LECTURE 8

LEXICAL ANALYSIS

SUBJECTS

uOttawa
L'Université canadienne
Canada's university

The role of the lexical analyzer

Specification of tokens

Finite state automata

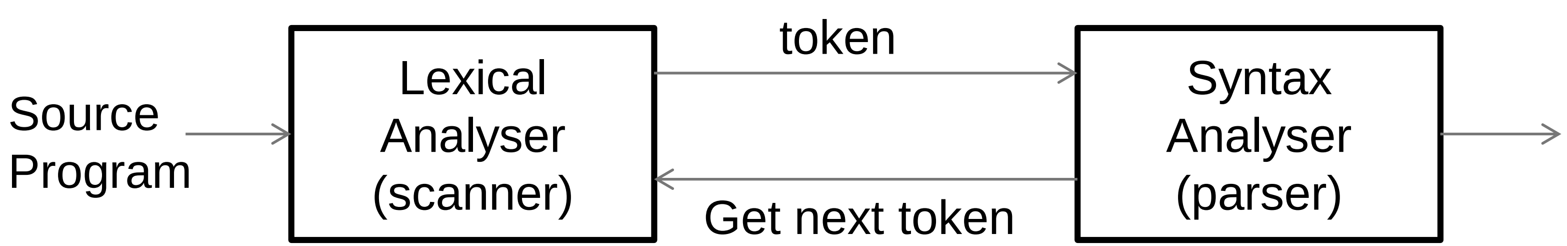
- DFA
- NFA

Recognizer

THE ROLE OF LEXICAL ANALYZER

Lexical analyzer is the first phase of a compiler

- Task: read input characters and produce a sequence of tokens that the parser uses for syntax analysis
- Remove white spaces



LEXICAL ANALYSIS

uOttawa
L'Université canadienne
Canada's university

There are several reasons for separating the analysis phase of compiling into lexical analysis and syntax analysis (parsing):

- Simpler (layered) design
- Compiler efficiency

Specialized tools have been designed to help automate the construction of both separately

LEXEMES

uOttawa
L'Université canadienne
Canada's university

Lexeme: sequence of characters in the source program that is matched by the pattern for a token

- A lexeme is a basic lexical unit of a language
- Lexemes of a programming language include its
  - **Identifiers**: names of variables, methods, classes, packages and interfaces…
  - **Literals**: fixed values (e.g. "1", "17.56", "0xFFE" …)
  - **Operators**: for Maths, Boolean and logical operations (e.g. "+", "-", "&&", "|" …)
  - **Special words**: keywords (e.g. "if", "for", "public" …)

TOKENS, PATTERNS, LEXEMES

Token: category of lexemes

A pattern is a rule describing the set of lexemes that can represent a particular token in source program

LEXEME AND TOKEN

```
Index = 2 * count +17;
```

Lexemes	Tokens
Index	identifier
=	equal_sign
2	int_literal
*	multi_op
Count	identifier
+	plus_op
17	int_literal
;	semicolon

LEXICAL ERRORS

Few errors are discernible at the lexical level alone

- Lexical analyzer has a very localized view of a source program

Let some other phase of compiler handle any error

SPECIFICATION OF TOKENS

We need a powerful notation to specify the patterns for the tokens

- Regular expressions to the rescue!!

In the process of studying regular expressions, we will discuss:

- Operations on languages
- Regular definitions
- Notational shorthands

RECALL: LANGUAGES

Σ: alphabet, it is a finite set consisting of all input characters or symbols

Σ^*: closure of the alphabet, the set of all possible strings in Σ, including the empty string ε

A (formal) language is some specified subset of Σ^*

OPERATIONS ON LANGUAGES

Operation	Definition
union of L and M written $L \cup M$	$L \cup M = \{s \mid s \in L \text{ or } s \in M\}$
concatenation of L and M written LM	$LM = \{st \mid s \in L \text{ and } t \in M\}$
Kleene closure of L written L^*	$L^* = \bigcup_{i=0}^{\infty} L^i$
positive closure of L written L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$

OPERATIONS ON LANGUAGES

Non-mathematical format:

- **Union between languages L and M:** the set of strings that belong to at least one of both languages
- **Concatenation of languages L and M:** the set of all strings of the form **st** where **s** is a string from **L** and **t** is a string from **M**
- **Intersection between languages L and M:** the set of all strings which are contained in both languages
- **Kleene closure (named after Stephen Kleene)**: the set of all strings that are concatenations of **0 or more** strings from the original language
- **Positive closure** : the set of all strings that are concatenations of **1 or more** strings from the original language

REGULAR EXPRESSIONS

Regular expression is a compact notation for describing a string.

Typically an identifier is a letter followed by zero or more letters or digits → letter (letter | digit)*

| : or

*: zero or more instance of

RULES

ε is a regular expression that denotes {ε}, the set containing empty string

If *a* is a symbol in Σ, then *a* is a regular expression that denotes {*a*}, the set containing the string *a*

Suppose *r* and *s* are regular expressions denoting the languages L and M, then

- (*r*) |(*s*) is a regular expression denoting $L \cup M$.
- (*r*)(*s*) is regular expression denoting LM
- (*r*) * is a regular expression denoting (L)*.

PRECEDENCE CONVENTIONS

The unary operator * has the highest precedence and is left associative.

Concatenation has the second highest precedence and is left associative.

| has the lowest precedence and is left associative.

$(a)|(b)^*(c) \rightarrow a|b^*c$

PROPERTIES OF REGULAR EXPRESSION

Axiom	Description
$r \mid s = s \mid r$	$\mid$ is commutative
$r \mid (s \mid t) = (r \mid s) \mid t$	$\mid$ is associative
$(rs)t = r(st)$	concatenation is associative
$r(s \mid t) = rs \mid rt$ $(s \mid t)r = sr \mid tr$	concatenation distributes over $\mid$
$\varepsilon r = r$ $r\varepsilon = r$	ε is the identity for concatenation
$r^* = (r \mid \varepsilon)^*$	relation between * and ε
$r^{**} = r^*$	* is idempotent

EXAMPLES OF REGULAR EXPRESSIONS

Let $\Sigma = \{a, b\}$

1. $a|b$ denotes $\{a, b\}$

2. $(a|b)(a|b)$ denotes $\{aa, ab, ba, bb\}$
   i.e., $(a|b)(a|b) = aa|ab|ba|bb$

3. a^* denotes $\{\varepsilon, a, aa, aaa, \ldots\}$

4. $(a|b)^*$ denotes the set of all strings of a's and b's (including ε)
   i.e., $(a|b)^* = (a^*b^*)^*$

5. $a|a^*b$ denotes $\{a, b, ab, aab, aaab, aaaab, \ldots\}$

REGULAR DEFINITIONS

uOttawa
L'Université canadienne
Canada's university

If Σ is an alphabet of basic symbols, then a *regular definition* is a sequence of definitions of the form:

$d_1 \rightarrow r_1$
$d_2 \rightarrow r_2$
…
$d_n \rightarrow r_n$

Where each d_i is a distinct name, and each r_i is a regular expression over the symbols in $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$,

- i.e., the basic symbols and the previously defined names.

EXAMPLE OF REGULAR DEFINITIONS

$$\textit{letter} \rightarrow (a \mid b \mid c \mid \ldots \mid z \mid A \mid B \mid C \mid \ldots \mid Z)$$

$$\textit{digit} \rightarrow (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)$$

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

$$\textit{integer} \rightarrow (+ \mid - \mid \varepsilon) \; (0 \mid (1 \mid 2 \mid 3 \mid \ldots \mid 9) \; \textit{digit}^*)$$

$$\textit{decimal} \rightarrow \textit{integer} \; . \; (\; \textit{digit} \;)^*$$

$$\textit{real} \rightarrow (\; \textit{integer} \mid \textit{decimal} \;) \; \mathtt{E} \; (+ \mid -) \; \textit{digit}^*$$

NOTATIONAL SHORTHANDS

Certain constructs occur so frequently in regular expressions that it is convenient to introduce notational short hands for them

We have already seen some of these short hands:

1. **One or more instances:** a+ denotes the set of all strings of one or more a’s
2. **Zero or more instances:** a* denotes all the strings of zero or more a’s
3. **Character classes:** the notation [abc] where a, b and c denotes the regular expresssion a | b | c
4. **Abbreviated character classes:** the notation [a-z] denotes the regular expression a | b | …. | z

NOTATIONAL SHORTHANDS

Using character classes, we can describe identifiers as being strings described by the following regular expression:

[A-Za-z][A-Za-z0-9]*

FINITE STATE AUTOMATA

uOttawa
L'Université canadienne
Canada's university

Now that we have learned about regular expressions

- How can we tell if a string (or lexeme) follows a regular expression pattern or not?

We will again use state machines!

- This time, they are not UML state machines or petri nets
- We will call them: Finite Automata

The program that executes such state machines is called a *Recognizer*

FINITE AUTOMATA

uOttawa
L'Université canadienne
Canada's university

A *recognizer* for a language is a program that takes as input a string *x* and answers

- "Yes" if *x* is a lexeme of the language
- "No" otherwise

We compile a regular expression into a recognizer by constructing a generalized transition diagram called a *finite automaton*

A finite automaton can be *deterministic* or *nondeterministic*

- Nondeterministic means that more than one transition out of a state may be possible on the same input symbol

NONDETERMINISTIC FINITE AUTOMATA (NFA)

uOttawa
L'Université canadienne
Canada's university

A set of states S

A set of input symbols that belong to alphabet Σ

A set of transitions that are triggered by the processing of a character

A <u>single</u> state s_0 that is distinguished as the start *(initial) state*

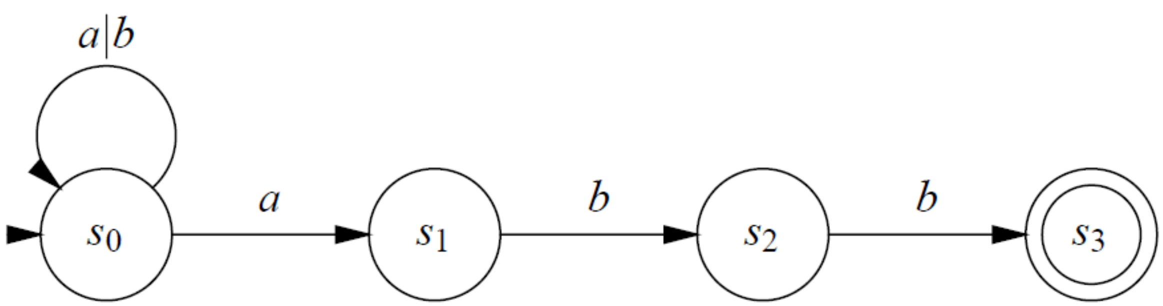
A <u>set</u> of states F distinguished as *accepting (final) states*.

EXAMPLE OF AN NFA

The following regular expression

(a|b)*abb

Can be described using an NFA with the following diagram:



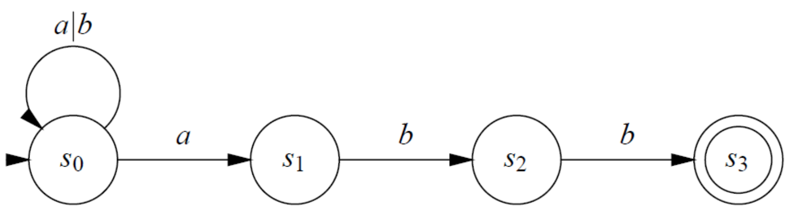
EXAMPLE OF AN NFA

The previous diagram can be described using the following table as well

Remember the regular expression was:

(a|b)*abb

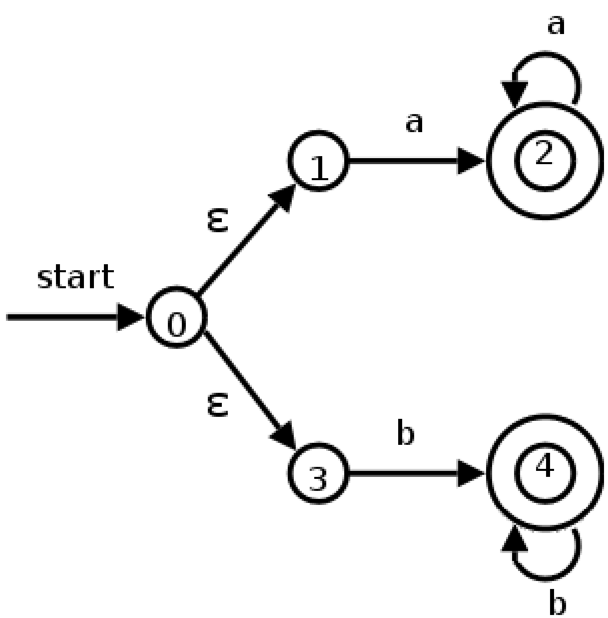
	a	b
s_0	$\{s_0, s_1\}$	$\{s_0\}$
s_1	–	$\{s_2\}$
s_2	–	$\{s_3\}$



ANOTHER NFA EXAMPLE

NFA accepting the following regular expression:

aa*|bb*



DETERMINISTIC FINITE AUTOMATA (DFA)

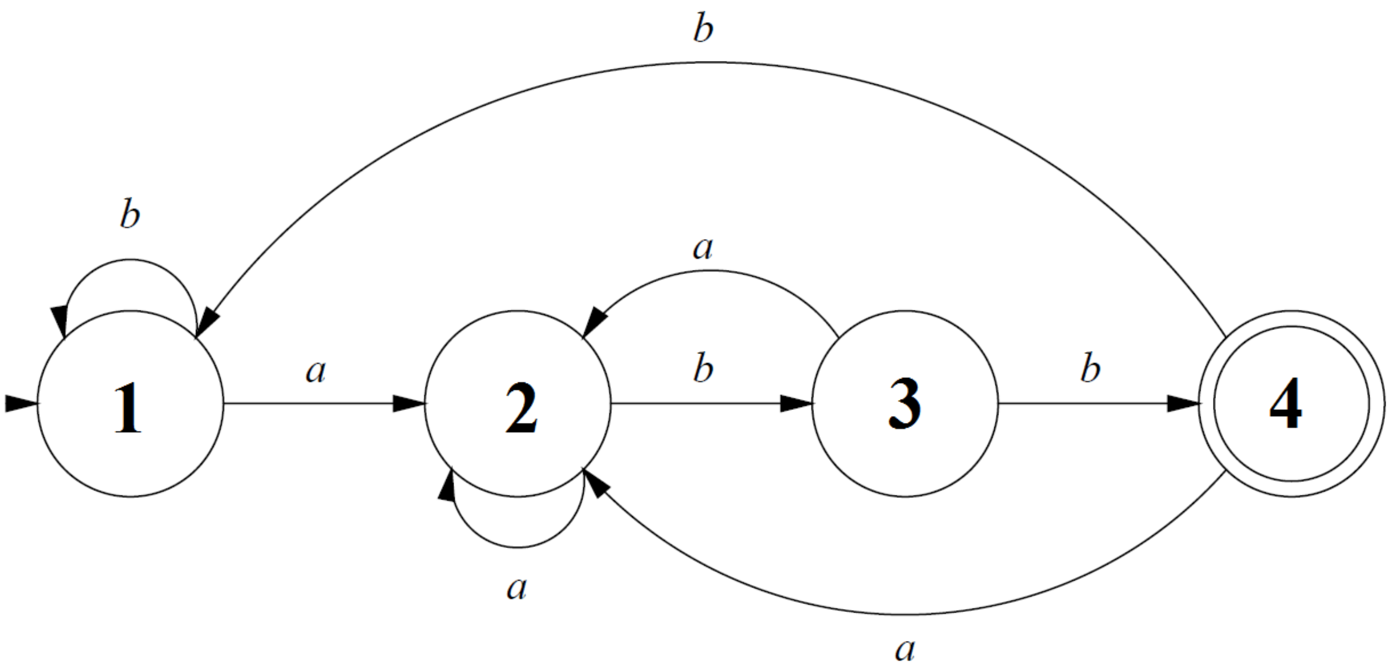
A DFA is a special case of a NFA in which

- No state has an ε-transition
- For each state *s* and input symbol *a*, there is at most one edge labeled *a* leaving *s*

ANOTHER DFA EXAMPLE

For the same regular expression we have seen before

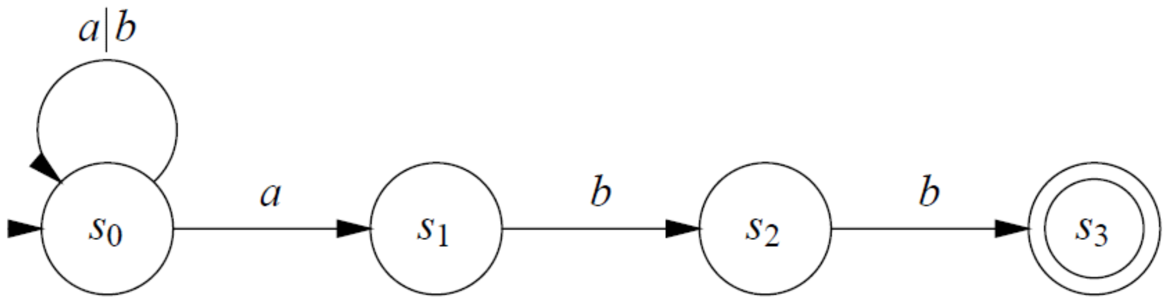
(a|b)*abb



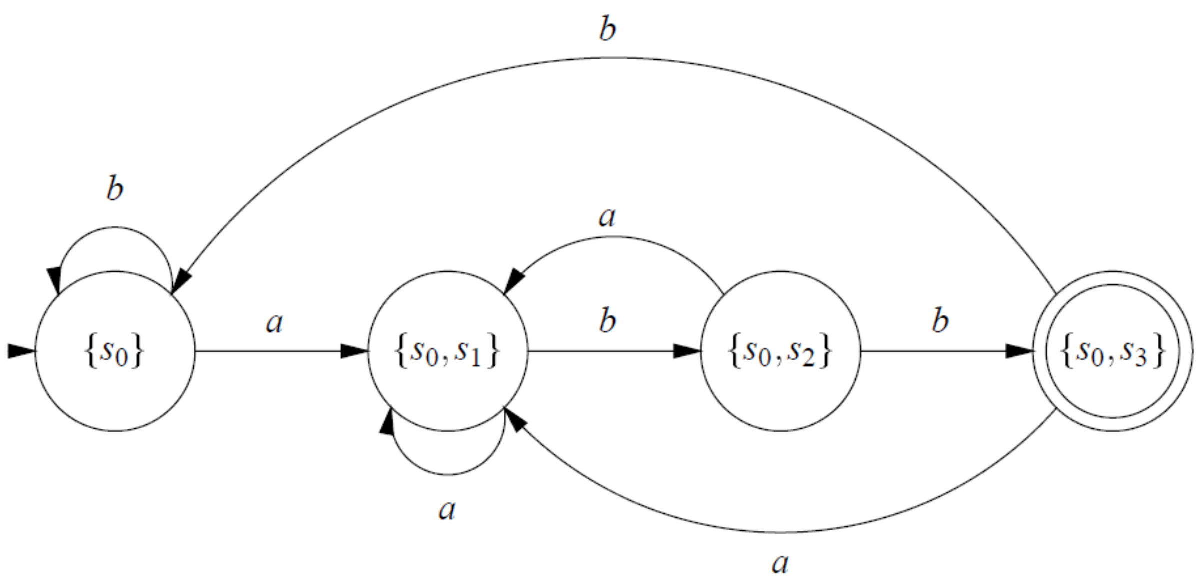
NFA VS DFA

Always with the regular expression: (a|b)*abb

NFA:

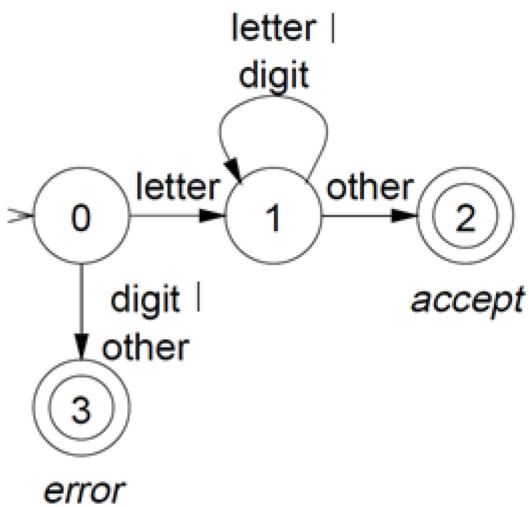


DFA:



EXAMPLE OF A DFA

Recognizer for identifier:



identifier

$$letter \rightarrow (a \mid b \mid c \mid \ldots \mid z \mid A \mid B \mid C \mid \ldots \mid Z)$$

$$digit \rightarrow (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)$$

$$id \rightarrow letter\ (\ letter \mid digit\ )^*$$

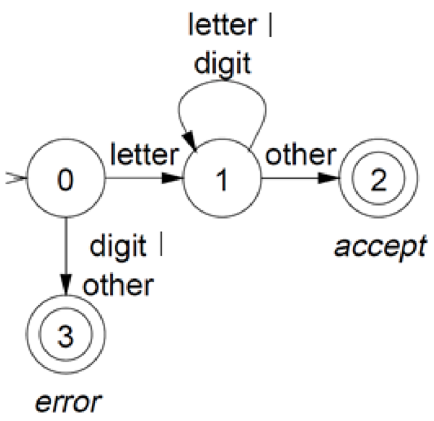
TABLES FOR THE RECOGNIZER

`char_class:`

	$a-z$	$A-Z$	$0-9$	other
value	letter	letter	digit	other

`next_state:`

class	0	1	2	3
letter	1	1	—	—
digit	3	1	—	—
other	3	2	—	—



To change regular expression, we can simply change tables…

CODE FOR THE RECOGNIZER

Recognizer algorithm:

```
char ← next_char();
state ← 0;          /* code for state 0 */
done ← false;
token_value ← ""    /* empty string */
while( not done ) {
   class ← char_class[char];
   state ← next_state[class,state];
   switch(state) {
      case 1:     /* building an id */
         token_value ← token_value + char;
         char ← next_char();
         break;
      case 2:     /* accept state */
         token_type = identifier;
         done = true;
         break;
      case 3:     /* error */
         token_type = error;
         done = true;
         break;
   }
}
return token_type;
```

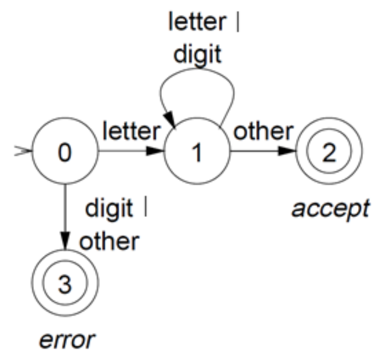
For reference:

char_class:

	$a-z$	$A-Z$	$0-9$	other
value	letter	letter	digit	other

next_state:

class	0	1	2	3
letter	1	1	—	—
digit	3	1	—	—
other	3	2	—	—



THANK YOU!

QUESTIONS?