Compilation and Program Analysis (#2) : Lexing, Parsing

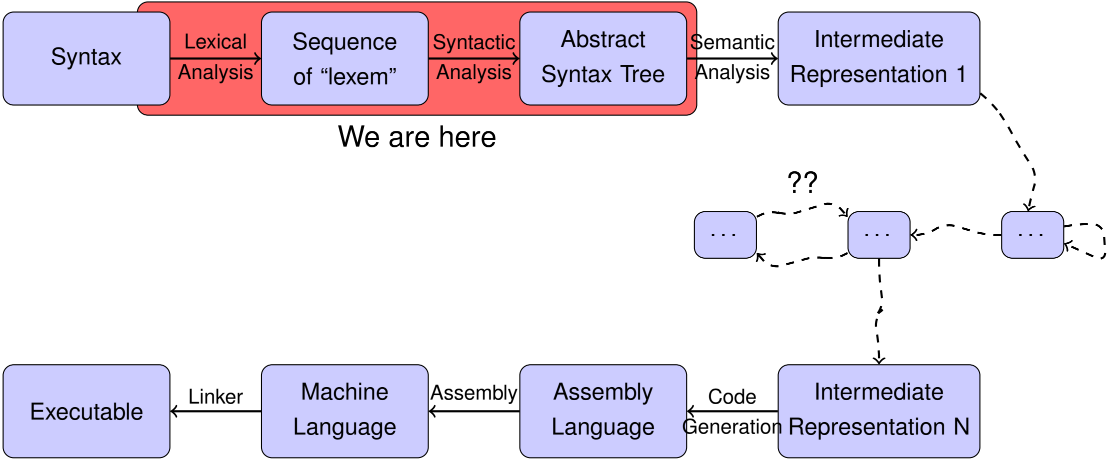
Laure Gonnord & Matthieu Moy & Gabriel Radanne & other

https://compil-lyon.gitlabpages.inria.fr/

Master 1, ENS de Lyon et Dpt Info, Lyon1

2023-2024

A Standard™ Compiler Pipeline



Goal of this chapter

- Understand the syntactic 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

What for ?

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

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

- Group characters into a list of **tokens**, e.g.:
  - The word “int” stands for type integer (predefined identifier in most languages, keyword here);
  - A sequence of letters stands for a identifier (typically, a variable);
  - A sequence of digits stands for an integer literal;
  - ...

1. Lexical Analysis
   - Principles
   - Tools

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

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

1. Lexical 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** of, e.g. ANTLR: a set of regular expressions with actions (`Toto.g4`);
- **output** of ANTLR: the lexer, a file (`Toto.java`) that contains the corresponding automaton (input of the lexer = program to compile, output = sequence of tokens)

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 generator;
- ANTLR is multi-language (not only Java).

Lexing with ANTLR: example

Lexing rules:

- Must start with an upper-case letter;
- Follow 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
```

Running an ANTLR lexer (for debug)

Compilation (using the java backend)

```
$ antlr4 Toto.g4        # produces several Java files
$ javac *.java          # compiles into xx.class files
$ echo 'foo bar hello 42' | \
    java org.antlr.v4.gui.TestRig Tokens tokens -tokens
[@0,0:2='foo',<ID>,1:0]
[@1,4:6='bar',<ID>,1:4]
[@2,8:12='hello',<'hello'>,1:8]
[@3,14:15='42',<INT>,1:14]
[@4,17:16='<EOF>',<EOF>,2:0]
```

Lexer rules: quick reference

`NAME : ... ;`	: rule definition (upper case for lexer)
`(...)`	: grouping
`|`	: alternative, e.g. `(a|b)`
`'s'`	: char or string literal. `'\n'` for newline.
`.`	: any character
`a .. b`	: range, e.g. `('0'..'9')`
`{...}`	: action
`+`	: 1 or more, e.g. `('0' .. '9')+`
`*`	: 0 or more
`?`	: optional (or semantic predicate)
`[...]`	: choice between characters, e.g. `[abc]`
`~[...]`	: match not, e.g. `~[abc]`
`// ...`	: single-line comment
`/* ... */`	: multi-line comment

Lexer rules : dealing with ambiguities

A grammar with ambiguities

```
FLOAT: [0-9]+ '.' [0-9]+;
INT: [0-9]+;
DOT: '.';

IF: 'if';
ID: [a-zA-Z]+;
THEN: 'then';
```

Ambiguities:

4.2	`INT`, `DOT`, `INT` or `FLOAT` ?
if	`ID` or `IF` ?
then	`ID` or `THEN` ?

Two rules to resolve ambiguities (with ANTLR, and most lexer generators) :

1. Longest match;
2. In case of tie, first rule in the grammar file.

Lexer rules : dealing with ambiguities

A grammar with ambiguities

```
FLOAT: [0-9]+ '.' [0-9]+;
INT: [0-9]+;
DOT: '.';

IF: 'if';
ID: [a-zA-Z]+;
THEN: 'then';
```

Ambiguities:

4.2 `INT`, `DOT`, `INT` or `FLOAT` ?

if `ID` or `IF` ?

then `ID` or `THEN` ?

Two rules to resolve ambiguities (with ANTLR, and most lexer generators) :

1. Longest match;
2. In case of tie, first rule in the grammar file.

Actions in an ANTLR lexer

- Basic flow: g4 → Java/Python (“host code”) → Execution
- Alternative: embed host code into g4 file

Foo.g4

```
lexer grammar XX;
@header { // Some init (host) code...
}
@members { // Some global (host) variables
}
// More optional blocks are available
// rules with actions: code within {...} is
// inserted in the generated host code and
// executed when matching.
FOO : 'foo' {System.out.println("foo found");
     };
```

Compilation (using the java backend):

```
$ antlr4 Foo.g4
$ javac *.java
$ java org.antlr.v4.gui.TestRig \
       Foo tokens
```

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 ;
```

2 Syntactic Analysis

- Principles
- Tools

What's Parsing ?

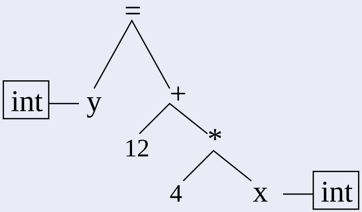
Relate tokens by structuring them.

Flat tokens

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

⇒ **Parsing** ⇒

Accept → Structured tokens



For now

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

What's behind ?

From a Context-free Grammar, produce a Pushdown Automaton[1] (already seen in L3 course?)

[1] Automate à Pile

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 :

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

- 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.

2 Syntactic Analysis

- Principles
- Tools

Tools: parser generators

- Parser generator: builds a Pushdown Automaton from a grammar definition;
- Ex: yacc (C), javacup (Java), OCamlyacc, **ANTLR**, ...
- **input** of ANTLR: a set of grammar rules with actions (`Toto.g4`);
- **output** of ANTLR: a file (`Toto.java`) that contains the corresponding Pushdown 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 applied");}
  | // nothing for empty alternative
  ;
```

Parser rules: quick reference

`name :   ...   ;`	: rule definition (lower-case for parsing rules)
same as lexer	: same meaning, in particular
`(...)`	: grouping
`|`	: alternative, e.g. `(a|b)`
new in parser	: rules can call each other recursively (`a: B a | ;`)

- Compile/execute with:

```
antlr4 explLexer.g4 explParser.g4
javac *.java
echo 'aabb' | java org.antlr.v4.gui.TestRig expl prog -gui
```

Parser rules: recommended format

```
// Do
rule: alternative A
    | alternative B
    | empty alternative
    ; // aligned with the |

// Don't (empty alternative hardly visible)
rule:
   | alternative A
   | alterantive B
   ;
```

Parser rules : dealing with ambiguities

A grammar with ambiguities

```
parse: expr EOF;

expr: expr '*' expr
    | expr '+' expr
    | INT
    ;

INT: [0-9]+;
WS: [ \t\n\r]+ -> skip;
```

Ambiguities:

1+2*3 (1+2)*3 or 1+(2*3) ? (precedence)

1+2+3 (1+2)+3 or 1+(2+3) ? (associativity)

ANTLR rules:

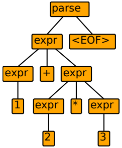
1. First alternative (`'*'` here) has highest precedence
2. Left-associative by default (customizeable, e.g. `'^'<assoc=right>`)

Parser rules : dealing with ambiguities

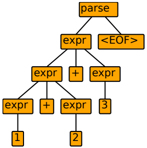
A grammar with ambiguities

```
parse: expr EOF;

expr: expr '*' expr
    | expr '+' expr
    | INT
    ;

INT: [0-9]+;
WS: [ \t\n\r]+ -> skip;
```

1+2*3 (1+2)*3 or 1+(2*3) ? (precedence)



1+2+3 (1+2)+3 or 1+(2+3) ? (associativity)



ANTLR rules:

1. First alternative (`'*'` here) has highest precedence
2. Left-associative by default (customizeable, e.g. `'^'<assoc=right>`)

ANTLR4 expressivity

ALL(*) = Adaptive LL(*)

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

Further reading (SIGPLAN Notices'14 paper, T. Parr, K. Fisher)

https://www.antlr.org/papers/allstar-techreport.pdf

Left recursion

ANTLR allows left recursion (but right recursion usually more efficient):

```
a: a b;
```

But not indirect left recursion. $X_1 \to \ldots \to X_n$

There exist algorithms to eliminate indirect recursions.

Lists

ANTLR allows lists:

```
prog: statement+ ; // one or more statements
block: statement* ; // zero or more statements
```

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.

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

▶ Next course!