CST8152 - Compilers
Article #6
Grammars and Languages

Informal Introduction

Computer languages have much in common with other human languages. Understanding their structure is the key to translating code written in them into an unambiguous set of instructions that can be accurately executed by a computing machine. The structure of a language is defined by its **GRAMMAR**.

Grammars

The informal rules for a syntactically correct English sentence as described in the book "The Language Instinct" are

Who Does What To Whom How Where When

or

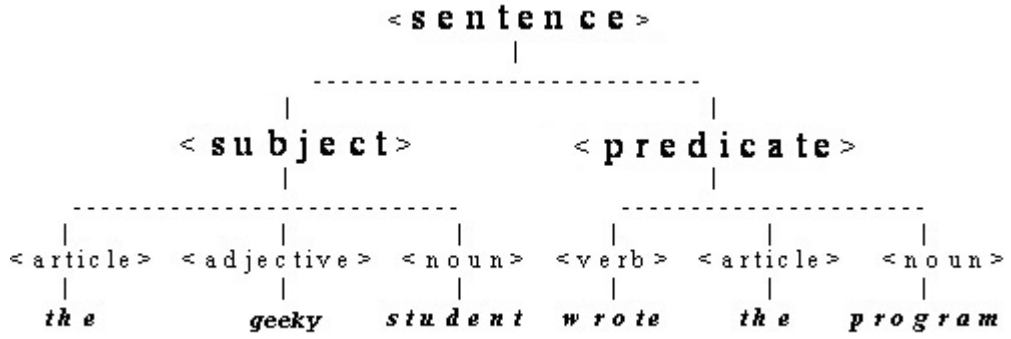
Who Did What To Whom How Where When

This works in every day life, but is not very helpful if we want to develop a translator.

We need some well defined notation allowing us to describe the syntactical rules of a language without any ambiguities. We need some type of metalanguage – a language for describing languages.

Consider the following sentence: "*The geeky student wrote the program*".

In terms of the grammar of the English language this can be broken down into a **hierarchical parse tree**



Note the use of **meta-symbols** like `<subject>` enclosed in <> to denote the **syntactic entities**. These constitute a special language that describes a language. Being a language about a language, it is a **metalanguage**.

The parse tree suggests a form for the metalanguage as a set of rules for the correct ways the syntactic entities may be formed. The following metalanguage is that of Noum Chomsky in a notation proposed by Jonh Backus. Later we will use a more compact form called Backus-Naur Form (**BNF**). The **rules** for the grammar appear to be

```
<sentence>              ::= <subject> <predicate>
<subject>               ::= <article> <adjective> <noun>
<predicate>             ::= <verb> <direct object>
<direct object>         ::= <article> <noun>
<article>               ::= the
<adjective>             ::= geeky
<verb>                  ::= wrote
<noun>                  ::= student
<noun>                  ::= program
```

the metasymbol ::= means “has the form of” or “may be composed of”.
A single rule is called a **production** since what is on the left-hand side can produce a more detailed string on the right-hand side.
the metasymbols enclosed in <> (<sentence> etc.) are called **nonterminals** since they will be replaced by applying the production.
The symbols in bold (**the**, **geeky**, etc.) are called **terminals** since they terminate the syntax (they are the **leaf nodes**). The set of terminal symbols forms the **sentence** that we finally recognize:

“*The geeky student wrote the program*”

An slightly different notation for writing the productions is:

```
<sentence>              → <subject> <predicate>
<subject>               → <article> <adjective> <noun>
<predicate>             → <verb> <direct object>
<direct object>         → <article> <noun>
<article>               → the
<adjective>             → geeky
<verb>                  → wrote
<noun>                  → student
<noun>                  → program
```

BNF allows an abbreviation of alternative productions:

```
<sentence>              → <subject> <predicate>
<subject>               → <article> <adjective> <noun>
<predicate>             → <verb> <direct object>
<direct object>         → <article> <noun>
<article>               → the
<adjective>             → geeky
<verb>                  → wrote
<noun>                  → student | program
```

Languages

A **language** is the set of sentences that are generated from the grammar. So given the grammar above, possible sentences are:

1. "*The geeky student wrote the program*"
2. "*The geeky program wrote the student*"

The second sentence doesn't look right (unless "student" is the name of output of the program) and therefore fails to be **semantically** correct, even though it is syntactically correct according to the grammar.

Scanning and Parsing (Lexical and Syntactic Analysis)

Given a string of characters, how do we know it is a valid sentence according to the grammar? There are two issues here:

1. **lexical analysis (the scanner)** - constructing the string of tokens (terminal symbols) from the string of characters
2. **syntactic analysis (the parser)** –validating (recognizing) the token string against the grammar.

This division is not really so precise since parsing in its general meaning is also part of the scanning process.

1. **In lexical analysis** (the scanner) we attempt to collect the incoming characters into tokens, or terminal symbols of the grammar. So the following string of characters:

"*The geeky student wrote the program*"

generates the tokens **ARTICLE**, **ADJECTIVE**, **VERB**, **NOUN** (represented in the program code as symbolic constants, possibly with the values 0, 1, 2, 3), with their lexical values ***the, geeky, wrote, student, program*** . So when the input stream has been scanned we end with the symbol table containing the entries:

token	attribute - lexeme
ARTICLE	the
ADJECTIVE	geeky
NOUN	student
VERB	wrote
ARTICLE	the
NOUN	program

(In a real programming language, the attribute may be a string of arbitrary length called a **lexeme**, pointed to by the attribute pointer).

However the task of the scanner is also to reject an input string like:

"*Thr gyeke stuent wnote te pgram*"

since no tokens can be identified.

Therefore the task of the scanner in lexical analysis is to parse the input stream of characters, group them (into lexemes) and recognize them as tokens matching the patterns associated with the tokens.

In doing this we will use a simpler grammar for describing the lexemes called a **regular** grammar for which is equivalent to **regular expressions** notation for describing strings (lexemes).

2. **In syntactic analysis**, the role of the parser is to confirm (recognize) that the sentence of tokens satisfies the grammar. This is **parsing the token stream**.

Suppose the scanner has recognized the input
"*The geeky student wrote the program*"
as the string of tokens:
ARTICLE ADJECTIVE NOUN VERB ARTICLE NOUN

The syntactic analyzer (parser) must then confirm that it is valid grammatically by using the productions of the grammar to construct (usually implicitly) a syntax tree. If the tree fails, then the string of tokens is not a sentence of the language.
This can be done by **top-down** or **bottom-up** parsing technique**.**

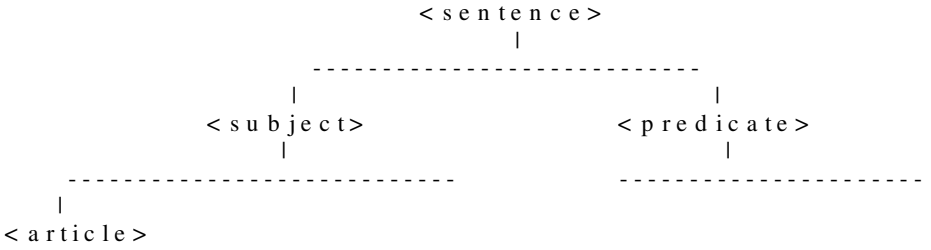
Top-Down parsing
In this we begin with the <sentence> nonterminal, called the **start** symbol, and by successive applications (**derivations**) of the productions attempt to arrive at the sentence. At each stage one token is consumed. Here are the productions, labeled for identification:

<sentence>	→ <subject> <predicate>	1.
<subject>	→ <article> <adjective> <noun>	2.
<predicate>	→ <verb> <direct object>	3.
<direct object>	→ <article> <noun>	4.
<article>	→ **the**	5.
<adjective>	→ **geeky**	6.
<verb>	→ **wrote**	7.
<noun>	→ **student | program**	8.

Starting from the left of the token string (a **leftmost derivation**) we look at the first token **ARTICLE**. The ? indicates our position in the token stream (the token currently being inspected is called the **lookahead**):

ARTICLE ADJECTIVE NOUN VERB ARTICLE NOUN
?

We begin at the start nonterminal <sentence>. We find using rule 1. that a sentence starts with a <subject> that starts with an <article>. So the parse tree at this point is

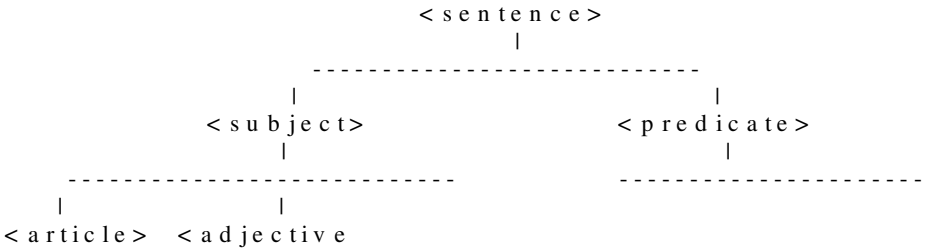


Now that **ARTICLE** has been recognized, it is consumed, the token string is shorter and we are now attempting to recognize **ADJECTIVE**

ADJECTIVE NOUN VERB ARTICLE NOUN

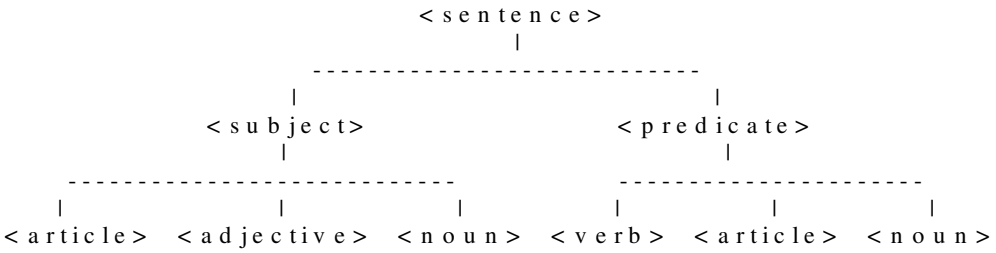
?

We see from rule 2 that a <subject> is an <article> followed by an <adjective>. So **ADJECTIVE** is consumed and the parse tree is:



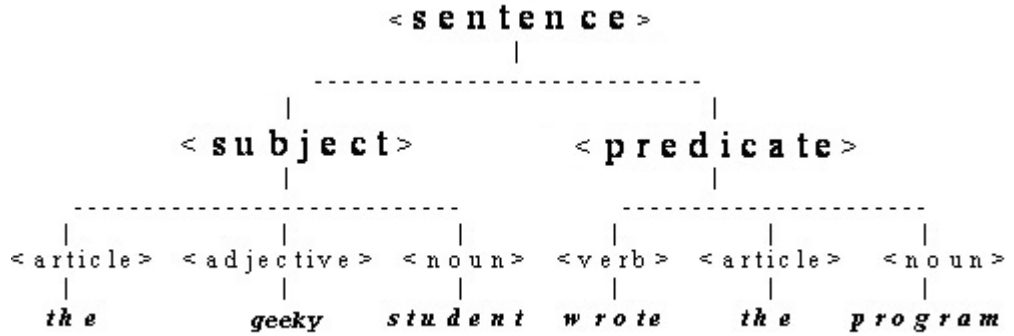
and <adjective> is consumed.

We continue in this way until all the input tokens are consumed and the parse tree builds the token string in order:



The input token string has been successfully parsed and is grammatically valid.

Note that because we used the scanner to do a prior lexical analysis, the actual lexemes do not appear in this tree. If we had included the scanner in the complete process, the complete tree would have been:



Once we have defined the grammar for a sentence, we can define the grammar for a novel (program)

```
<novel>          → <sentences>
<sentences>      → <sentence> <sentences> | <sentence>
```

Since the nonterminal <sentences> appears on both side of the production it means that a <sentence> is **self-describing**. The <sentences> is described in terms of itself and is therefore **recursive**.

Humans express themselves recursively in written, spoken and programming languages. Useful programming languages are also recursive and can use recursive algorithms to implement the parser (**recursive descent parser**). However, recursion will cause problems in the code implementation of the grammar that we will need to solve.

Informal Specification of the Calc Language

The Calc language is a language for programming simple calculations with whole numbers. The Calc program is a sequence of ***assignment statements***. An assignment statement consists of an identifier representing a variable followed by an **=** symbol, followed by an ***arithmetic expression.*** An ***arithmetic expression*** is an infix expression constructed from ***variables***, ***integer literals***, and the ***operators*** *plus* (**+**), *minus* (**-**), *multiplication* (*****), and *division* (/). The arithmetic expression always evaluates to an integer value which is assigned to the variable on the left side of the = sign. If a variable is used in an expression, the variable must have a previously assigned value. Before termination the program prints automatically the values of all variables introduced in the program.

Example of a Calc program:

```
a = 1 * 5
b = a / 3
c = a + b * c / d
```

Output:

```
a  -> 5
b -> 1
c -> 6
```

Lexical Grammar for the Calc Language

<vid> -> <letters>
<letters> -> <letter> | <letters><letter>
<letter > -> a | b | … | z
<num> -> <digits>
<digits> -> <digit> | <digits> <digit>
<digit> -> 0 | 1 | 2 | … | 9
<operators> -> + | - | * | / | =

Syntactical Grammar for the Calc Language (Calc Syntax)

<program> -> <statements>
<statements> -> <statement> <statements> |<statement>
<statement> -> <assignment>
<assignment> -> vid = <expression>
<expression> -> vid
| num
| <expression> + <expression>
| <expression> * <expression>
| <expression> - <expression>
| <expression> / <expression>

The Syntactical Grammar for Lambda Calculus

The syntax for lambda calculus is very simple:

<expression> -> constant
| variable
| (<expression> <expression>)
| (λ variable. <expression>)

Where constants in this grammar are numbers or certain predefined functions like +, -, *, /. Variables are names like *x* or *y*.

Example:

(λ x. (+ 1 x)) or (λ x. + 1 x)

This is a definition of a function that adds 1 to an arbitrary number x. It is called *lambda abstraction a*nd is represented by the 4th production (or rule) in the grammar.
The basic operation of the lambda calculus is the *application* of expressions such as the lambda abstractions. The expression

(λ x. + 1 x) 2

represents the application of the function that adds 1 to x to the constant 2. Lambda calculus provides a reduction rules that permits 2 to be substituted for x in the lambda abstraction and removing the lambda producing the value:

(λ x. + 1 x) 2 => (+1 2) => 3

Reference: Programming Languages, Principles and Practice, 2nd ed., by Kenneth C. Louden .