Lexing, Parsing

Laure Gonnord

http://laure.gonnord.org/pro/teaching/capM1.html

Laure.Gonnord@ens-lyon.fr

Master 1, ENS de Lyon

sept 2017

Analysis Phase

source code

↓

lexical analysis

↓

sequence of “lexems” (*tokens*)

↓

syntactic analysis (Parsing)

↓

abstract syntax tree (*AST*)

↓

semantic analysis

↓

abstract syntax (+ symbol table)

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

1. Lexical Analysis
   - Principles
   - Tools
2. Syntactic Analysis
3. Abstract Syntax Tree and evaluators

What for ?

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

$\Longrightarrow$ [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* ;
  - ...

Principle

- Take a lexical description : $E = (\underbrace{E_1}_{\text{Tokens class}} | \ldots | E_n)^*$
- Construct an automaton.

Example - lexical description (“lex file”)

$E = ((0|1)^+|(0|1)^+.(0|1)^+|'+')^*$

What's behind

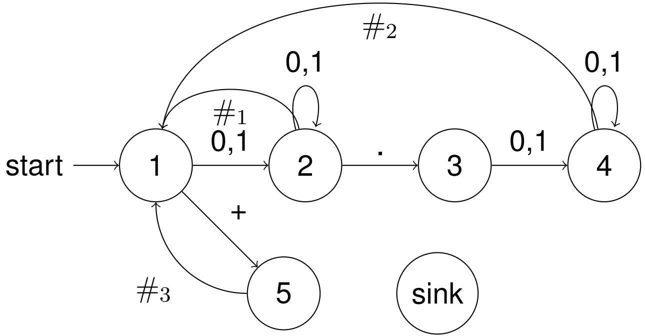
Regular languages, regular automata :

- Thompson construction ▶ non-det automaton
- Determinization, completion
- Minimisation

Construction/use of automaton

Let $E = (\underbrace{(0|1)^+}_{\#_1} | \underbrace{(0|1)^+.(0|1)^+}_{\#_2} | \underbrace{'+'}_{\#_3})^*$ and $\Sigma = \{0, 1, +, ., \#_1, \#_2, \#_3\}$.

We define $E_\# = ((0|1)^+\#_1|(0|1)^+.(0.1)^+\#_2|' +' \#_3)$.



Source C. Alias drawn by former M1 students

Remarks

- Notion of ambiguity.
- Compaction of transition table.

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).

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

```
grammar CountLines;

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

r   : (NEWLINE)* ;

NEWLINE : [\r\n] {
  nbLines++;
  System.out.println("Current lines:"+nbLines);} ;
WS : [ \t]+ -> skip ;
```

What's Parsing ?

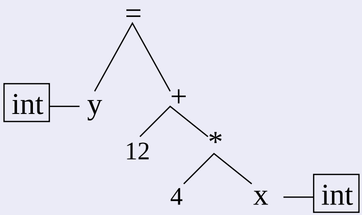
Relate tokens by structuring them.

Flat tokens

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

$\Rightarrow$ **Parsing** $\Rightarrow$

Accept $\rightarrow$ Structured tokens



In this course

Only write acceptors or a little bit more.

What's behind ?

From a Context-free Grammar, produce a Stack Automaton
(already seen in L3 course ?)

Recalling grammar definitions

Grammar

A **grammar** is composed of :

- A finite set N of non terminal symbols
- A finite set Σ of terminal symbols (disjoint from N)
- A finite set of production rules, each rule of the form $w \to w'$ where w is a word on $\Sigma \cup N$ with at least one letter of N. w' is a word on $\Sigma \cup N$.
- A start symbol $S \in N$.

Example

Example :

$$S \rightarrow aSb$$

$$S \rightarrow \varepsilon$$

is a grammar with $N = ...$ and ....

Associated Language

Derivation

G a grammar defines the relation :

$$x \Rightarrow_G y \text{ iff } \exists u, v, p, q x = upv \text{ and } y = uqv \text{ and } (p \rightarrow q) \in P$$

- A grammar describes a **language** (the set of words on Σ that can be derived from the start symbol).

Example - associated language

$$S \to aSb$$

$$S \to \varepsilon$$

The grammar defines the language $\{a^n b^n, n \in \mathbf{N}\}$

$$S \to aBSc$$

$$S \to abc$$

$$Ba \to aB$$

$$Bb \to bb$$

The grammar defines the language $\{a^n b^n c^n, n \in \mathbf{N}\}$

Context-free grammars

Context-free grammar

A **CF-grammar** is a grammar where all production rules are of the form $N \to (\Sigma \cup N)^*$.

Example :

$$S \to S + S | S * S | a$$

The grammar defines a language of arithmetical expressions.

- Notion of **derivation tree**.

Draw a derivation tree of a*a+a, of S+S !

Recognizing grammars

Grammar type	Rules	Decidable
Regular grammar	$X \to aY, X \to b$	YES
Context-free grammar	$X \to u$	YES
Context-sensitive grammar	$uXv \to uwv$	YES
General grammar	$u \to v$	NO

Parser construction

There exists algorithms to recognize class of grammars :

- Predictive (descending) analysis (LL)
- Ascending analysis (LR)

▶ See the Dragon book.

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.

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* $k \geqslant 1$ *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.

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

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

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"); }//java

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.

Computing attributes with semantic actions

ArithExprParser.g4 - Warning this is java

```
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; }
;
```

So Far ...

ANTLR has been used to :

- Produce acceptors for context-free languages ;
- Do a bit of computation on-the-fly.

⇒ In a classic compiler, parsing produces an **Abstract Syntax Tree**.

▶ Next course !