Grammars: Raw lecture notes

Terence Parr
University of San Francisco
parrt@cs.usfca.edu

Abstract

1. Language

A *language* is just the set of *sentences* that are valid. A sentence is a sequence of symbols or tokens (words). At deeper level, sentences have structure in that words can't be in random order. e.g., *dogs like cats* vs *cats like dogs*.

In Western languages, we have a two level problem. First we have to break up the sequence of characters into vocabulary symbols. For example, here is a simple sentence using Morse code:

```
I like toast
..   .- .. -.- .  - --- .- ... -
I    L  I  K   E  T O   A  S   T
```

The symbols or tokens represent the *vocabulary*. The *syntax* is an abstract notion describing the rules of the sentence whereas a sentence is an instance of that language. By analogy, you can think of the difference between a class definition and an instance of that class.

There are lots of ways to express syntax. Here are two obvious choices:

x, xx, xxx, ...	delineation
one-or-more x's	english

This gets laborious so we tend to use more mathematical notation for compactness:

$\{x^n \mid n > 0\}$	set notation
`x+`	regular expressions (regex)
`'x'+`	ANTLR

It turns out we need a more sophisticated language for describing nested structures within language (such as parenthesized subexpressions).

Q. What is the set notation for "one-or-more x followed by one-or-more y"? What is the set notation for "one-or-more x followed by the same number of y"? What is the set notation for "One x or two y"?

2. CFGs

A *grammar* is a set of rules that describe the set of valid sentences in a language, L. The rules have a very specific format and therefore follow a language, a *metalanguage*. The rules are called *production rules* and say how to generate strings in the language. There are rule names, *non-terminals*, and vocabulary symbols (*terminals* or *tokens*). The rules are of the form

$leftside \rightarrow rightside$

Context free grammar: *CFG* is a grammar where $leftside$ has to be a single non-terminal:

$expression \rightarrow \mathbf{id}$

There can be multiple rules:

$expression \rightarrow \mathbf{integer}$
$expression \rightarrow \mathbf{id}$

Or using the alternation operator:

$expression \rightarrow \mathbf{id}|\mathbf{integer}$

For formal grammars we tend to use single single capital letters for non-terminals in CFG notation:

$E \rightarrow \mathbf{id}$
$E \rightarrow \mathbf{integer}$

Examples

Language $L = \{a, b\}$.

$A \rightarrow \mathbf{a}$
$A \rightarrow \mathbf{b}$

Or, $A \rightarrow \mathbf{a}|\mathbf{b}$

Infinite language: $L = \{a^*\}$:
$A \rightarrow aA$
$A \rightarrow \epsilon$

or

$A \rightarrow aA$
$A \rightarrow$

$L = \{a^+\}$ is described by:

$A \rightarrow aA$
$A \rightarrow a$

draw the RTN

$L = \{a^n b^n | n \geq 1\}$ is CF because we can create a context free grammar to describe it:

$A \rightarrow aAb$
$A \rightarrow ab$

Some languages or non-context-free, such as $L = \{a^n b^n c^n | n \geq 1\}$; there is no way to make sure all three symbols appear the same number of times. Again, note that there is a difference between a language and its specification (the rules we use to define it). In this case, there is no context free grammar that can describe that language. Also note that there are an infinite number of grammars for describing a single language L.

2.1 Formal CFG definition

A CFG grammar $G = (N, T, P, S)$ has elements:

- N is the set of nonterminals (rule names)
- T is the set of terminals (tokens)
- P is the set of productions
- $S \in N$ is the start symbol

$A \in N$	Nonterminal
$a, b, c, d \in T$	Terminal
$X \in (N \cup T)$	Production element
$\alpha, \beta, \delta \in X^*$	Sequence of grammar symbols
$u, v, w, x, y \in T^*$	Sequence of terminals
ϵ	Empty string
\$	End of file "symbol"

The set of all context-free languages is identical to the set of languages accepted by pushdown automata (*PDA*) and all contexts we languages can be parsed in $O(n^3)$ time. ANTLR is technically $O(n^4)$ but performs linearly $O(n)$ in practice.

3. Regular grammars

A regular grammar is a restricted form of a context free grammar that can describe the so-called *regular languages*. You can think of this as being equivalent to what basic regular expressions can describe. You cannot describe nested structures with a regular grammar because there is no way to make the necessary recursive calls. There is no stack.

A regular grammar has a single non-terminal on the left like a CFG, but the right-hand side can only be: empty, a sequence of terminals, a sequence of terminals followed by a non-terminal, but that's it. All regular languages can be recognized by a finite state machine in linear time. These state machines are called NFA/DFA.

Here's an example of a non-regular language: $L = \{a^n b^n | n \geq 1\}$ because we have no memory but $\{a^n b^m | n \geq 0, m \geq 0\}$ is regular. Here is the regular grammar $A \rightarrow a^* b^*$.

draw state machine

4. Derivations

How do we know what the language looks like, the set of sentence, given a grammar? Grammars can both "generate" and "recognize" input sentences (if we create a parser from the grammar). In formal languages, we tend to think of grammars as generating strings and we use special notation to describe how to derive sentences from a grammar.

A *derivation* is a sequence of steps that gradually transforms a grammar symbol (or grammar fragment) into a sentence (sequence of symbols). For example, given grammar $A \rightarrow a$, there is only one derivation: A is transformed to a using rule $A \rightarrow a$. More formally, we use a derivation operator $\Rightarrow$:
$\alpha \Rightarrow \beta$
$\alpha \Rightarrow^* \beta$
$\alpha \Rightarrow^+ \beta$
for arbitrary grammar fragments α and β.

To answer the question "is a sentence in the language described by a grammar," we try to find a derivation from the start symbol of the grammar to that sentence. For example, consider the following grammar:

$A \rightarrow aAb$
$A \rightarrow ab$

The derivation/generation of ab is $A \Rightarrow ab$. The derivation of $aabb$ is $A \Rightarrow aAb \Rightarrow aabb$. We replace symbols on the right-hand side according to the rules of the grammar. We are free to choose any alternative production for a nonterminal when replacing.

There are *leftmost* and *rightmost derivations*.

$S \rightarrow \textbf{if}\ E\ \textbf{then}\ S$
$S \rightarrow \textbf{return}\ E$
$E \rightarrow \textbf{id}$

$S \Rightarrow_{lm} \textbf{if}\ E\ \textbf{then}\ S$
$\Rightarrow_{lm} \textbf{if x then}\ S$
$\Rightarrow_{lm} \textbf{if x then return}\ E$
$\Rightarrow_{lm} \textbf{if x then return y}$
or
$S \Rightarrow_{rm} \textbf{if}\ E\ \textbf{then}\ S$
$\Rightarrow_{rm} \textbf{if}\ E\ \textbf{then return}\ E$
$\Rightarrow_{rm} \textbf{if}\ E\ \textbf{then return y}$
$\Rightarrow_{rm} \textbf{if x then return y}$

We get different derivations but we get the **same tree**. Show parse tree of A then if-then-else.

Formally, the language generated by grammar fragment α is: $L(\alpha) = \{w \mid \alpha \Rightarrow^* w\}$ and the language of grammar G is $L(G) = \{w \mid S \Rightarrow^* w\}$. Language L is CF *iff* there exists a CFG for L.

α is *sentential form* (grammar fragment) if S derives to it.

Proof that G gens L means show every string gen'd by G is in L and every string in L can be gen'd by G.

5. Parse trees

A parse tree is just a record of the replacements made in a derivation. Grab the derivation by the start symbol and give it a big shake. Show parse tree of `aaa` given $A \rightarrow aA|\epsilon$.

Q. What does parse tree look like for `aab` given $A \rightarrow aA|b$.

6. Ambiguity

See section 5.4 in ANTLR 4 book. p69 in printed version.

"You can't put too much water in a nuclear reacto."

"I once shot an elephant in my pajamas. How he got in my pajamas I will never know." — Groucho Marx

If there is more than one lm or rm derivation for same input, normally it's an error in computer languages. L is syntactically ambiguous in this case; e.g., expressions. Also note that an ambiguous language L gives ambig G. For example, in the language C++ `T(i)` is both a typecast and a function call so it is syntactically ambiguous. Expressions are the typical example used to describe ambiguous grammars:

$E \rightarrow E * E$
$E \rightarrow E + E$
$E \rightarrow \textbf{id}$

`1+2*3` has two interps. show trees. The parser must recognize `1*2+3` as `(1*2)+3` not `1*(2+3)`.

One way to resolve is to convert to non-Left-recursive version: $E \rightarrow \textbf{id}\ (*\ E|+\ E)^*$. The other way is to order the alternatives, which is what ANTLR does to show precedence.

Q. Show the parse tree for `1+2+3` using the non-left recursive grammar $E \rightarrow \textbf{id}\ (*\ E|+\ E)^*$. Now show it for `1+2*3`. Do you think it represents the precedence of the operators correctly?

Q. Modify the expression language to allow **int** integer values. Modify the expression language to allow nested parenthesis. Modify it so it allows array references.

7. Left recursion

Left recursion occurs when a derivation of A yields another string with A on the left edge. $\exists A \Rightarrow^* A\alpha$

Indirectly left-recursive rules call themselves through another rule; e.g., $A \rightarrow B$, $B \rightarrow A$. Hidden left-recursion occurs when an empty production exposes left recursion; e.g., $A \rightarrow BA$, $B \rightarrow \epsilon$. Here's an obvious case of direct left recursion:
$A \rightarrow Aa$
$A \rightarrow b$

Here is an indirect left recursion case:

$A \rightarrow Ba \mid b$
$B \rightarrow A$

Q. What sentences does this grammar generate?
Here is right recursion:

$A \rightarrow aA$
$A \rightarrow b$

8. Left factoring

$S \rightarrow \textbf{if}\, E \,\textbf{then}\, S \,\textbf{else}\, S$
$S \rightarrow \textbf{if}\, E \,\textbf{then}\, S$

$S \rightarrow \textbf{if}\, E \,\textbf{then}\, S S'$
$S' \rightarrow \textbf{else}\, S$
$S' \rightarrow$

or EBNF notation:

$S \rightarrow \textbf{if}\, E\, \textbf{then}\, S\, (\textbf{else}\, S)?$

9. EBNF

That example introduces us to extended BNF (EBNF). We typically use a variation on `yacc` notation, which flips things so that non-terminals are lowercase and terminals are uppercase like constants:

```
a : A* B* ; // extended; yacc doesn't allow *
```

or

```
a : 'a'* 'b'* ; // ANTLR notation
```

```
grammar Ex; // generates class ExParser
// action defines ExParser member: enum_is_keyword
@members {boolean enum_is_keyword = true;}
stat: expr '=' expr ';' // production 1
    | expr ';'          // production 2
    ;
expr: expr '*' expr
    | expr '+' expr
    | expr '(' expr ')' // f(x)
    | id
    ;
id  : ID | {!enum_is_keyword}? 'enum' ;
ID  : [A-Za-z]+ ; // match id with upper, lowercase
WS  : [ \t\r\n]+ -> skip ; // ignore whitespace
```

10. Designing grammars

See chapter 5 in ANTLR 4 reference guide.