# Syntax Analysis

## MIF08

Laure Gonnord

Laure.Gonnord@univ-lyon1.fr

Lyon 1

# Goal of this chapter

- Understand the syntaxic structure of a language;
- Separate the different steps of syntax analysis;
- Be able to write a syntax analysis tool for a simple language;
- Remember: syntax$\neq$semantics.

# Syntax analysis steps

How do **you** read text ?

- Text=a sequence of symbols (letters, spaces, punctuation);
- Group symbols into tokens:
  - Words: groups of letters;
  - Punctuation;
  - Spaces.

## Syntax analysis steps

How do **you** read text ?

- Text=a sequence of symbols (letters, spaces, punctuation);
- Group symbols into tokens: **Lexical analysis**
  - Words: groups of letters;
  - Punctuation;
  - Spaces.

# Syntax analysis steps

How do **you** read text ?

- Text=a sequence of symbols (letters, spaces, punctuation);
- Group symbols into tokens: **Lexical analysis**
  - Words: groups of letters;
  - Punctuation;
  - Spaces.
- Group tokens into:
  - Propositions;
  - Sentences.

# Syntax analysis steps

How do **you** read text ?

- Text=a sequence of symbols (letters, spaces, punctuation);
- Group symbols into tokens: **Lexical analysis**
  - Words: groups of letters;
  - Punctuation;
  - Spaces.
- Group tokens into: **Parsing**
  - Propositions;
  - Sentences.

# Syntax analysis steps

How do **you** read text ?

- Text=a sequence of symbols (letters, spaces, punctuation);
- Group symbols into tokens: **Lexical analysis**
  - Words: groups of letters;
  - Punctuation;
  - Spaces.
- Group tokens into: **Parsing**
  - Propositions;
  - Sentences.
- Then proceed with word meanings:
  - Definition of each word.
    ex: a dog is a hairy mammal, that barks and...
  - Role in the phrase: verb, subject, ...

# Syntax analysis steps

How do **you** read text ?

- Text=a sequence of symbols (letters, spaces, punctuation);
- Group symbols into tokens: **Lexical analysis**
  - Words: groups of letters;
  - Punctuation;
  - Spaces.
- Group tokens into: **Parsing**
  - Propositions;
  - Sentences.
- Then proceed with word meanings: **Semantics**
  - Definition of each word.
    ex: a dog is a hairy mammal, that barks and...
  - Role in the phrase: verb, subject, ...

# Syntax analysis steps

How do **you** read text ?

- Text=a sequence of symbols (letters, spaces, punctuation);
- Group symbols into tokens: **Lexical analysis**
  - Words: groups of letters;
  - Punctuation;
  - Spaces.
- Group tokens into: **Parsing**
  - Propositions;
  - Sentences.
- Then proceed with word meanings: **Semantics**
  - Definition of each word.
    ex: a dog is a hairy mammal, that barks and...
  - Role in the phrase: verb, subject, ...

Syntax analysis=Lexical analysis+Parsing

# Outline

1. Lexical Analysis aka Lexing
2. Parsing

# What for ?

```
int y = 12 + 4*x ;
```

⟹ [TINT, VAR("y"), EQ, INT(12), PLUS, INT(4), FOIS, VAR("x"), PVIRG]

- Group characters into a list of **tokens**, e.g.:
  - The word “int” stands for *type integer*;
  - A sequence of letters stands for a *variable*;
  - A sequence of digits stands for an *integer*;
  - ...

# What's behind

From a Regular Language, produce a Finite State Machine (see **LIF15**)

# Tools: lexical analyzer constructors

- Lexical analyzer constructor: builds an automaton from a regular language definition;
- Ex: Lex (C), JFlex (Java), OCamllex, **ANTLR** (multi), ...
- **input**: a set of regular expressions with actions (`Toto.g4`);
- **output**: a file(s) (`Toto.java`) that contains the corresponding automaton.

# Analyzing text with the compiled lexer

- The **input of the lexer** is a text file;
- Execution:
  - Checks that the input is accepted by the compiled automaton;
  - Executes some actions during the “automaton traversal”.

# Lexing tool for Java: ANTLR

- The official webpage : `www.antlr.org` (BSD license);
- ANTLR is both a lexer and a parser;
- ANTLR is multi-language (not only Java).

► During the labs; we will use the Python back-end (here, demo in java)

# ANTLR lexer format and compilation

## .g4

```
grammar XX;

@header {
// Some init code...
}

@members {
// Some global variables
}
// More optional blocks are available

--->> lex rules
```

Compilation:

```
antlr4 Toto.g4        // produces several Java files
javac *.java          // compiles into xx.class files
grun Toto r           // Run analyzer with starting rule r
```

# Lexing with ANTLR: example

Lexing rules:

- Must start with an upper-case letter;
- Follow the usual extended regular-expressions syntax (same as egrep, sed, ...).

## A simple example

```
grammar Hello;

// This rule is actually a parsing rule
r  : HELLO ID ; // match "hello" followed by an identifier

HELLO : 'hello' ;         // beware the single quotes
ID : [a-z]+ ;             // match lower-case identifiers
WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
```

# Lexing - more than regular languages

## Counting in ANTLR - CountLines.g4

```
lexer grammar CountLines;

// Members can be accessed in any rule
@members {int nbLines=0;}

NEWLINE : [\r\n] {
  nbLines++;
  System.out.println("Current lines:"+nbLines);
} ;

SK : ([a-z]+|[ \t]+) -> skip ;
```

```
antlr4 Toto.g4          // produces several Java files
javac *.java            // compiles into xx.class files
grun Toto 'tokens'      // Run the lexical analyser only
```

# Outline

1. Lexical Analysis aka Lexing

2. Parsing
   - Semantic actions / Attributes

# What's Parsing ?

Relate tokens by structuring them.

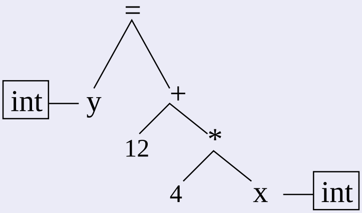
## Flat tokens

[TINT, VAR("y"), EQ, INT(12), PLUS, INT(4), FOIS, VAR("x"), PVIRG]

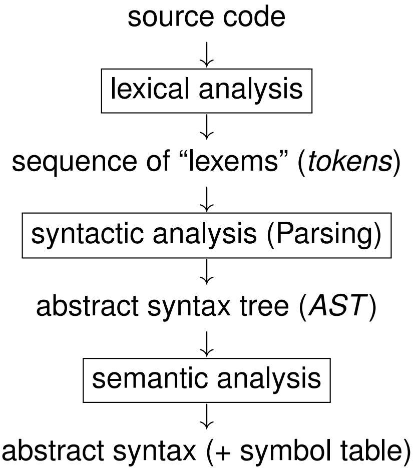
⇒ **Parsing** ⇒

Yes/No +

## Structured tokens



# Analysis Phases



source code

↓

lexical analysis

↓

sequence of “lexems” (*tokens*)

↓

syntactic analysis (Parsing)

↓

abstract syntax tree (*AST*)

↓

semantic analysis

↓

abstract syntax (+ symbol table)

# What's behind ?

From a Context-free Grammar, produce a Stack Automaton (see **LIF15**).

# Tools: parser generators

- Parser generator: builds a stack automaton from a grammar definition;
- Ex: yacc(C), javacup (Java), OCamlyacc, **ANTLR**, ...
- **input** : a set of grammar rules with actions (`Toto.g4`);
- **output** : a file(s) (`Toto.java`) that contains the corresponding stack automaton.

# Lexing vs Parsing

- Lexing supports ($\simeq$ regular) languages;
- We want more (general) languages $\Rightarrow$ rely on context-free grammars;
- To that intent, we need a way:
  - To declare terminal symbols (**tokens**);
  - To write grammars.

▶ Use both Lexing rules and Parsing rules.

# From a grammar to a parser

The grammar must be **context-free**:

```
S-> aSb
S-> eps
```

- The grammar rules are specified as **Parsing rules**;
- $a$ and $b$ are terminal tokens, produced by Lexing rules.

On board: notion of derivation tree (see also exercise session2)

# Parsing with ANTLR: example 1/2

## AnBnLexer.g4

```
lexer grammar AnBnLexer;

// Lexing rules: recognize tokens
A: 'a' ;
B: 'b' ;

WS : [ \t\ r\n ]+ -> skip ; // skip spaces, tabs, newlines
```

# Parsing with ANTLR: example 2/2

### AnBnParser.g4

```
parser grammar AnBnParser;
options {tokenVocab=AnBnLexer;} // extern tokens definition

// Parsing rules: structure tokens together
prog : s EOF ; // EOF: predefined end-of-file token
s : A s B
    | ;  // nothing for empty alternative
```

# ANTLR expressivity

LL(*)

> *At parse-time, decisions gracefully throttle up from conventional fixed $k \geq 1$ lookahead to arbitrary lookahead.*

Further reading (PLDI'11 paper, T. Parr, K. Fisher)

`http://www.antlr.org/papers/LL-star-PLDI11.pdf`

# Left recursion

ANTLR permits left recursion:

```
a: a b;
```

But not indirect left recursion.

$$\begin{tikzcd} X_1 \arrow[r] & \ldots \arrow[r] & X_n \arrow[ll, bend right] \end{tikzcd}$$

There exist algorithms to eliminate indirect recursions.

# Lists

ANTLR permits lists:

```
prog: statement+ ;
```

Read the documentation!

https://github.com/antlr/antlr4/blob/master/doc/index.md

# Outline

1. Lexical Analysis aka Lexing
2. Parsing
   - Semantic actions / Attributes

# Semantic actions

**Semantic actions**: code executed each time a grammar rule is matched.

### Printing as a semantic action in ANTLR

```
s : A s B { System.out.println("rule s"); }

s : A s B { print("rule s"); }//python
```

Right rule : Python/Java/C++, depending on the back-end

```
antlr4 -Dlanguage=Python2
```

- We can do more than acceptors.

# Semantic actions - attributes

**An attribute** is a set attached to non-terminals/terminals of the grammar

They are usually of two types:

- synthetized: sons $\rightarrow$ father.
- inherited: the converse.

# Semantic attributes for numerical expressions 1/2

$$\begin{array}{rcll} e & ::= & c & \textit{constant} \\ & | & x & \textit{variable} \\ & | & e + e & \textit{add} \\ & | & e \times e & \textit{mult} \\ & | & \ldots & \end{array}$$

Let’s come to an attribution. On board.

# Semantic attributes 2/2 : Implem

Implementation of the former actions (java):

## ArithExprParser.g4

```
parser grammar ArithExprParser;
options {tokenVocab=ArithExprLexer;}

prog : expr EOF { System.out.println("Result: "+$expr.val); } ;

expr returns [ int val ] : // expr has an integer attribute
  LPAR e=expr RPAR { $val=$e.val; }
| INT { $val=$INT.int; } // implicit attribute for INT
| e1=expr PLUS e2=expr // name sub-parts
  { $val=$e1.val+$e2.val; } // access attributes
| e1=expr MINUS e2=expr { $val=$e1.val-$e2.val; }
;
```