A brief introduction to grammars

Helmuth Trefftz
Distance Education Instrutor
Computer Science Department
Maharishi University of Management
htrefftz@eafit.edu.co

May, 2002

1 Introduction

This document is a very brief description of the formal grammars that are usually utilized to describe the structure of computer programs. Only the basic concepts that serve as foundations for the rest of the CS505 course are covered. A good coverage from the perspective of formal methods can be found in *Programming Language Syntax and Semantics* [Wat91]. The classic textbook on compilers, *Compilers: Principles, Techniques and Tools* [ASU86], covers grammars from the perspective of compiler construction.

2 Grammars

Every language has a set of rules that define when a phrase is well formed. Such set of rules define the *grammar* of the language. A phrase that is not build according to the rules is not well formed. In English, for instance, the phrase

```
The cat is black
```

is well formed. But the phrase

```
eat run star the,
```

despite of being formed with English words, is not a well formed phrase.

Most constructions found in computer languages can be **formally** described using either *context-free grammars* or *regular grammars*. This classification of formal grammars was defined by linguist Noam Chomsky in 1956 [Cho56]. Chomsky defined other two types of grammars, but since those are not usually used for defining computer language constructs, we will not cover them in this document.

We will start with a simple example from natural language. We will define a grammar to define a number of sentences. Our sentences will have a Subject, a Verb and an Object. The subjects can be: `I, The cat, The dog`. The verbs can be: `see, eats`. The objects can be: `a bird, the mountain`.

Using this simple grammar, we can generate phrases such as

```
I see the mountain
```

or

```
The cat eats a bird
```

Note that we can also create "strange" sentences, such as or

```
The cat see the mountain
```

, which is not correct in English language but is a well-formed sentence of our language.

BNF notation [Nau63] , first introduced to describe the language Algol 60, is a convenient way of describing formal grammars. Our simple grammar, described in BNF notation, looks like this:

Sentence	::=	Subject Verb Object
Subject	::=	*I*
Subject	::=	*The cat*
Subject	::=	*The dog*
Verb	::=	*see*
Verb	::=	*eats*
Object	::=	*the mountain*
Object	::=	*a bird*

Note the following elements in the BNF definition of our grammar:

- *Terminal symbols*: symbols that form part of the final phrases (*The cat, the mountain, see,...*).
- *Nonterminal symbols*: symbols used during the generation phase, they never appear in the final phrases (Sentence, Subject, Verb and Object).
- *Start symbol*: we always start generating a new phrase by expanding this symbol (Sentence).
- *Production rules*: each rule that specifies how to *expand* a nonterminal symbol.

3 Context-free grammars

How do we form phrases that belong to the language? We always begin with the *Start symbol*. In our grammar the Start symbol is "Sentence". We replace the Start symbol with the right hand side of any production that starts with "Sentence". In our case there is only one such production, and we replace "Sentence" with "Subject Verb Object". Are we done yet? No, we are not done while there are still *nonterminal* symbols in the phrase. So we continue replacing nonterminal symbols by any right hand side of the corresponding productions. In our case, for instance, we replace "Subject" by the *terminal symbol* "I". We replace "Verb" by the terminal symbol "see" and finally, we replace "Object" by the terminal symbol "the mountain". Now we are done, because there are no more nonterminal symbols in the phrase. The process of repetitive application of productions to form a phrase is called *derivation* of the phrase.

How do we know if a given phrase belongs to the language defined by a grammar? We try to construct a valid derivation for the phrase. If such a derivation exists, the phrase is valid and belongs to the language. Note that there can be no valid derivation, one derivation (and only one) or several derivations for a given phrase. This is further considered in section 3.3

A context-free grammar is a quadruple:

$$G = (\mathbf{T}, \mathbf{N}, S, \mathbf{P})$$

where $\mathbf{T}$ is a set of terminal symbols, $\mathbf{N}$ is a set of nonterminal symbols, S is the start symbol and $\mathbf{P}$ is a set of productions. Each production rule in $\mathbf{P}$ is written in the form $N ::= \alpha$, where $N \in \mathbf{N}$ is a nonterminal symbol and $\alpha \in (\mathbf{T} \cup \mathbf{N})*$ is a string of terminal and nonterminal symbols.

The key issue here is that only a single nonterminal symbol can appear on the left-hand side of any production. On the right-hand side of productions, any string of terminals and nonterminals can be found.

3.1 Syntax trees

A derivation of a phrase can be represented graphically using a *syntax tree*.

Phrase "I see the mountain" of our grammar can be described by the syntax tree in figure 1. The children of a node in the tree are the right hand side of a production that expands that nonterminal. For instance, we start by applying the following production:

Sentence ::= Subject Verb Object

This production corresponds the first branching in the tree, where "Sentence" is the node and "Subject", "Verb" and "Object" are the children.

3.2 Operator precedence

In most computer languages, multiplication and division have precedence over addition and subtraction. This means that $1+2*3$ is evaluated to 7 $(1+(2*3))$,

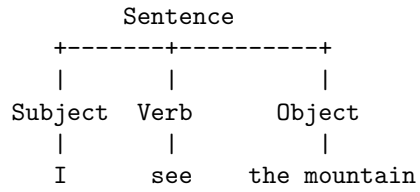

Figure 1: Syntax tree for *I see the mountain*

instead of 9 $((1 + 2) * 3)$.

The order of execution can be easily observed in syntax trees. The syntax tree for $(1+2)*3$ is described in figure 2. Note that $1+2$ is closer to the bottom of the tree. This means that $1 + 2$ is executed first and only then can the result be added to 3. In this case operator $+$ has more precedence than operator $*$.

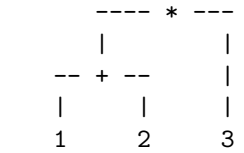

Figure 2: Syntax tree for $(1 + 2) * 3$

The syntax tree for $1 + (2 * 3)$ is described in figure 3. Note that in this case $2 * 3$ is closer to the bottom of the tree. This means that $2 * 3$ is executed first and only then the result is added to 1. In this case, operation $*$ has precedence over operator $+$, despite of appearing later in the phrase.

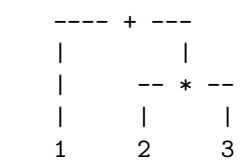

Figure 3: Syntax tree for $1 + (2 * 3)$

By generating different syntax trees, different grammars can enforce – or not – precedence. For instance, the grammar described in figure 4 enforces precedence of * over +, as illustrated by the syntax tree of $1 + 2 * 3$ in figure 5. Note that the grammar is built in a clever way that always pushes multiplicative operations further down in the tree. The effect is that multiplications are executed *before* additions are. It is left to the reader to find the derivation for phrase $1 * 2 + 3$. Note that, despite of the order of appearance in the sentence, multiplications are placed further down in the *syntax tree* and are, therefore, executed before additions are.

Expr	::=	Term
		| Expr + Term
Term	::=	Fact
		| Term * Fact
Fact	::=	*0*
		| *1*
		...
		| *9*

Figure 4: Precedence enforced by the grammar

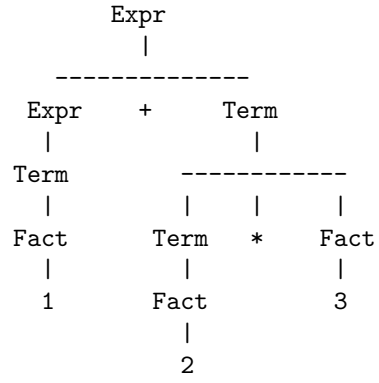

Figure 5: Syntax tree for a grammar that enforces precedence of * over +

Expr	::=	Num
		| Expr + Expr
Num	::=	*0*
		| *1*
		...
		| *9*

Figure 6: Ambiguous grammar

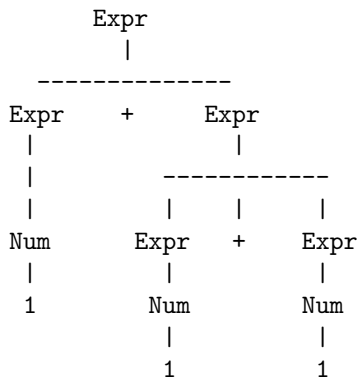

Figure 7: Syntax tree $1 + (1 + 1)$

3.3 Ambiguity

There is a relation between the derivation of a phrase and its meaning, as explained in section 3.2. For this reason it is desirable to have one and only one derivation for each sentence in the language. Otherwise, a sentence could be assigned different meanings.

A grammar is ambiguous if there can be more than one derivation for a given valid sentence. Figure 6, for instance, describes an ambiguous grammar. Sentence $1 + 1 + 1$ can be derived with different trees, as described in figure 8 and figure 7. Note that both derivations of the same sentence are valid. But in the first case, the first addition is executed first. In the second case, the second addition is executed first. Thus, we have proved that grammar 6 is ambiguous.

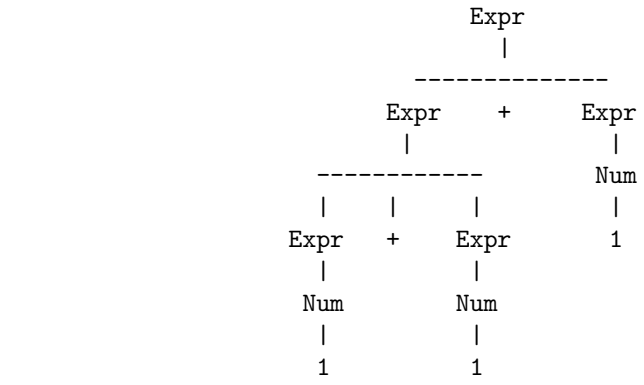

Figure 8: Syntax tree $(1 + 1) + 1$

References

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[Cho56] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, pages 113 – 124, 1956.

[Nau63] P. Naur. Revised report on the algorithmic language algol 60. *Communications of the ACM*, 6:1 – 20, 1963.

[Wat91] D. A. Watt. *Programming Language Syntax and Semantics*. Prentice Hall International, 1991.