evaluation.

Ambiguity is a challenging problem and even determining whether a given grammar is unambiguous is a computationally unsolvable problem. There exist ad hoc techniques to remove ambiguity from a grammar. In our example, we can redesign the above grammar by introducing new nonterminals:

`<expr>` → `<term> | <expr> + <term>`

`<term>` → `<factor> | <term> × <factor>`

`<factor>` → `(<expr>) | a`

The intuitive idea is that, in the modified grammar, variable `<expr>` genereates a sum of "terms" and each variable `<term>` generates a product of "factors". The modified grammar generates the same terminal strings (expressions) as the original grammar and it is unambiguous. (The example may be discussed in more detail in class.)

Pushdown automata (Section 10.5)

A grammar is a generative mechanism: the process begins with the start variables and applies rewriting rules to produce a terminal string. On the other hand, when a compiler receives a program (that is, a terminal string) as input and has to process the parse tree based on the program. For this purpose we use a machine model called *pushdown automaton*

that is equivalent to a context-free grammar. The parser stage of a compiler will be, roughly speaking, a deterministic pushdown automaton.

A pushdown automaton (PDA) can write symbols on a stack and read them back later. Writing a symbol "pushes down" all the other symbols on the stack. The symbol on the top of the stack can be read and removed ("pop" operation), see Figure **??**.

A pushdown automaton can be depicted as a directed graph. Similarly as for state diagrams, the states are depicted by nodes (circle) in the graph. The arrows labeling the state transitions need to include both the input symbol read and the operation performed on the stack.

According to notational conventions used in our textbook, a transition from state q_1 to state q_2 labeled by

$$c, d \mapsto w$$

indicated that the computation can switch from state q_1 to state q_2 by reading c from the input, popping (removing) d from the top of the stack and pushing string w to the stack. Here c is an individual input symbol or the empty string, d is an individual stack symbol or the empty string, and, w is a string of stack symbols. That is, one operation pops at most on symbol from the stack but an operation can push a string of stack symbols.

A pushdown automaton begins processing the input in the initial state and with the stack contents empty. The computation accepts the input if at the right end of the input the machine is in an accepting state and the stack is empty.

More details of the definition of pushdown automata and examples are given on pages 216–219 in the textbook.

Note. While the definition of state diagrams (DFAs or NFAs) is fairly consistent in different sources, the precise details of a pushdown automaton definition vary depending on different textbooks.[2] In the lectures and assignments we will follow the notational conventions of our

[2]For example, some textbooks use different acceptance conditions for a computation.

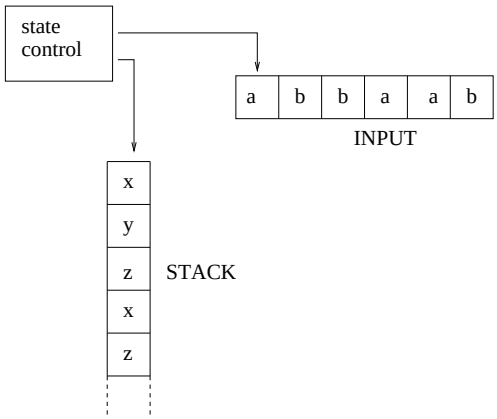

Figure 1: A pushdown automaton

textbook.

Example. Construct a pushdown automaton for the language

$$\{a^i b^i \mid i \geq 0\} \cup \{a^{2i} b^i \mid i \geq 0\}$$

(To be done in class.)

Pushdown automata recognize exactly the class of context-free languages. We will go through the construction showing how to construct a pushdown automaton recognizing the language generated by a grammar. The converse inclusion is omitted in this course.

Pumping lemma for context-free languages (Section 10.6)

Context-free languages form a larger class than regular languages. Context-free grammars have important applications in specifying the syntax of programming languages, however, context-free grammars cannot specify certain programming language feature, like the requirement that variable names need to be declared before their use.

To have an understanding of the limitations of context-free grammars we need a tool for proving that a language is not context-free. The *context-free pumping lemma* is based on similar ideas as our earlier pumping lemma for regular languages. Roughly speaking, the

difference is that in the context-free pumping we have to repeat in parallel two substrings.

Pumping Lemma for Context-Free Languages. For every context-free language L there is a constant p such that every string $s \in L$ of length at least p can be written as

$$s = uvwxy$$

where

(i) $v \neq \varepsilon$ or $x \neq \varepsilon$

(ii) $|vwx| \leq p$

(iii) for each $i \geq 0$, $uv^iwx^iy \in L$

Similarly as with regular languages, the use of the context-free pumping lemma is a proof by contradiction. For the sake of contradiction we assume that the language is context-free. Then from the pumping lemma we get the constant p and select a string s of length at leastr p. To derive a contradiction we have to prove that no matter how we try to divide s into five parts $uvwxy$ the parts cannot satisfy the above conditions (i), (ii), (iii).

Because the context-free pumping lemma repeats two substrings and we consider decompositions into five parts, the details of the proof are some times more complicated and involve a case analysis to cover all possible ways to divide s into five parts. The following two examples will be covered in detail in class.

Examples. Show that the following languages are not context-free:

a) $\{a^ib^ic^i \mid i \geq 0\}$

b) $\{ww \mid w \in \{a, b\}^*\}$

Note that above the language $\{ww \mid w \in \{a, b\}^*\}$ abstracts the idea that context-free grammars cannot specify the requirement that variable names have to be declared before

their use. Assuming there is no upper limit on the length of a variable name, a grammar specifying the requirement would need to generate strings of the form

$$\ldots w \ldots w \ldots, \text{ for } w \in \Sigma^*$$

where ... refer to other parts of the program that are omitted in the abstraction.

Applications of grammars: RNA secondary structure prediction using context-free grammars

Regular languages and finite state machines are often used in DNA and protein sequence matching problems (see notes for week 1). The use of regular languages is based on the assumption that the mutations that caused variations in the sequence occurred independently of each other.

The situation is different when we consider the secondary (or three dimensional) structure of an RNA molecule. RNA molecules with same secondary structure usually have the same function. Because of the way RNA molecules fold, in order to preserve the secondary structure, variations to nucleotides in one part of the sequence must be matched in the corresponding bonded subsequence (the bonded subsequences are called a *stem*.) This type of nested dependencies can be described using context-free grammars.

We consider a simple example. The RNA nucleotides contain four different bases: adenine (A), guanine (G), cytosine (C), uracil (U). C and G, and A and U, respectively, are complementary. In the below grammar we use lower case letters (a, g, c, u) since they are the terminals of the grammar.

To generate RNA structures we use a grammar (V, Σ, P, S) where $V = \{S\}$, $\Sigma = \{a, g, c, u\}$, and P consists of the productions:

(i) $S \longrightarrow aSu \mid uSa \mid cSg \mid gSc$

(ii) $S \longrightarrow aS \mid cS \mid gS \mid uS$

(iii) $S \longrightarrow Sa \mid Sc \mid Sg \mid Su$

(iv) $S \longrightarrow SS \mid \varepsilon$

A parse tree for the terminal string

$$ccugagaggcaaccuagaaggu \tag{1}$$

is given in Figure **??**. The parse tree corresponds to the RNA secondary structure with two stems (or bonded subsequences) that is illustrated in Figure **??**.

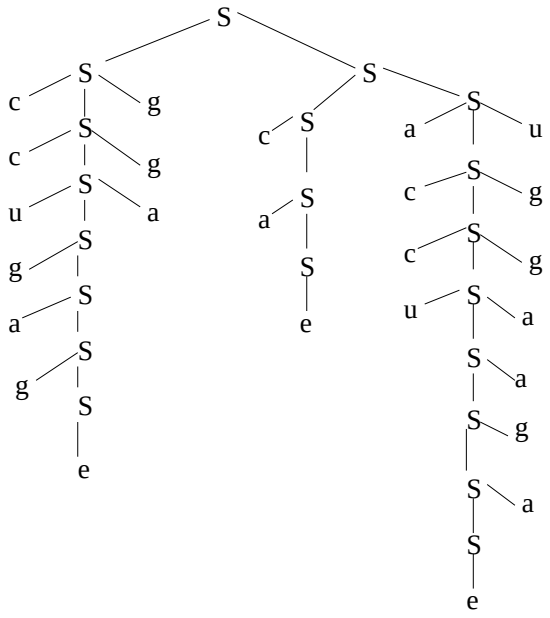

Figure 2: Parse tree for an example RNA structure. In the figure, "e" denotes the empty string ε.

From the parse tree we can read the secondary structure: the application of the grammar rules (i) produces a stem of bonded base pairs. Note that the grammar is *ambiguous*: the same RNA sequence (**??**) has many other parse trees. The ambiguity is, for the most part, caused by the possibility of applying the rules (ii), (iii) in different order. Derivations that differ only in the order of application of the rules (ii) and (iii) correspond to the same secondary structure.

Above we have outlined how context-free grammars can be used to model RNA folding. A more detailed study tells us that complementary pairs (C–G and A–U) are the likeliest

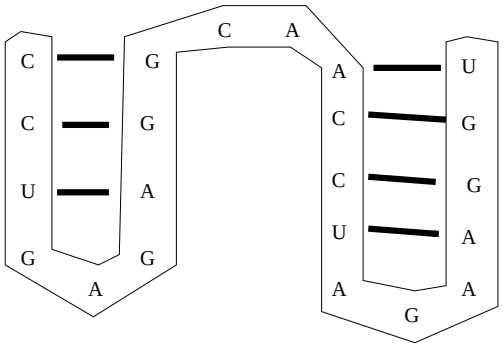

Figure 3: An RNA secondary structure. The thick lines indicate bonds between complementary bases in a stem.

candidates to form base pairs but, sometimes, also other pairs are formed. For more accurate secondary structure prediction the grammar rules need to be augmented with probabilities. The computerized methods used for RNA secondary structure prediction use probabilistic models called *stochastic context-free grammars.* An introductory survey can be found e.g. in [**?**].

References

[1] R. Durbin, S. Eddy, A. Krogh, G. Mitchison, *Biological sequence analysis: Probabilistic models of proteins and nucleic acid.* Cambridge University Press, 1998.