



编译原理

Compiler Principles

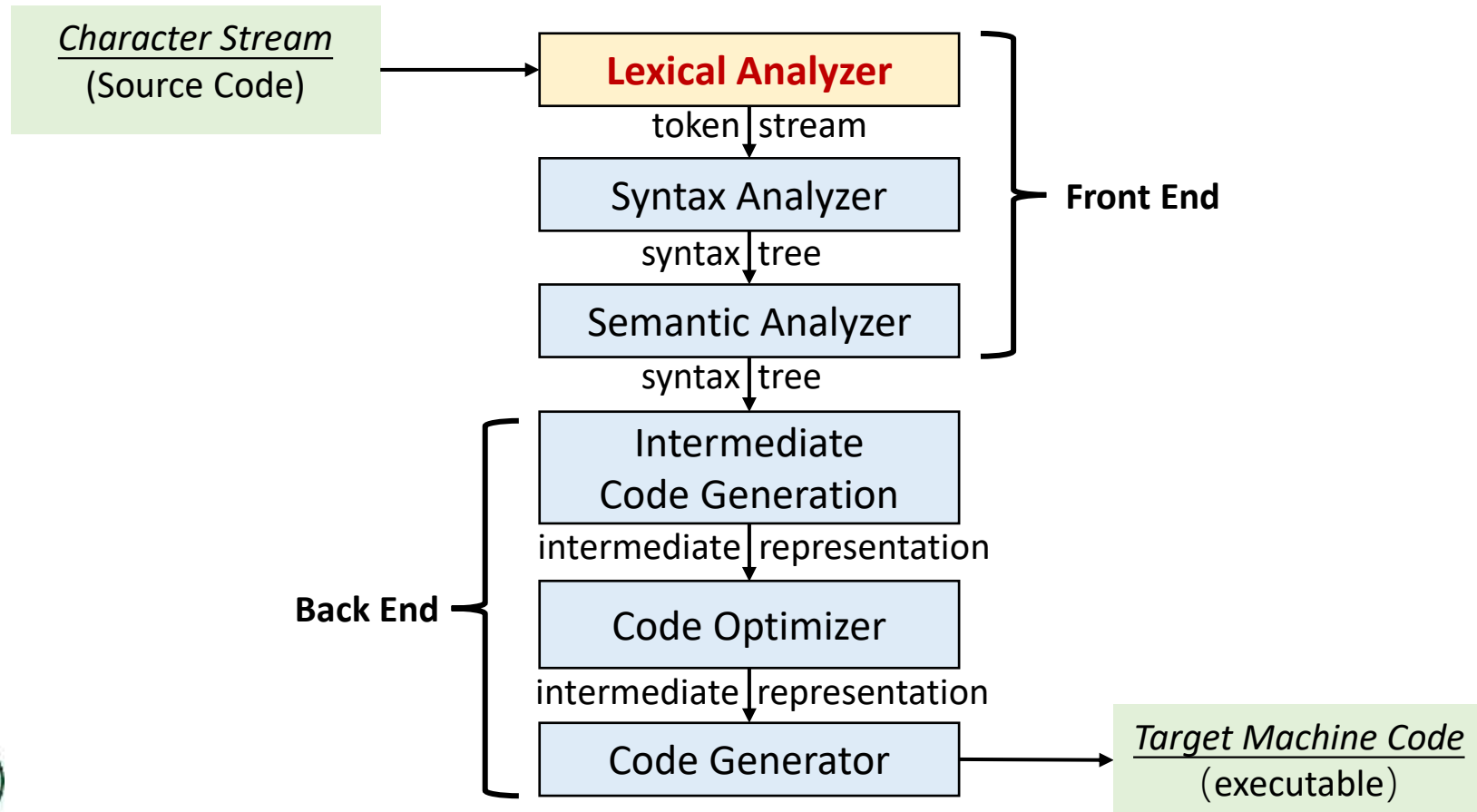
Lecture1

Lexical Analysis: Intro & Regular Expressions

赵帅

计算机学院
中山大学

The Starting Point



The Starting Point



Lexical Analysis:

```
while (y<z){  
    int x = a + b;  
    y += x;  
}
```

HOW?



(keyword, **while**)
(id, **y**)
(sym, **<**)
(id, **z**)
(id, **x**)
(id, **a**)
(sym, **+**)
(id, **b**)
(sym, **;**)
(id, **y**)
(sym, **+=**)
(id, **x**)
(sym, **;**)



What is Lexical Analysis?

- **Lexical Analysis** is the **process** of **identifying the substrings** (called lexeme[词素]) and **generating tokens**[词] by identifying token class.
- **Task:** Reading the source program as a string of characters and diving it up into tokens.

- **Step**

1. Remove comments: ~~/* simple example */~~
2. Identify substrings: 'while' '(' 'i' '==' 'j'
3. Identify token classes: (keyword, 'while'), (Lpar, '('), (id, 'i'), (Rpar, ')')

```
/* simple example */  
while (i<z)  
    i++;
```



What is Lexical Analysis[词法分析]?



- Example

```
/* simple example */  
while (i<z)  
    i++;
```

w	h	i	l	e		(i	<	z)	\	n	\	t	i	+	+	;
---	---	---	---	---	--	---	---	---	---	---	---	---	---	---	---	---	---	---

w	h	i	l	e		(i	<	z)	\	n	\	t	i	+	+	;
---	---	---	---	---	--	---	---	---	---	---	---	---	---	---	---	---	---	---

w	h	i	l	e		(i	<	z)	\	n	\	t	i	+	+	;
---	---	---	---	---	--	---	---	---	---	---	---	---	---	---	---	---	---	---

Keep scanning...

w	h	i	l	e		(i	<	z)	\	n	\	t	i	+	+	;
---	---	---	---	---	--	---	---	---	---	---	---	---	---	---	---	---	---	---

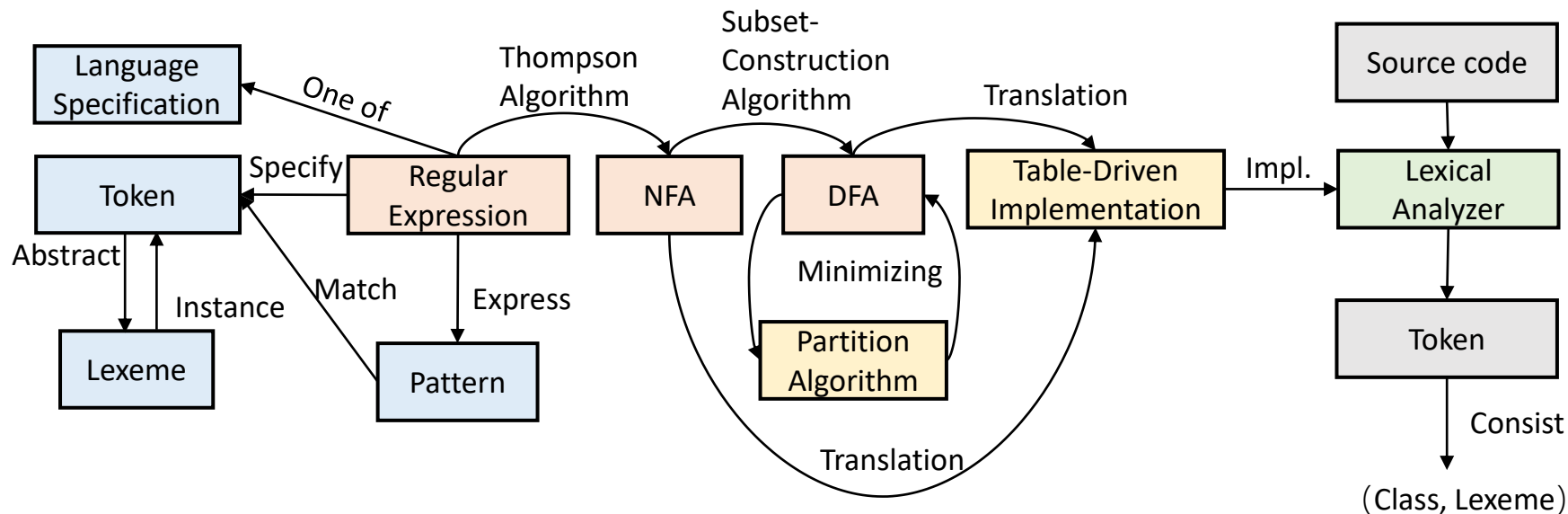
'while'

Token class

(keyword, 'while')

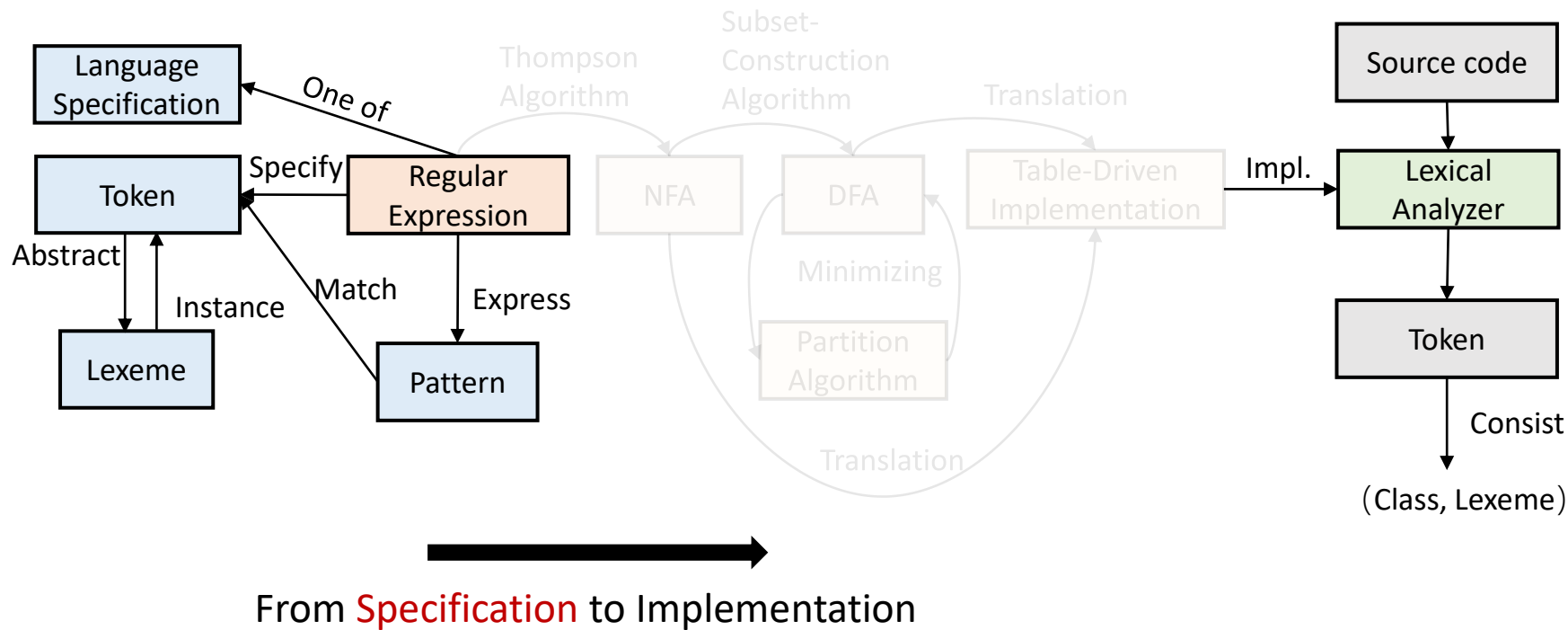


Lexical Analysis[词法分析]



From Specification to Implementation

Content



- **Token**[词法单元/词]: a “word” in language (smallest unit with meaning)
 - ◆ A token is a pair consisting of a token name and an optional attribute value.
 - ◆ A token is a tuple (class, lexeme)
 - ◆ The token name (i.e., class) is an abstract symbol representing a kind of lexical unit[词法单位], e.g., a particular keyword, or a sequence of input characters denoting an identifier.
- **Lexeme**[词素]: A lexeme is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token[词法单元的一个实例]



The Categories of Tokens

- **Numbers**: a non-empty string of consecutive digits
- **Keyword**: a fixed set of reserved words (“for”, “if”, “else”, ...)
- **Whitespace**: a non-empty sequence of blanks, tabs, newlines
- **Identifier**: user-defined name of an entity to identify

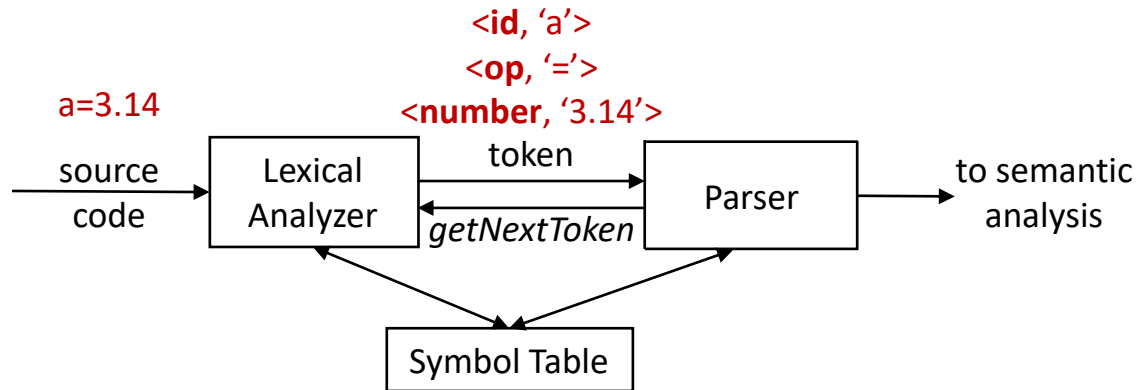


- Which of the following names are **NOT** accepted by Java?
 - A. `int 1var = 0;`
 - B. `int _var = 0;`
 - C. `int $var = 0;`
 - D. `int main = 0;`
 - E. `int while = 0;`
- In terms of **JAVA coding convention**, which of the above is GOOD coding practice?



The Role of Lexical Analysis

- Lexical analysis is also called **Tokenization** (or **Scanner**) [词法分析也称为扫描器]
 - ◆ Partition input string into a sequence of tokens.
 - ◆ Classify each token according to its role (token class), by matching patterns
 - ◆ Pass tokens to syntax analyzer (also called Parser) [语法分析器]
 - Parser relies on token classes to identify roles (e.g., a keyword is treated differently than an identifier)



Lexical Analysis: Design

- Define a **finite** set of token classes [定义词法单元类别]
 - ◆ Describe all items of interest
 - ◆ keyword, identifier, whitespace...
 - ◆ Depends on both the language and the design of parser
- Determine which string belongs to which token class [识别字符串属于哪个类别]

```
if (a == 3.14)
```

```
    stmt1;
```

```
else
```

```
    stmt2;
```

→ Should identify '=' or '=='

→ keyword or identifier?



- **An implementation must do two things**
 - ◆ Recognize the token class that the substring belongs to[识别分类]
 - ◆ Return the value or lexeme.
- The lexer usually strips out comments and whitespace (e.g., blank, newline, tab, etc.)[丢弃无意义词]
- If token classes are non-ambiguous, tokens can be recognized in a single left-to-right scan of the input string.
- Problem can occur when classes are ambiguous[二义性]



- C++: Nested template declarations

`vector<vector<int>> myVector`

- ◆ `(vector < (vector < (int >> myVector)))`
- ◆ `vector < vector < int >> myVector`

Template syntax ?

Stream syntax ?

Operator?

- Ambiguity

- ◆ `vector<vector<int>>`
- ◆ `cin >> var`

Q: Is '>>' a stream operator or two consecutive brackets?



- “look ahead” may be required to resolve **ambiguity**[消除歧义]
 - ◆ Extracting some tokens requires looking at the larger context or structure
 - ◆ Structure emerges only at the parsing stage with the parse tree
 - ◆ `vector<vector<int>>`
 - ◆ Hence, sometimes feedback from the parser[语法解析器] is needed for lexing
 - This complicates the design of lexical analysis
 - Should minimize the amount of looking ahead
- Usually, tokens do not overlap[通常无重叠]
 - ◆ Tokenizing can be done in one pass without parser feedback
 - ◆ Clean division between lexical and syntax analyses

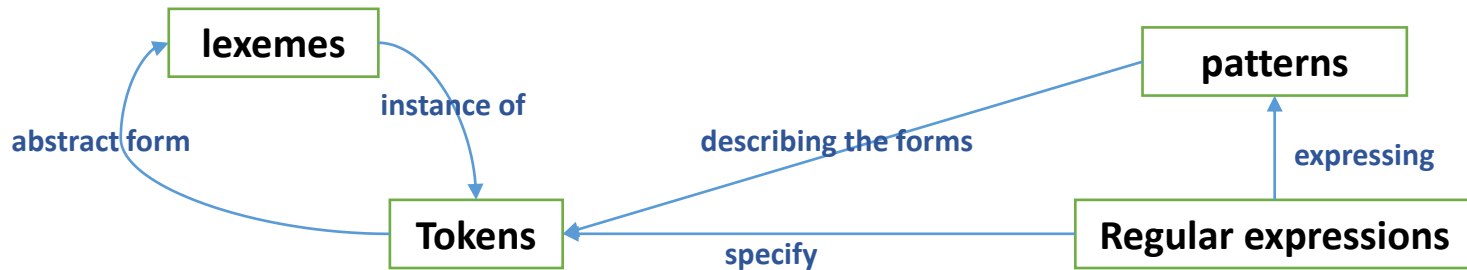


Token Specification[定义]



- Identifying token class: which set of strings belong to which token class?

- ◆ **Question:** How to describe string patterns? [模式]
- ◆ Use **regular expressions**[正则表达式] to define token class.



- Regular Expression is a good way to specify tokens.
 - ◆ Simple yet effective
 - ◆ Tokenizer[分词器] implementation can be generated automatically from specification (using a translation tool)



Language: Definition



- **Alphabet** Σ [字母表]: A finite set of symbols.
 - ◆ Symbol: letter, digit, punctuation, ...
 - ◆ Example: $\{0, 1\}$, $\{a, b, c\}$, ASCII, ...
- **String**[串]: A string over an alphabet is a finite sequence of symbols drawn from that alphabet. [字母表中符号的有穷序列]
 - ◆ Example: abc (length: $|abc|=3$), ϵ (empty string, length: $|\epsilon|=0$)
- **Language**[语言]: Any countable set of strings over some fixed alphabet. [某个给定字母表上一个任意的可数的串集合]
 - ◆ $\Sigma = \{a, b\}$, then $\{\}, \{ab, ba\}, \{a, aa, aaaa, \dots\}$ are all languages over Σ
 - ◆ An empty set[空集] ($\Phi = \{\}$) is a language
 - ◆ The set containing only an empty string ($\{\epsilon\}$) is also a language.
 - Φ and $\{\epsilon\}$ are not equal



Language: Example

- Examples:
 - ◆ Alphabet Σ = (set of) English characters
 - Language L = (set of) English sentences
 - ◆ Alphabet Σ = (set of) Digits, +, -
 - Language L = (set of) Integer numbers
- Languages are subsets of all possible strings
 - ◆ Not all strings of English characters are (valid) sentences
 - E.g., aaa bbb ccc
 - ◆ Not all sequences of digits and signs are valid integers
 - E.g., 125+, 1-25



- **Union**[并]: similar operation on sets, i.e., $A \cup B$, denoted as **A | B**
- **Concatenation**[连接]: all strings formed by taking a string from the first language and a string from the second language in all possible ways, denoted as **AB**
- **Closure**[闭包]: the (Kleene) closure of a language L is the set of strings by concatenating L zero or more times, denoted as **L^*** ,
 - ◆ $L^0 = \{\epsilon\}$, $L^i = L^{i-1}L$;
 - ◆ $L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \dots$
 - ◆ $L^+ = L^1 \cup L^2 \cup L^3 \cup \dots$ (the positive closure[正闭包], Kleene closure without L^0 . That is, ϵ will not be in L^+ unless it is in L itself.)
 - ◆ $L^+ = LL^*$



Example



- Language: $L = \{a, b\}$, $D = \{0, 1\}$
- $LUD = \{a, b\} \cup \{0, 1\} = \{a, b, 0, 1\}$
- $LD = \{a, b\}\{0, 1\} = \{a0, b0, a1, b1\}$
- $L^3 =$ Quiz
- $L^* = \bigcup_{i=0}^{\infty} L^i = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \dots$
 $= \{\epsilon\} \cup \{a, b\} \cup \{a, b\}^2 \cup \{a, b\}^3 \dots$
 $= \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, \dots\}$
- $L^+ = \bigcup_{i=1}^{\infty} L^i =$
 $= \{a, b\} \cup \{a, b\}^2 \cup \{a, b\}^3 \dots$
 $= \{a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, \dots\}$



Example Cont.

- $L = \{A, B, \dots, Z, a, b, \dots, z\}$, $D = \{0, 1, \dots, 9\}$
 - ◆ L and D are languages whose strings happen to be of length one
 - ◆ Some other languages that can be constructed from L and D are:
- $L \cup D$: the set of letters and digits, i.e., language with 62 strings of length one
- LD : the set of 520 strings of length two, each is one letter followed by one digit
- L^4 : the set of all 4-letter strings
- L^* : the set of all strings of letters, including ϵ , the empty string
- D^+ : the set of all strings of one or more digits
- $L(L \cup D)^*$: t

Quiz



- **Regular expressions** [正则表达式] are to describe all languages that can be built from the operators applied to the symbols of some alphabet.
- **Regular Expression is a simple notation**
 - ◆ Can express simple patterns (e.g., repeating sequences)
 - ◆ Not powerful enough to express English (or even C)
 - ◆ But powerful enough to express tokens (e.g., identifiers)
- **Function: Represent patterns of strings of characters** [描述字符串的模式]
- Languages that can be expressed using regular expressions are called **Regular Languages** [正则语言]. More complex languages need more complex notations.



Build Regular Expressions[构建正则表达式]



- The regular expressions are built recursively out of **smaller** regular expressions.
- Each regular expression **r** denotes a language **L(r)**
 - ♦ defined recursively from the languages denoted by r's subexpressions.
- **Atomic[原子] Regular Expressions**
 - ♦ Smallest RE that cannot be broken down further
 - ♦ The symbol **ϵ** is a regular expression matches the empty string.
 - $L(\epsilon) = \{""\}$
 - ♦ For any symbol **a**, the symbol **a** is a regular expression that just matches **a**.
 - $L(a) = \{“a”\}$
 - ♦ Empty set is ϕ , not the same as ϵ
 - $\text{size}(\phi) = 0$; $\text{size}(L(\epsilon)) = 1$; $\text{length}(\epsilon) = 0$;



Build Regular Expressions[构建正则表达式]



- **Compound Regular Expressions**[复合正则表达式]
 - ◆ Large REs built from smaller ones
- Suppose r and s are REs denoting languages $L(r)$ and $L(s)$
 - ◆ (r) is a RE denoting the language $L(r)$
 - this says that we can add additional pairs of parentheses[小括号] around expressions without changing the language they denote.
 - ◆ $(r)|(s)$ is a RE denoting the language $L(r) \cup L(s)$
 - ◆ $(r)(s)$ is a RE denoting the language $L(r)L(s)$
 - ◆ $(r)^*$ is a RE denoting the language $(L(r))^*$
- REs often contain unnecessary $()$, which could be dropped
 - ◆ $(A) \equiv A$: A is a RE
 - ◆ $(a)|((b)^*(c)) \equiv a | b^*c$



Operator Precedence[运算符优先级]



- Regular expression operator precedence is

(A)

A*

A B

A | B

So $ab^*c|d$ is parsed as $((a(b^*))c)|d$

$a(b^*)c|d$

$(a(b^*))c|d$

$((a(b^*))c)|d$



Common REs[常用表达]



- One or more instances: $A^+ \equiv AA^*$
- Zero or one instance: $A? \equiv A \mid \varepsilon$
- Characters: $[a_1 a_2 \dots a_n] \equiv a_1 \mid a_2 \mid \dots \mid a_n$
- Range: $[a-z] \equiv 'a' \mid 'b' \mid \dots \mid 'z'$



Common REs[常用表达] Cont.



- Excluded range: **complement[补集] of $[a-z] \equiv [^a-z]$**
 - Symbol $^$ is also used to match the left end of a line. Symbol $$$ matches the right end of a line.
 - E.g., **$^[^aeiou]*$$** matches any complete line not containing a lower-case vowel[元音]
 - The context will make the meaning of $^$ clear.
- Identifier: strings of letters or digits, starting with a letter
 - letter = 'A' | ... | 'Z' | 'a' | ... | 'z' or,
 - letter = **$[A-Za-z]$**
 - digit = **$[0-9]$**
 - identifier = **letter (letter | digit) ***



RE Examples



Regular Expression	Explanation
a^*	0 or more a's (ϵ , a, aa, aaa, aaaa, ...)
a^+	1 or more a's (a, aa, aaa, aaaa, ...)
$(a \mid b)(a \mid b)$	(aa, ab, ba, bb)
$(a \mid b)^*$	all strings of a's and b's (including ϵ)
$(aa \mid ab \mid ba \mid bb)^*$	all strings of a's and b's of even length
$[a-zA-Z]$	shorthand for " $a \mid b \mid \dots \mid z \mid A \mid B \mid \dots \mid Z$ "
$[0-9]$	shorthand for " $0 \mid 1 \mid 2 \mid \dots \mid 9$ "
$0([0-9])^*0$	Quiz
$1^*(0 \mid \epsilon)1^*$	binary strings that contain at most one zero
$(0 \mid 1)^*00(0 \mid 1)^*$	Quiz

Q : are (a^*b^*) and $(ab)^*$ equivalent?

Q : are $(a \mid b)^*$ and $(a^*b^*)^*$ equivalent?



Different REs of the Same Language



- $(a|b)^* = ?$

$$\begin{aligned}(L(a|b))^* &= (L(a) \cup L(b))^* = (\{a\} \cup \{b\})^* = \{a, b\}^* \\ &= \{a, b\}^0 \cup \{a, b\}^1 \cup \{a, b\}^2 \cup \dots \\ &= \{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}\end{aligned}$$

- $(a^*b^*)^* = ?$

$$\begin{aligned}(L(a^*b^*))^* &= (L(a^*)L(b^*))^* \\ &= L(\{\epsilon, a, aa, \dots\}\{\epsilon, b, bb, \dots\})^* \\ &= L(\{\epsilon, a, b, aa, ab, bb, \dots\})^* \\ &= \epsilon \cup \{\epsilon, a, b, aa, ab, bb, \dots\}^1 \cup \{\epsilon, a, b, aa, ab, bb, \dots\}^2 \\ &\quad \cup \{\epsilon, a, b, aa, ab, bb, \dots\}^3 \cup \dots\end{aligned}$$

RE	Language
----	----------

$(r) (s)$	$L(r) \cup L(s)$
-----------	------------------

$(r)(s)$	$L(r)L(s)$
----------	------------

$(r)^*$	$(L(r))^*$
---------	------------



More Example

- Typical regular expression for tokens, let
 - ◆ RE: letter = [A-Za-z]
 - ◆ RE: digit = [0-9]
- **Keywords:** 'if', 'else', 'then', 'for'
 - ◆ RE: 'i' 'f' | 'e' 'l' 's' 'e' | ... = 'if' | 'else' | 'then' | ...
- **Unsigned Integer:** digit digit*
- **Whitespace:** a non-empty sequence of blanks, newline and tabs
 - ◆ ' ' | '\n' | '\t'



REs in Programming Language



Symbol	Meaning		
<code>\d</code>	Any decimal digit, i.e. [0-9]		
<code>\D</code>	Any non-digit char, i.e., [^0-9]		
<code>\s</code>	Any whitespace char, i.e., [\t\n\r\f\v]		
<code>\S</code>	Any non-whitespace char, i.e., [^ \t\n\r\f\v]		
<code>\w</code>	Any alphanumeric char, i.e., [a-zA-Z0-9_]		
<code>\W</code>	Any non-alphanumeric char, i.e., [^a-zA-Z0-9_]		
<code>.</code>	Any char	<code>\.</code>	Matching “.”
<code>[a-f]</code>	Char range	<code>[^a-f]</code>	Exclude range
<code>^</code>	Matching string start	<code>\$</code>	Matching string end
<code>(...)</code>	Capture matches		

<https://docs.python.org/3/howto/regex.html>



Lexical Specification of a Language



- **S0:** write a regex for the lexemes of each token class
 - ◆ Numbers = digit^+
 - ◆ Keywords = 'if' | 'else' | ...
 - ◆ Identifiers = $\text{letter}(\text{letter} \mid \text{digit})^*$
- **S1:** construct R , matching all lexemes for all tokens
 $R = \text{numbers} + \text{keywords} + \text{identifiers} + \dots = R_1 + R_2 + R_3 + \dots$
- **S2:** let input be $x_1 \dots x_n$, for $1 \leq i \leq n$, check $x_1 \dots x_i \in L(R)$
- **S3:** if successful, then we know $x_1 \dots x_i \in L(R_j)$ for some j
- **S4:** remove $x_1 \dots x_i$ from input and go to step S2



- Some strings can be matched by different regular expressions
- Language definition must give disambiguating rules
 - ◆ When a string can be either an identifier or a keyword, **keyword interpretation is preferred**[关键字优先识别]
 - ◆ Always choose the longer token to match (**Maximal match**[最长匹配])
 - ◆ Rule of thumb: choose **the one listed first**[匹配顺序]
 - ◆ if no rule matches?
 - $x_1 \dots x_i \notin L(R) \rightarrow \text{Error}$

```
if (a==3.14)  
    stmt1;  
else  
    stmt2;
```

'==' will always be identified first
due to the rule of Maximal match



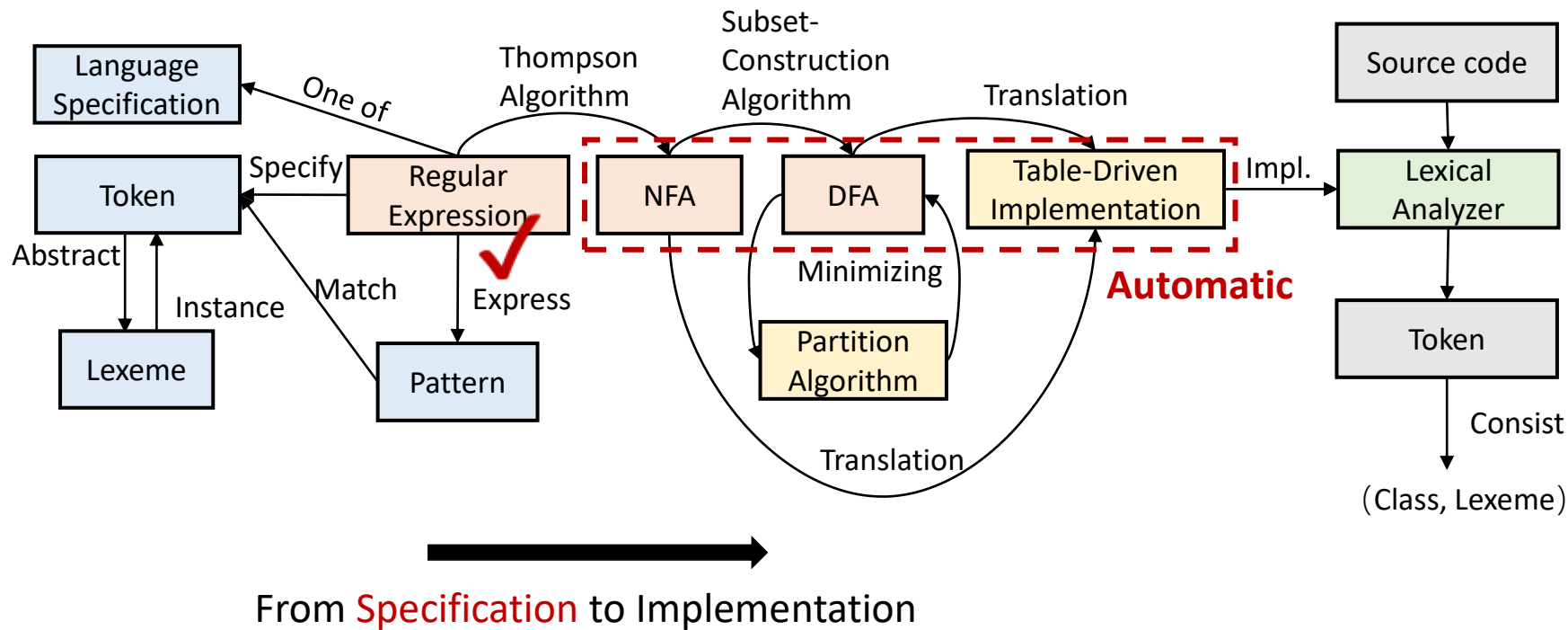
Summary



- Concepts
 - Token & Lexeme
 - Alphabet & String & Language
- Use Regular expressions to specify tokens for lexical analysis.
- Build Regular Expressions.
- Regular expression is only a language specification:
 - ◆ An implementation is still needed
 - ◆ Next: to construct a token recognizer for languages given by regular expressions – by using **finite automata**.



Summary



Further Reading

- Dragon Book

- ◆ Comprehensive Reading:

- Section 1.1, 1.2, 1.6
 - Section 2.6 and 3.1–3.2 for introduction to scanner.
 - Section 3.3 for regular expressions and regular definitions.

- ◆ Skip Reading:

- Section 1.3, 1.4, 1.5
 - Section 3.4–3.5 and 3.8 for scanner generator.

