Formal Languages and Compilers
Lecture VI: Lexical Analysis

Alessandro Artale

Free University of Bozen-Bolzano
Faculty of Computer Science – POS Building, Room: 2.03
artale@inf.unibz.it
http://www.inf.unibz.it/∼artale/

Formal Languages and Compilers — BSc course

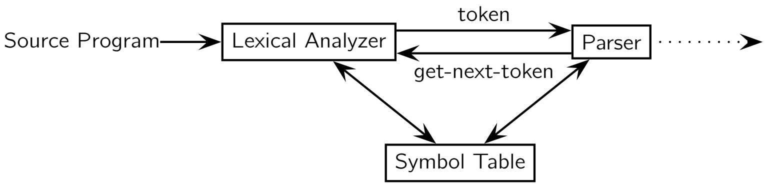
2019/20 – Second Semester

Summary of Lecture VI

- Lexemes and Tokens.
- Lexer Representation: Regular Expressions (RE).
- Implementing a Recognizer of RE's: Automata.
- Lexical Analyzer Implementation.
  - Breaking the Input in Substrings.
  - Dealing with conflicts.
  - Lexical Analyzer Architecture.

Lexical Analyzer

- Lexical Analysis is the first phase of a compiler.
- **Main Task:** Read the input characters and produce a sequence of **Tokens** that will be processed by the Parser.
- Upon receiving a "*get-next-token*" command from the Parser, the input is read to identify the next token.
- While looking for next token it eliminates comments and white-spaces.
- **Tokens** are treated as **Terminal Symbols** in the Grammar for the source program.



Token Vs. Lexeme

- In general, a set of input strings (Lexemes) give rise to the same Token. For example:

TOKEN	LEXEME	DESCRIPTION
var	var	language keyword
if	if	language keyword
then	then	language keyword
else	else	language keyword
while	while	language keyword
...	...	...
relation	$<, <=, =, <>, >, >=$	comparison operators
id	position, A1, x, y, ...	sequence of letters and digits
num	14, 3.14, 6.02E23, ...	numeric constant

Attributes for Tokens

- When a Token can be generated by different Lexemes the Lexical Analyzer must transmit also the Lexeme to the subsequent phases of the compiler.
- Such information is specified as an **Attribute** associated to the Token.
- Usually, the attribute of a Token is a pointer to the symbol table entry that keeps information about the Token.

Important!

- The Token influences parsing decisions: Parser relies on the token distinctions, e.g., identifiers are treated differently than keywords;
- The Attribute influences the semantic and code generation phase.

Attributes for Tokens: An Example

Example. Let us consider the following assignment statement:

$$\texttt{E} := \texttt{M} * \texttt{C} * *2$$

then the following pairs $\langle$**token**,attribute$\rangle$ are passed to the Parser:

$\langle$**id**, pointer to symbol-table entry for E$\rangle$
$\langle$**assign-op**, $\rangle$
$\langle$**id**, pointer to symbol-table entry for M$\rangle$
$\langle$**mult-op**, $\rangle$
$\langle$**id**, pointer to symbol-table entry for C$\rangle$
$\langle$**exp-op**, $\rangle$
$\langle$**num**, integer value 2$\rangle$.

- Some Tokens have a null attribute: the Token is sufficient to identify the Lexeme.
- From an implementation point of view, each token is encoded as an integer number.

Identifiers Vs. Keywords

- Programming languages use fixed strings to identify particular **Keywords**–e.g., **if**, **then**, **else**, etc.
- Since keywords are just identifiers the Lexical Analyzer must distinguish between these two possibilities.
- If keywords are *reserved*—not used as identifiers—we can initialize the symbol-table with all the keywords and mark them as such.
- Then, a string is recognized as an identifier only if it is not already in the symbol-table as a keyword.

Summary

- Lexemes and Tokens.
- Lexer Representation: Regular Expressions (RE).
- Implementing a Recognizer of RE's: Automata.
- Lexical Analyzer Implementation.
  - Breaking the Input in Substrings.
  - Dealing with conflicts.
  - Lexical Analyzer Architecture.

Main Objective

What we want to accomplish:

- Given a way to describe Lexemes of the input language, automatically generate the Lexical Analyzer.

This objective can be split in the following sub-problems:

1. **Lexer specification language:** How to represent Lexemes of the input language.
2. **The lexical analyzer mechanism:** How to generate Tokens starting from Lexeme representations.
3. **Lexical analyzer implementation:** Coding (1) + (2) + breaking inputs into substrings corresponding to the pair Lexeme/Token.

Problem 1. Lexer Specification Language: Regular Expressions

- Regular Expressions are the most popular specification formalisms to describe Lexemes and map them to Tokens.
- **Example.** An identifier is made by a letter followed by zero or more letters or digits:

  $$\textbf{letter (letter} \mid \textbf{digit)}^*$$

  The vertical bar $\mid$ means "or";
  Parentheses are used to group sub-expressions;
  The $*$ means zero or more occurrences.

Regular Expressions

Each Regular Expression, say R, denotes a Language, $L(R)$. The following are the rules to build them over an alphabet $\mathbf{V}$:

1. If $a \in \mathbf{V} \cup \{\epsilon\}$ then a is a Regular Expression denoting the language $\{a\}$;
2. If R, S are Regular Expressions denoting the Languages $L(R)$ and $L(S)$ then:
    1. $R \mid S$ is a Regular Expression denoting $L(R) \cup L(S)$;
    2. RS is a Regular Expression denoting the concatenation $L(R)L(S)$, i.e. $L(R)L(S) = \{rs \mid r \in L(R) \text{ and } s \in L(S)\}$;
    3. R^* is a Regular Expression denoting $L(R)^*$, zero or more concatenations of $L(R)$, i.e. $L(R)^* = \bigcup_{i=0}^{\infty} L(R)^i$;
    4. (R) is a Regular Expression denoting $L(R)$.

Regular Expressions: Examples

Example. Let $\mathbf{V} = \{a, b\}$.

1. The Regular Expression $a \mid b$ denotes the Language $\{a, b\}$.
2. The Regular Expression $(a \mid b)(a \mid b)$ denotes the Language $\{aa, ab, ba, bb\}$.
3. The Regular Expression a^* denotes the Language of all strings of zero or more a's, $\{\epsilon, a, aa, aaa, \ldots\}$.
4. The Regular Expression $(a \mid b)^*$ denotes the Language of all strings of a's and b's.

Regular Expressions: Shorthands

Notational shorthands are introduced for frequently used constructors.

1. $+$, *One or more instances.* If R is a Regular Expression then $R^+ \equiv RR^*$.
2. ?, *Zero or one instance.* If R is a Regular Expression then $R? \equiv \epsilon \mid R$.
3. *Character Classes.* If $a, b, \ldots, z \in \mathbf{V}$ then $[a, b, c] \equiv a \mid b \mid c$, and $[a - z] \equiv a \mid b \mid \ldots \mid z$.

Regular Definitions

- Regular Definitions are used to give names to regular Expressions and then to re-use these names to build new Regular Expressions.
- A *Regular Definition* is a sequence of definitions of the form:
  $\mathbf{D_1} \rightarrow R_1$
  $\mathbf{D_2} \rightarrow R_2$
  $\ldots$
  $\mathbf{D_n} \rightarrow R_n$
  Where each $\mathbf{D_i}$ is a distinct name and each R_i is a Regular Expression over the extended alphabet $\mathbf{V} \cup \{\mathbf{D_1}, \mathbf{D_2}, \ldots, \mathbf{D_{i-1}}\}$.
- **Note:** Such names for Regular Expression will be often the Tokens returned by the Lexical Analyzer. As a convention, names are printed in **boldface**.

Regular Definitions: Examples

Example 1. Identifiers are usually strings of letters and digits beginning with a letter:

$$\begin{array}{rcl} \textbf{letter} & \rightarrow & A \mid B \mid \ldots \mid Z \mid a \mid b \mid \ldots \mid z \\ \textbf{digit} & \rightarrow & 0 \mid 1 \mid \cdots \mid 9 \\ \textbf{id} & \rightarrow & \textbf{letter}(\textbf{letter} \mid \textbf{digit})^* \end{array}$$

Using *Character Classes* we can define identifiers as:

$$\textbf{id} \quad \rightarrow \quad [A - Za - z][A - Za - z0 - 9]^*$$

Example 2. Numbers are usually strings such as 5230, 3.14, 6.45E4, 1.84E-4.

$$
\begin{array}{rcl}
\textbf{digit} & \to & 0 \mid 1 \mid \cdots \mid 9 \\
\textbf{digits} & \to & \textbf{digit}^{+} \\
\textbf{optional-fraction} & \to & (.\textbf{digits})? \\
\textbf{optional-exponent} & \to & (\mathtt{E}(+ \mid -)?\textbf{digits})? \\
\textbf{num} & \to & \textbf{digits} \;\; \textbf{optional-fraction} \;\; \textbf{optional-exponent}
\end{array}
$$

Summary

- Lexemes and Tokens.
- Lexer Representation: Regular Expressions (RE).
- Implementing a Recognizer of RE's: Automata.
- Lexical Analyzer Implementation.
  - Breaking the Input in Substrings.
  - Dealing with conflicts.
  - Lexical Analyzer Architecture.

Problem 2. The Lexical Analyzer Mechanism. Finite Automata

- We need a mechanism to recognize Regular Expressions and so associating Tokens to Lexemes.
- While Regular Expressions are a *specification language*, Finite Automata are their *implementation*.
  - Given an input string, w, and a Regular Language, L, they answer "yes" if $w \in L$ and "no" otherwise.

Deterministic Finite Automata

A Deterministic Finite Automata, DFA for short, is a tuple: $A = (S, \mathbf{V}, \delta, s_0, F)$:

- S is a finite non empty set of *states*;
- $\mathbf{V}$ is the *input symbol alphabet*;
- $\delta : S \times \mathbf{V} \rightarrow S$ is a *total* function called the *Transition Function*;
- $s_0 \in S$ is the *initial state*;
- $F \subseteq S$ is the set of *final states*.

Deterministic Finite Automata (Cont.)

To define when an Automaton *accepts* a string we extend the transition function, δ, to a multiple transition function $\hat{\delta} : S \times \mathbf{V}^* \rightarrow S$:

$$\begin{array}{rcl} \hat{\delta}(s, \epsilon) & = & s \\ \hat{\delta}(s, xa) & = & \delta(\hat{\delta}(s, x), a); \quad \forall x \in \mathbf{V}^*, \forall a \in \mathbf{V} \end{array}$$

A DFA accepts an input string, w, if starting from the initial state with w as input the Automaton stops in a final state:

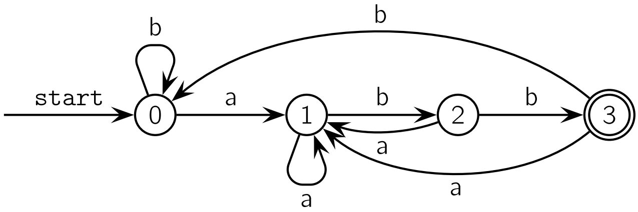
$$\hat{\delta}(s_0, w) = f, \text{ and } f \in F.$$

Language accepted by a DFA, $A = (S, \mathbf{V}, \delta, s_0, F)$:

$$\mathbf{L}(A) = \{w \in \mathbf{V}^* \mid \hat{\delta}(s_0, w) \in F\}$$

Transition Graphs

- A DFA can be represented by Transition Graphs where the nodes are the states and each labeled edge represents the transition function.
- The initial state has an input arch marked `start`. Final states are indicated by double circles.
- **Example.** DFA that accepts strings in the Language $L((a \mid b)^{*}abb)$.



Transition Table

- Transition Tables implement transition graphs, and thus Automata.
- A Transition Table has a row for each state and a column for each input symbol.
- The value of the cell (s_i, a_j) is the state that can be reached from state s_i with input a_j.
- **Example.** The table implementing the previous transition graph will have 4 rows and 2 columns, let us call the table δ, then:

$$\begin{array}{ll} \delta(0,a)=1 & \delta(0,b)=0 \\ \delta(1,a)=1 & \delta(1,b)=2 \\ \delta(2,a)=1 & \delta(2,b)=3 \\ \delta(3,a)=1 & \delta(3,b)=0 \end{array}$$

Summary

- Lexemes and Tokens.
- Lexer Representation: Regular Expressions.
- Implementing a Recognizer of RE's: Automata.
- Lexical Analyzer Implementation.
  - Breaking the Input in Substrings.
  - Dealing with conflicts.
  - Lexical Analyzer Architecture.

Main Objective

What we want to accomplish:

- Given a way to describe Lexemes of the input language, automatically generate the Lexical Analyzer.

This objective can be split in the following sub-problems:

1. **Lexer specification language:** How to represent Lexemes of the input language (Regular Expressions).
2. **The lexical analyzer mechanism:** How to generate Tokens starting from Lexeme representations (Side-Effect of Automata recognition process).
3. **Lexical analyzer implementation:** Coding (1) + (2) + breaking inputs into substrings corresponding to the pair Lexeme/Token.

Lexical Analyzer Implementation

- Given that the source program is just a single string, the Lexical Analyzer must do two things:
  1. Breaking the input string by recognizing substrings corresponding to Lexemes;
  2. Return the pair $\langle$**Token, Attribute**$\rangle$ for each Lexeme.
- Compare this procedure to Automata:
  1. Automata accept or reject a string, they do not “brake” it.
  2. We need to do more than just implement an Automaton.

Lexical Analyzer Implementation (Cont.)

The goal is to *partition* the source program into Lexemes which is implemented by:

1. Reading left-to-right, recognizing one lexeme at a time.
2. To “brake” an input string we need to solve **conflicts**. E.g., `pos` Vs. `position` as identifiers.

Summary

- Lexemes and Tokens.
- Lexer Representation: Regular Expressions.
- Implementing a Recognizer of RE's: Automata.
- Lexical Analyzer Implementation.
  - Breaking the Input in Substrings.
  - Dealing with conflicts.
  - Lexical Analyzer Architecture.

Dealing with Conflicts

There are two possible **conflicts**:

Case 1. The same Lexeme is recognized by two different RE's.

Case 2. The same RE can recognize portion of a Lexeme.

Dealing with Conflicts (Case 1)

Let's assume we have the following RE's:

$$\begin{array}{rcl} \mathbf{R_1} & \rightarrow & abb \\ \mathbf{id} & \rightarrow & \mathbf{letter}(\mathbf{letter} \mid \mathbf{digit})^* \end{array}$$

- The Lexeme "*abb*" matches both $\mathbf{R_1}$ and **id**.
- **Solution: Ordering between RE's.** If $''a_1 \ldots a_n'' \in L(R_j)$ and $''a_1 \ldots a_n'' \in L(R_k)$ we use the RE listed first (j, if $j < k$).
- **Remark.** To distinguish between keywords and identifiers insert the regular expression for keywords before the one for identifiers.

Dealing with Conflicts (Case 2)

Let's consider the RE for Identifiers and the string "`position`":

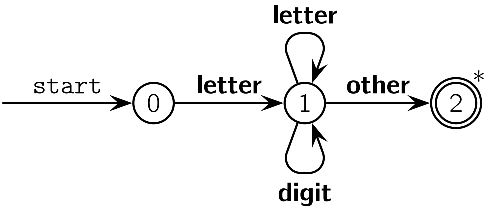
$$\textbf{id} \rightarrow \textbf{letter}(\textbf{letter} \mid \textbf{digit})^*$$

- "`p`" matches **id**;
- "`po`" matches **id**;
- . . .;
- "`position`" matches **id**.
- **Solution: Maximal Lexemes.** The lexeme for a given token must be maximal.

Maximal Lexemes Strategies (1)

Solution 1. Use Automata with *Lookahead*.

- *Example*: Automaton for **id** with lookahead.



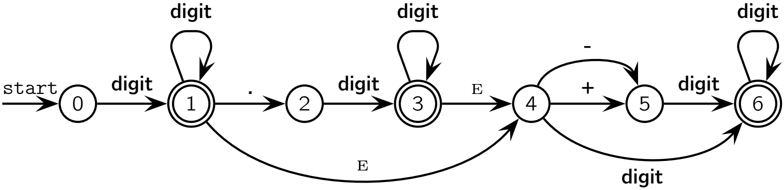
- The label **other** refers to any character not indicated by any other edge living the node;
- The $*$ indicates that we read an extra character and we must retract the *forward* input pointer by one character.

Maximal Lexemes Strategies (2)

Solution 2. Change the response to non-acceptance.

- Don't stop when reaching an accepting state;
- When failure occurs revert back to the last accepting state.
- This technique is the preferred one—used also by `Lex`.

Example: Automaton for numbers:



Try with the following input: "`61.23Express`".

Exercise. Modify the above Automaton to comply with the Lookahead technique.

Conflict Resolution in Lex

When several possible lexemes of the input match one or more RE's:

1. Always prefer a longer lexeme.
2. If the longest lexeme is matched by two or more RE's prefer the RE listed first.

Summary

- Lexemes and Tokens.
- Lexer Representation: Regular Expressions.
- Implementing a Recognizer of RE's: Automata.
- Lexical Analyzer Implementation.
  - Breaking the Input in Substrings.
  - Dealing with conflicts.
  - Lexical Analyzer Architecture.

From Regular Expressions to Lexical Analyzers

1. For each Token write the associated Regular Expression;
2. Build the Automaton for each Regular Expression;
3. Combine all the Automata (A_i) into a single big Automaton adding a new *start state* with ϵ-transitions (ϵ-NFA) to each of the start states of the A_i Automata.
4. Read the input to map Lexemes into Tokens till we successfully read the whole input or the Automaton fails without recognizing the input left (case of Lexical error).

Lexical Analyzer Example

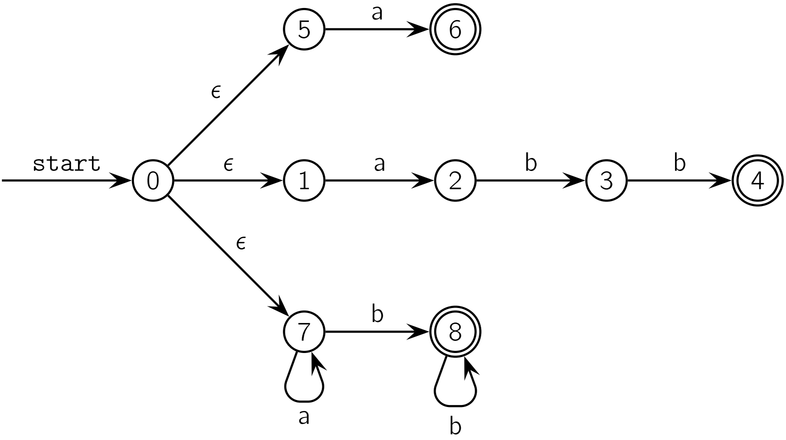
Consider the following Regular Expressions:

$$\begin{array}{lcl} \mathbf{R_1} & \rightarrow & a \\ \mathbf{R_2} & \rightarrow & abb \\ \mathbf{R_3} & \rightarrow & a^* b^+ \end{array}$$

Construct the Automaton and consider as input "`abb`".

Lexical Analyzer Example (Cont.)

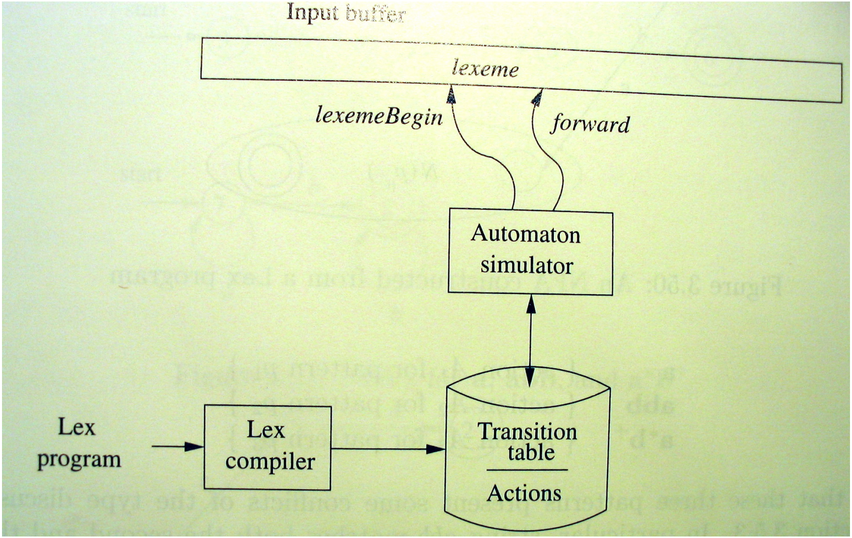
Single NFA Automaton.



Reading the Input

- The lexical analysis is the only phase that reads the input.
- Such reading can be time consuming: It is necessary to adopt efficient buffering techniques (see the book, Chapter 3.2, for more details).
- Two pointers, *begin* and *forward*, to the input buffer are maintained.
- The *begin* pointers always points at the beginning of the lexeme to be recognized.

Lexical Analyzer Architecture



Breaking the Input into Lexemes

- Two pointers, *begin* and *forward*, to the input buffer are maintained:
  1. Initially, both pointers point to the first character of the Lexeme to be found;
  2. The *forward* pointer scans ahead the input until there are no more next states in the Automaton—we are sure that the longest lexeme has been found.
  3. We go back in the sequence of set of states (assuming the Automaton is NFA) till we find a set that contains one or more accepting states, otherwise **fail**.
  4. If there are more accepting states prefer the state associated with the RE listed first.
  5. When the Lexeme is successfully processed transmit the token and its attribute to the parser, and set both pointers to the next character immediately past the Lexeme.
  6. If there are no more input characters then **succeed** else go to point 1.

Summary of Lecture VI

- Lexemes and Tokens.
- Lexer Representation: Regular Expressions.
- Implementing a Recognizer of RE's: Automata.
- Lexical Analyzer Implementation.
  - Breaking the Input in Substrings.
  - Dealing with conflicts.
  - Lexical Analyzer Architecture.