

What is *parsing*

- **Parsing** is the process of analyzing an input sequence in order to determine its grammatical structure with respect to a given formal grammar.
- It is formally named **syntax analysis**.
- A **parser** is a computer program that carries out this task.

[from: <http://en.wikipedia.org/wiki/Parsing>]

The process

- **Lexical analysis:**
 - the input character stream is split into meaningful symbols (tokens) defined by a grammar of regular expressions
 - Example: the lexical analyzer takes "12*(3+4)^2" and splits it into the tokens 12, *, (, 3, +, 4,), ^ and 2.
- **Syntax analysis,**
 - checking if the tokens form an legal expression, w.r.t. a CF grammar
 - Limitations - cannot check (in a programming language): types or proper declaration of identifiers
- **Semantic parsing**
 - works out the implications of the expression validated and takes the appropriate actions.
 - Examples: an interpreter will evaluate the expression, a compiler, will generate code.

Type of parsers

- **Top-down parser**
 - start with the start symbol
 - try to transform it to the input
 - the parser starts from the largest elements and breaks them down into incrementally smaller parts
 - **LL parsers** are examples of top-down parsers
- **Bottom-up parser**
 - start with the input,
 - attempt to rewrite it to the start symbol
 - the parser attempts to locate the most basic elements, then the elements containing these, and so on.

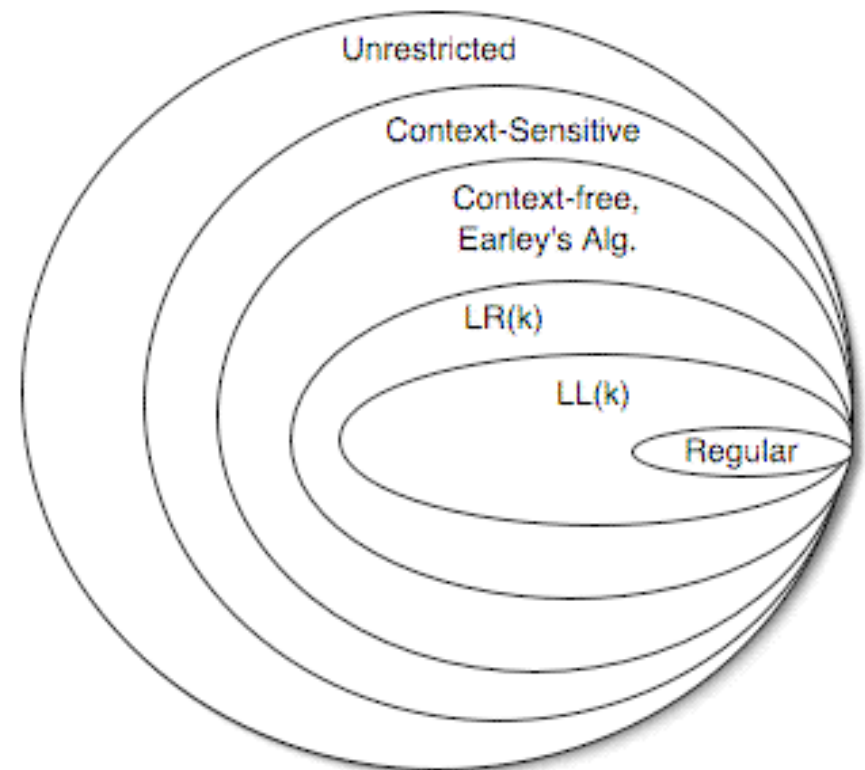
Note: LL is based on leftmost derivation

Recursive descent parser

- A **recursive descent parser** is a top-down parser
 - built from a set of mutually-recursive procedures
 - where each procedure usually implements one of the production rules of the grammar
- A **predictive parser** is a recursive descent parser that never backtracks
 - can be done only for a subset of CF grammars: LL(k)

[from:

<http://lambda.uta.edu/cse5317/notes/node13.html>]



Grammar Hierarchy

Picture from:

<http://www.cs.usfca.edu/~parrt/course/652/lectures/grammar.hierarchy.gif>

1. Simplify the grammar

- Transform a grammar in two ways:
 - eliminate left recursion
 - perform left factoring

and you have a LL(1) grammar

- Example: $A \rightarrow A a \mid b$, that denotes the language $L(b.a^*)$

becomes: $A ::= b A'$

$$A' ::= a A' \mid \epsilon$$

- Suppose you are parsing “baaa”, you know you need:
 - rule 1, to generate the “b”
 - rule 2 to generate the “a” (3 times)
 - rule 3 to end the process

$A ::= b A'$

$A' ::= a A' \mid \text{epsilon}$

2. How to build the parser

- For each nonterminal write one procedure
- For each nonterminal in the r.h.s. of a rule, call nonterminal's procedure
- For each terminal, compare the current token with the expected terminal
 - if there are multiple productions for a nonterminal, use if-then-else to choose which rule to apply
- *if there was a left recursion in a production, we would have had an infinite recursion*

$A ::= b A'$

$A' ::= a A' \mid \text{epsilon}$

3. Define the parser

The program looks like:

```
void A() {  
    if (current_token == SYMBOL_B) {  
        read_next_token(); Aprime();  
    }  
}  
  
void Aprime() {  
    if (current_token == SYMBOL_A) {  
        read_next_token(); Aprime();  
    }  
}
```

with some actions in between to create a tree.

Example with a more complex grammar

Another transformation is called **left factoring**. It helps predict which rule to use without backtracking

$E ::= T + E$

| $T - E$

| T

is transformed into:

$E ::= T E'$

$E' ::= + E$

| $- E$

| *epsilon*

Links

- Building Top-Down Parsers
<http://www.cs.uky.edu/~lewis/essays/compilers/td-parse.html>
- A Simple Recursive-descent Parser (in java)
<http://www.faqs.org/docs/javap/c11/s5.html>
- Recursive Descent Parsing
<http://www-ist.massey.ac.nz/159357/Recdesent.pdf>