Programming Language Design

Lexical Analysis

Francisco Ortin

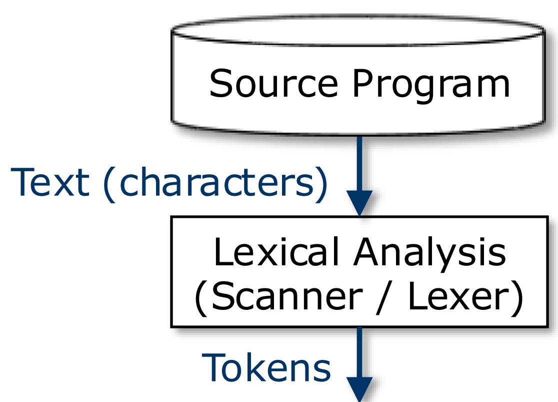
Department of Computer Science

Contents

- **Objectives of the Lexical Analyzer**
- Regular Expressions and Context-Free Grammars
- Implementation of Lexical Analyzers with ANTLR

Objective

- The **lexical analyzer** (scanner/lexer) is the **phase** of a language translator that reads the **source program**, as a sequence of characters, and divides it up into **tokens**
- A **token** is the minimum meaningful unit to be used by the parser
  - The process is similar to forming characters into words
  - Meaningless characters are discarded (e.g., new line, tabs, comments…)
- Tokens are commonly represented by **integers** (codes)



Example

Source:

41 Integers (characters)

```
while (a++ <= b) {
  // loop
  b += c;
}
```

⇩

Tokens:

13 Tokens

```
WHILE '(' ID INC LOW_EQ ID ')' '{' ID PLUS_ASG ID ';' '}'
```

'(' ')' '{' '}' represent character codes in Java

Question: How do we represent WHILE, ID… in Java?

Example

Source:

```
while (a++ <= b) {
 // loop
 b += c;
}
```

41 Integers (characters)

Blank spaces
Carriage return (\r, ASCII 13)
Line feed (\n, ASCII 10)
Tabs

New line in: *Windows = \r\n*
Unix (Mac 10+) = \n
Mac 9- = \r

Tokens:

13 Tokens

```
WHILE '(' ID INC LOW_EQ ID ')' '{' ID PLUS_ASG ID ';' '}'
```

'(' ')' '{' '}' represent character codes in Java

<u>Question</u>: How do we represent WHILE, ID... in Java?

Basic Concepts

- A **token** is the minimum meaningful unit to be used by the parser
  - `WHILE`, `IF`, `READ`, `INT`… (keywords)
  - `ID` (identifiers)
  - `'='`, `EQUAL`, `'+'`, `INC`… (operators)
  - `INT_CONSTANT`, `CHAR_CONSTANT`… (literals)
  - …
- A **lexeme** is the group of characters that form a token
  - `"while"`, `"if"`, `"read"`, `"int"`… (keywords)
  - `"a"`, `"factorial"`, `"letters"`, `"var1"`… (identifiers)
  - `"="`, `"=="`, `"+"`, `"++"`… (operators)
  - `"34"`, `"'a'"`… (literals)
  - …

Basic Concepts

- Therefore, the **objective** of a lexer is to
  - recognize <u>lexemes</u> in source files,
  - returning the appropriate <u>tokens</u>
- We specify the characters in lexemes a token may have by means of **patterns**
- How do we specify those patterns?

Contents

- Objective of the Lexical Analyzer
- **Regular Expressions and Context-Free Grammars**
- Implementation of Lexical Analyzers with ANTLR

Patterns

- By means of **formal languages** (grammars)
- Chomsky hierarchy:

	Language	**Grammar**	**Automaton**	
Type 0	Recursively enumerable	Unrestricted	Turing machine	
Type 1	Context-sensitive	Context-sensitive	Linear bounded automaton	
Type 2	Context-free	Context-free	Pushdown automaton	*Parsers & some lexers (ANTLR)*
Type 3	Regular	Regular	Finite state machine	*Most lexers (flex)*

Regular Languages

- Patterns of tokens are sometimes specified with **regular languages**
  - The ANTLR tool uses context-free grammars (CFG)
- A **regular language** over an alphabet Σ is either
  - The empty language $\varnothing$ (no input/string is accepted)
  - {A} or A, for A $\in \Sigma$ (including ε, the empty string)
  - Let *a* and *b* be regular languages, then $a \cup b$ or $a|b$ (union), $a \bullet b$ or *ab* (concatenation) and *a** (Kleene star) are regular languages
- **Regular expressions** formalize regular languages
- <u>Examples</u> of regular expressions:
  - $\varnothing$ (empty language; no program)
  - ε (the empty string; just one program)
  - (0|1)* (possibly empty sequence of 0 and 1)

Regular Languages

- **Set-builder notation** is a way to describe sets
  - It can also be used to specify regular languages
- Using **set-builder notation**, the following are examples of regular languages
  - $\{\varepsilon\}$
  - $\{A^n : n \geq 1\}$
  - $\{(A^n B^m)|(B^n A^m) : n,m \geq 0\}$
- Example of a **non-regular language**:
  - $\{ [^n (A|B)^m]^n : n,m \geq 0\}$

- Activity: write the regular expressions for the example regular languages above

Context-Free Grammars

- The ANTLR tool uses **Context-Free Grammars** (CFG) for both lexical and syntax analyzers
- CFGs are defined by the 4-tuple: $G=(V_N, V_T, P, S)$ where
  - V_T is a finite set of **terminals** (characters in lexical, tokens in syntax analysis); V_T is also called alphabet (sometimes called Σ)
  - V_N is a finite set of **non-terminal symbols**
  - S is the **start symbol**, $S \in V_N$
  - P is a finite set of **productions** (or rewrite rules)

  Every production $p \in P$ is formalized as

$$a \rightarrow \alpha$$

  where $a \in V_N$ and $\alpha \in (V_T \cup V_N)^*$

One-Step Derivations

- Example: Let G=({s,e}, {A,B}, *P*, s) where *P* is the set of rules:

  s $\rightarrow$ A e
  e $\rightarrow$ A
  e $\rightarrow$ B
  e $\rightarrow$ ε

- A **string** is a sequence of symbols (terminal and non-terminals), i.e., $\alpha \mid \alpha \in (V_N \cup V_T)^*$ (e.g., **s**, **A**, **A e**, **A e B**…)
- A **one-step derivation** (denoted as $\Rightarrow$) of a string is the application of one grammar production that transforms the string into another one
- Example: one-step derivations to recognize A:

Derivations			Production applied
s	$\Rightarrow$	A e	s $\rightarrow$ A e
	$\Rightarrow$	A	e $\rightarrow$ ε

- Questions: What is the language generated by G? Can you represent it with a regular expression?

CFG vs. Regular Expressions

- The main difference between Context-Free Grammars and Regular Expressions is that the former supports **recursion**
- Example: Let G=({e}, {A,B}, *P*, e) where *P* is the set of rules:

  e $\rightarrow$ A e B

  e $\rightarrow$ ε
- Question: What is the language generated by G
- Question: Is it possible to represent that language with a regular expression? Why?

Notation

- A common notation is the Backus Normal Form (**BNF**) that allows the use of the **|** meta-character
- So, the productions

  e $\rightarrow$ A e B

  e $\rightarrow$ ε
- Can also be expressed in BNF as:

  e $\rightarrow$ A e B

  | ε

  please, avoid

  e $\rightarrow$ A e B | ε
- **ANTLR**: We follow the ANTLR notation for
  - **Terminals**: (first letter) uppercased
  - **Non-terminals**: (first letter) lowercased

Activity: Context-Free Grammars

- In most programming languages there are **recurring syntax patterns**
- A common pattern is the specification of **lists of elements**
- Activity: Specify CFGs in BNF to define the following languages (use both left and right recursion)

  Use ***list*** as the start symbol (S)

1. $L(G_1) = \{\mathbf{A}^n : n \geq 1\}$ = **A**, **A A**, **A A A** …
2. $L(G_2) = \{\mathbf{A}^n : n \geq 0\}$ = ε, **A**, **A A** , **A A A** …
3. $L(G_3) = \{\mathbf{A}\ (\mathbf{,}\ \mathbf{A})^n : n \geq 0\}$ = **A**, **A,A**, **A,A,A** …
4. $L(G_4) = \{(\mathbf{A}\ (\mathbf{,}\ \mathbf{A})^n)? : n \geq 0\}$ = ε, **A**, **A,A**, **A,A,A** …
5. $L(G_5) = \{(\mathbf{A}\ (\mathbf{;}\ \mathbf{A})^n\mathbf{;})? : n \geq 0\}$ = ε, **A;**, **A;A;**, **A;A;A;**…

   where $V_T = \Sigma$ = alphabet = { **A , ;** }

Autonomous Optional Activity

- Do you remember the example of the following **non-regular language**?

  $\{ [^n(A|B)^m]^n : n,m \geq 0\}$

- Can you write a CFG to recognize it?

EBNF Notation

- ANTLR supports a powerful **Extended BNF** (EBNF) notation
- Let $r,s \in (V_T \cup V_N)$
  - *r*|*s*: Union; matches *r* or *s*
  - *r s*: Concatenation, matches *r* and then *s*
  - *r**: Kleen closure; zero or more repetitions of *r*
  - *r*+: Iteration; is equivalent to *rr**
  - *r*?: Option; matches the empty input or *r*
- <u>Question</u>: Using EBNF, write a CFG defining G_4

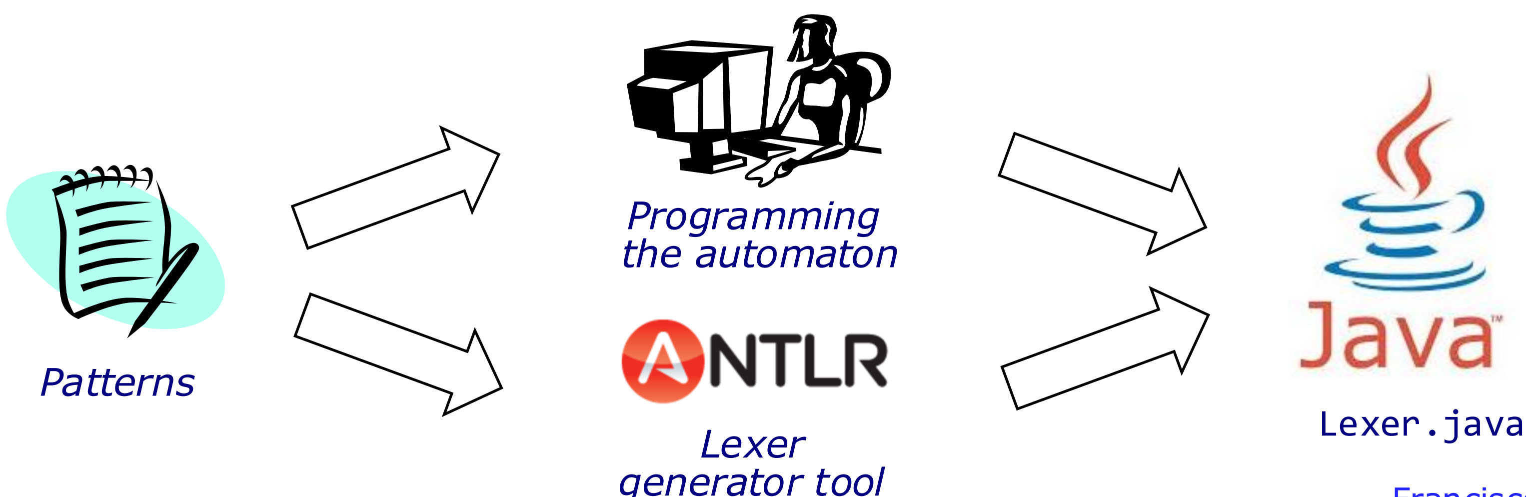
$L(G_4) = \{\varepsilon, \mathbf{A}, \mathbf{A,A}, \mathbf{A,A,A} \ldots\}$

Contents

- Objective of the Lexical Analyzer
- Regular Expressions and Context-Free Grammars
- **Implementation of Lexical Analyzers with ANTLR**

Implementation of Lexers

- Recall, **lexers** are commonly specified with **regular expressions** or **CFGs** representing the lexeme **patterns** of the recognized **tokens**
- Once we have the grammar, there are two ways to implement a lexer:
  - Implementing the automaton (lexer) **by hand**
  - Using a **tool** for generating lexical analyzers



ANTLR

- **ANTLR** ANother Tool for Language Recognition
- A **parser** and **lexer** generator for processing textual and binary files
  - It also provides **tree walkers** (i.e., AST grammars)
- Widely used to build languages, tools and frameworks
  - X (Twitter), Hadoop, Android, Lex Machina, Oracle, PayPal, NetBeans IDE, HQL Hibernate…
- <u>Lots of grammars for many languages</u> are available
- Implemented for Java, C#, Python, JavaScript, Go, C++ and Swift
- Developed by Terence Parr (University of San Francisco, Google)
- We will use ANTLR 4.x

ANTLR

- **ANTLR** <u>receives</u> the **lexical** and **syntactic specification** of a language and <u>generates</u> the **lexer** and **parser** implementations

`MyLang.g4`

Lexical and Syntactic specification (grammar)

ANTLR

`MyLangParser.java`

Java

`MyLangLexer.java`

Interface of the Lexer

- The interface of the `MyLangLexer` class is:
  - **`nextToken()`**`:Token` The main method; each time it is called, the following token is returned

Interface of the Lexer

- The interface of the **`MyLangLexer`** class is:
  - **`nextToken`**`():Token` The main method; each time it is called, the following token is returned
- The interface of the **`Token`** class is:
  - **`getType`**`():int` The token unique key
    - Keys are available as `public static final` fields in the `MyLangParser` class
    - The end of file is reached when `lexer.nextToken().getType()==MyLangParser.EOF`
  - **`getLine`**`():int` The token line
  - **`getCharPositionInLine`**`():int` The token column - 1
  - **`getText`**`():String` The token **lexeme**
- `MyLang`**`Lexer`**`(CharStream)` Constructor receiving any text stream (file, console, string…)

Interface of the Lexer

- Example use

```
CharStream input = CharStreams.fromFileName("input.txt");

MyLangLexer lexer = new MyLangLexer(input);
Token token;
while ((token = lexer.nextToken()).getType() != MyLangParser.EOF) {
    System.out.printf("Line: %d, column: %d, lexeme: '%s', " +
                      "token: %s.\n",
    token.getLine(),
    token.getCharPositionInLine()+1,
    token.getText(),
    lexer.getVocabulary().getDisplayName(token.getType())
    );
}
```

ANTLR Specification File

- The specification file has the following structure:

General Structure

Grammar Name

Options

Syntax rules

Lexical rules

Particular example (C--)

Cmm.g4

```
grammar Cmm;

@header {
  import ast.*;
  import types.*;
}

program: …
       ;
…

INT_CONSTANT: …
            ;
…
```

Non-terminals start with lowercase

Terminals start with uppercase

ANTLR Specification File

- Initially, we will just write lexical specifications (no syntax analysis yet)
- And we do not require any particular option, so the file will be

```
grammar Cmm;

program:
       ;

/* Lexical rules */

INT_CONSTANT: …
            ;
…
```

- How do we specify the lexical rules / productions?

ANTLR Specification File

- The **lexical rules** define the behavior of the lexer/scanner

  i.e., the implementation of `nextToken():Token`
- Each rule specifies the **pattern** of the different **lexemes** for a particular **token**
- Those patterns are expressed with **CFG**s in **EBNF** (Extended BNF) notation
- A very basic first example

```
grammar Cmm;

program:
       ;

INT_CONSTANT: ('0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9')+
            ;
```

EBNF Notation

- ANTLR patterns for **terminal** symbols (V_T)
  - Lexemes can be represented between `'` and `'`

    `'0'`, `'+'`, `'int'`
  - `\`: escape character
    - `'\''` (apostrophe), `'\\'` (backslash),
    - `\n`, `\r`, `\t`, `\b` and `\f`: special characters (`\b` = backspace, `\f` = form feed)
  - `.`: matches any character (wildcard)

- Question: Write a pattern to recognize Java / C char constants / literals

EBNF Notation

- ANTLR patterns for **terminal** symbols (V_T)
  - `'`*x*`'..'`*y*`'` (*x* and *y* being characters): matches any character between *x* and *y*, inclusively
  - `[`*x*`-`*y*`]`: identical to `'`*x*`'..'`*y*`'` (more common)
  - `[`*xyz*`]`: matches *x*, *y* or *z*; identical to `('x'|'`*y*`'|'`*z*`')` (more common)
  - `~`*t* (*t* being a set of characters): matches any single character not in *t*
- <u>Question</u>: Write a pattern to recognize a Java / C multiline comments (e.g., `/* … */`)
  - `.*?` *t* (*t* being a set of characters): non-greedy operator, equivalent to (`~`*t*)`*` *t*

EBNF Notation

- Lexemes can be represented between `'` and `'`
  `'0'`, `'+'`, `'int'`
- `\`: escape character
  - `'\''` (apostrophe), `'\\'` (backslash),
  - `\n`, `\r`, `\t`, `\b` and `\f`: special characters (`\b` = backspace, `\f` = form feed)
- `.`: matches any character (wildcard)
- `'`*x*`'..'`*y*`'` (*x* and *y* being characters): matches any character between *x* and *y*, inclusively
- `[`*x*`-`*y*`]`: identical to `'`*x*`'..'`*y*`'` (more common)
- `[`*xyz*`]`: matches *x*, *y* or *z*; identical to `('x'|'`*y*`'|'`*z*`')`
- `~`*t* (*t* being a set of characters): matches any single character not in *t*
- `.*?` *t* (*t* being a set of characters): non-greedy operator, equivalent to (`~`*t*)`*` *t*

- **Question**: Write a pattern to recognize any letter (English alphabet)

EBNF Notation

- Recall the following patterns **for any symbol** $(V_T \cup V_N)$
- Let $r,s \in (V_T \cup V_N)$
  - $r|s$: Union, matches r or s
  - $r\ s$: Concatenation, matches r and then s
  - $r*$: Kleen closure, zero or more repetitions of r
  - $r+$: Iteration, is equivalent to $rr*$
  - $r?$: Option, matches the empty input or r

Mandatory Activity

- Write an ANTLR grammar to recognize integer constants / literals
- Recall
  - Lexemes `'0'`, `'+'`, `'int'`, `'\''`
  - `.`: any character
  - `[x-y]`: `'`*x*`'..'`*y*`'`
  - `[`*xyz*`]`: `('x'|'`*y*`'|'`*z*`’)`
  - `~`*t* any single character not in *t*
  - `\`: escape character
  - *r*`|`*s*: Union, matches *r* or *s*
  - *r s*: Concatenation, matches *r* and then *s*
  - *r*`*`: Kleen closure, zero or more repetitions of *r*
  - *r*`+`: Iteration, is equivalent to *rr**
  - *r*`?`: Option, matches the empty input or *r*

Fragment

- It is possible to **reuse patterns**
- If a lexical pattern is too big, it is better to **break it into small patterns**
- In addition, those rules aimed at being <u>used by other rules</u> (i.e., they **<u>do not define a token</u>**) should be prefixed with the **`fragment`** keyword

```
grammar Fragment;

program: ;

fragment
DIGIT: [0-9]
     ;

INT_CONSTANT: '0'
            | [1-9] DIGIT*
            ;
```

Skip

- As mentioned, one of the objectives of the lexer is to discard meaningless characters (e.g., new line, tabs, comments…)
- ANTLR provides this functionality with **lexical rules that specify the lexemes to be discarded**, adding `-> skip` at the end of the production

```
grammar Skip;

program: ;

WHITE_SPACES: ' '+ -> skip
            ;
```

nextToken():Token

- So, what happens if?
  - No pattern is matched?
  - Two patterns are matched?
- What is the **algorithm** of the generated `nextToken()`?

```
Token nextToken() {
  while(current character is not end-of-file) {
    if (any pattern matches)
      return the token matching the first pattern
           that recognizes the longest lexeme
    else {
      System.err.println("line x:y token  " +
            "recognition error at 'character'");
      ignore character
  }   }
  return new Token(MyLangParser.EOF);
}
```

Mandatory Activity

- The following scanner recognizes integer literals

```
grammar IntLiteralsLang;

program:
       ;

INT_CONSTANT: '0'
            | [1-9][0-9]*
            ;
```

- What happens if a space, tabulation, line feed or carriage return appears?
- How can we solve it?
- Which tokens are recognized for the following input? `129 0102`

Mandatory Activity

- What does the following scanner return for the following source programs?

Source programs:

`int`	`while`	`variable`	`integer`	`hi`	`int3`

KewordsAndIDsLang.g4

```
grammar KewordsAndIDsLang;

program: ;

INT: 'int' ;
WHILE: 'while' ;
ID: [a-z]+ ;
WS: [ \t\n\r]+ -> skip ;
```

Autonomous Activity

- Write an **ANTLR lexical specification** file for the following patterns:
  - Identifiers

    `var1, a, var_2, __private, _`
  - Real constants (without exponent)

    `0.0, 1., .45`
  - Ignore single line comments

    `// This is one single-line comment`

Bibliography

- Alfred V. Aho, Monica S. Lam. Compilers: Principles, Techniques, and Tools, 2 Edition. Addison Wesley, 2006.
- Terence Parr. The Definitive ANTLR 4 Reference, 2nd edition. Pragmatic Bookshelf, 2013.
- Kenneth C. Louden. Compiler Construction, Principles and Practice. PWS Publishing, 1997.