Lexing, Parsing (CAP+MIF08)

Laure Gonnord & Matthieu Moy & other
https://compil-lyon.gitlabpages.inria.fr/

Master 1, ENS de Lyon et Dpt Info, Lyon1

2021-2022

Analysis Phase

source code

$\downarrow$

$\boxed{\text{lexical analysis}}$

$\downarrow$

sequence of “lexems” (*tokens*)

$\downarrow$

$\boxed{\text{syntactic analysis (Parsing)}}$

$\downarrow$

abstract syntax tree (*AST*)

$\downarrow$

$\boxed{\text{semantic analysis}}$

$\downarrow$

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

What for ?

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

$\Longrightarrow$ [TINT, VAR("y"), EQ, INT(12), PLUS, INT(4), TIMES, VAR("x"), SCOL]

- 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;
  - ...

1. Lexical Analysis
   - Principles
   - Tools

2. Syntactic Analysis
   - Principles
   - Tools

Principle

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

▶ And non trivial algorithmic issues (remove ambiguity, compact the transition table).

todo pour Christophe

trucs spécifiques esisar

1. Lexical Analysis
   - Principles
   - Tools
2. Syntactic Analysis
   - Principles
   - Tools

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

```
lexer grammar XX;
@header {
// Some init code...
}
@members {
// Some global variables
}
// More optional blocks are available
--->> lex rules
```

Compilation (using the java backend)

```
antlr4 Toto.g4          // produces several Java files
javac *.java            // compiles into xx.class files
java org.antlr.v4.gui.TestRig Toto tokens
```

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

```
lexer grammar Tokens;


HELLO : 'hello' ; // beware the single quotes
ID : [a-z]+ ; // match lower-case identifiers
INT : [0-9]+ ;
KEYWORD : 'begin' | 'end' | 'for' ; // perhaps this should be
     elsewhere
WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
```

Lexing - We can count!

Counting in ANTLR - CountLines2.g4

```
lexer grammar CountLines2;

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

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

1. Lexical Analysis
   - Principles
   - Tools
2. Syntactic Analysis
   - Principles
   - Tools

What's Parsing ?

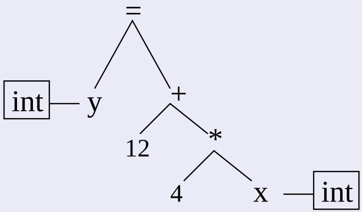
Relate tokens by structuring them.

Flat tokens

[TINT, VAR("y"), EQ, INT(12), PLUS, INT(4), TIMES, VAR("x"), SCOL]

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

Accept $\rightarrow$ Structured tokens



Analysis Phase

source code

$\downarrow$

$\boxed{\text{lexical analysis}}$

$\downarrow$

sequence of “lexems” (*tokens*)

$\downarrow$

$\boxed{\text{syntactic analysis (Parsing)}}$

$\downarrow$

abstract syntax tree (*AST*)

$\downarrow$

$\boxed{\text{semantic analysis}}$

$\downarrow$

abstract syntax (+ symbol table)

In this course

Only write acceptors “OK” or “Syntax Error”.

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:

$$S \to aSb$$

$$S \to \varepsilon$$

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**.

Exercise: draw a derivation tree of a*a+a (with the previous grammar).

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 then Parsing

Concretely, we need a way:

- To declare terminal symbols (**tokens**);
- To write grammars.

▶ Use both Lexing rules and Parsing rules.

Parsing with ANTLR: example

$$S \to aSb$$

$$S \to \varepsilon$$

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

Parsing with ANTLR: example (cont')

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 (cont')

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 {System.out.println("rule S");}
    | ; // 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. $X_1 \rightarrow \ldots \rightarrow X_n$

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

So Far ...

ANTLR has been used to:

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

$\Rightarrow$ In a classic compiler, parsing produces an **Abstract Syntax Tree**.

▶ Next course!