编译原理
Complier Principles

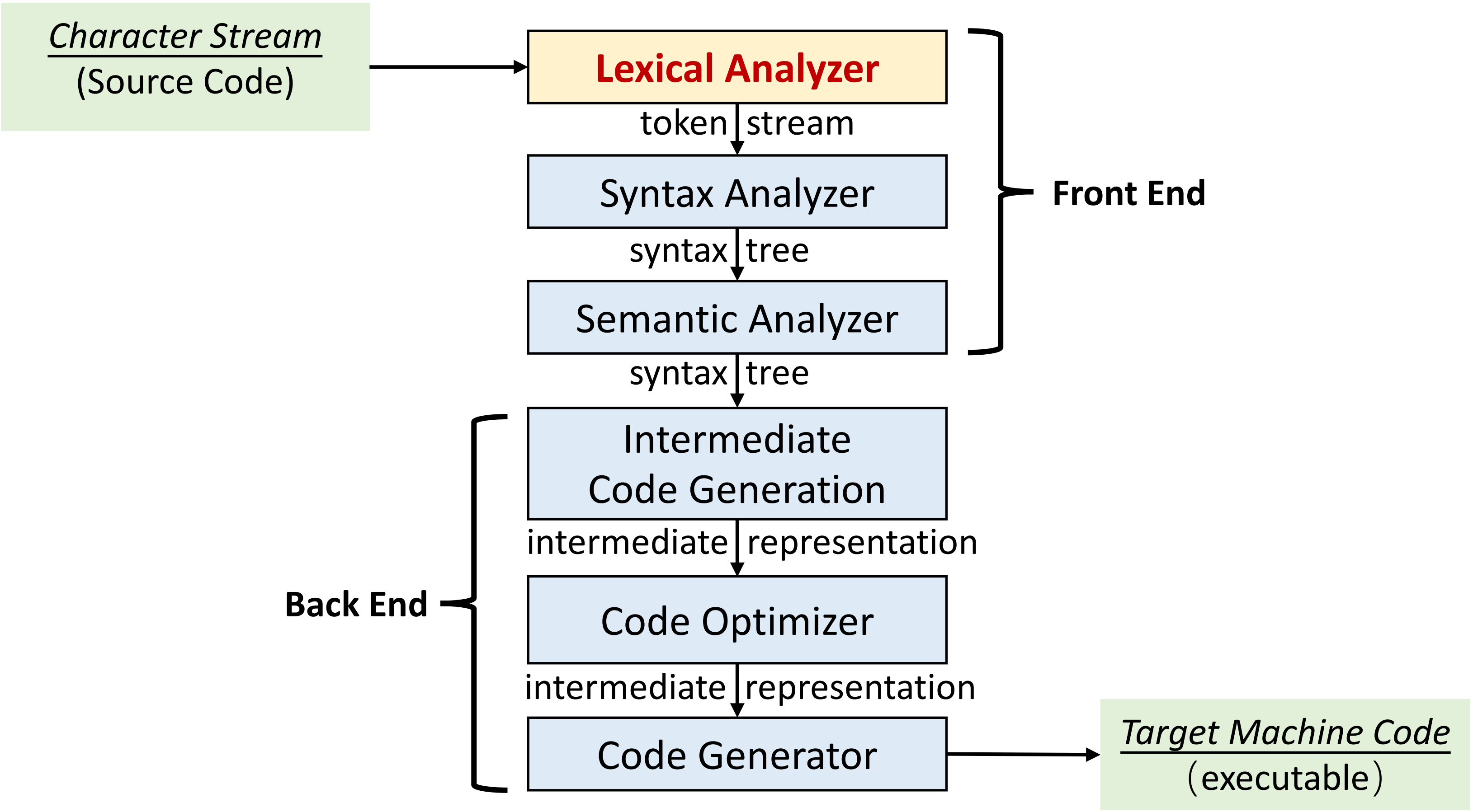
Lecture1
Lexical Analysis: Intro & Regular Expressions

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** by identifying the token class.
- **Task:** Reading the source program as a string of characters and diving it up into tokens.

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

- **Step**
  1. Remove comments: ~~/* simple example */~~
  2. Identify substrings: ‘while’ ‘(’ ‘i’ ‘==’ ‘j’……
  3. Identify token classes: (keyword, ‘while’), (lpar, ‘(’), (id, ‘i’), (rpar, ‘)’)……

What is Lexical Analysis[词法分析]?

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

- **Example**

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’ $\xrightarrow{\text{Token class}}$ (keyword, ‘while’)

Token & Lexeme [词法单元&词素]

- **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 (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

Quiz

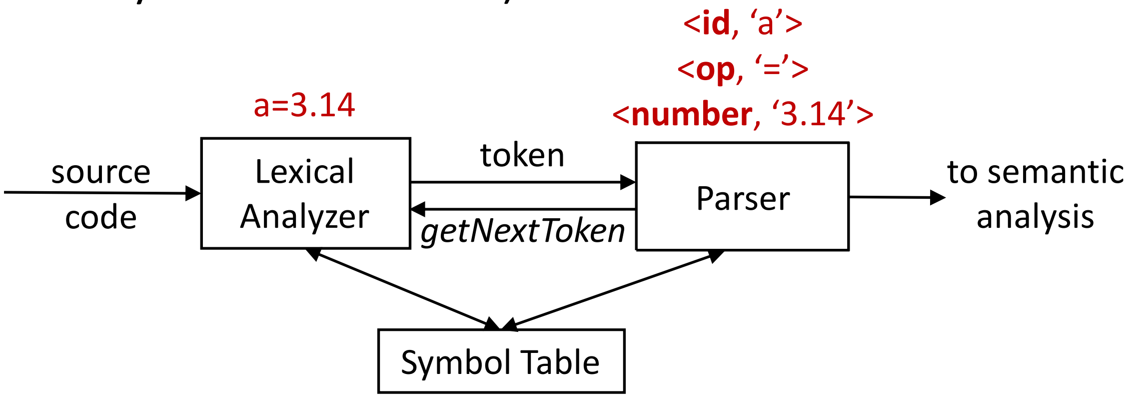
- **Which of the following names is NOT accepted by Java?**

  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).
  - 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 [定义词法单元的类别]
  - **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?

Lexical Analysis: Implementation

- **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, ect.)[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]

Challenges in Scanning

中山大學 SUN YAT-SEN UNIVERSITY

- **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

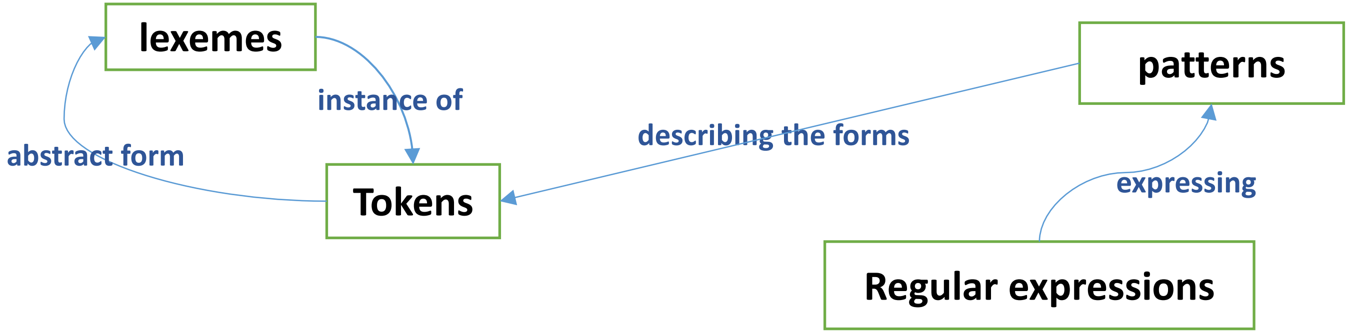
Q: Is '>>' a stream operator or two consecutive brackets?

Look Ahead[展望]

- "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
  - 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]

- **Question:** How to describe string patterns?[模式]
  - i.e., which set of strings belong to which token class?
  - 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 implementation can be generated automatically from specification (using a translation tool)



Language: Definition

SUN YAT-SEN UNIVERSITY

- **Alphabet ∑**[字母表]: A finite set of symbols.
  - Symbol: letter, digit, punctuation, …
  - Example: {0, 1}, ASCII {a, b, c}, …
- **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: |ε|= 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]
  - ∑ = {a, b}, then {}, {ab, ba}, {a, aa, aaaa, …} are all languages over ∑
  - An empty set[空集] (Φ) is a language
  - The set containing only an empty string ({ε}) is also a language.
    - Φ and {ε} 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 vaild integers
    - E.g., 125+, 1-25

Language: Operations[语言运算]

- **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 $\boldsymbol{AB}$
- **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 = {ε}, $L^{i} = L^{i-1}L$;
  - $L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \ldots$
  - $L^+ = L^1 \cup L^2 \cup L^3 \cup \ldots$(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}**
- ***L*∪*D*** = {a, b} ∪ {0, 1} = {a, b, 0, 1}
- ***LD*** = {a, b}{0, 1} = {a0, b0, a1, b1}
- $\boldsymbol{L^3}$= {a, b}^{3} = {a, b}{a, b}{a, b} = {aaa, aab, aba, abb, baa, bab, bba, bbb}
- $\boldsymbol{L^*}$= $\bigcup_{i=0}^{\infty} L^i = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \ldots$

  = {ε} ∪ {a, b} ∪ {a, b}^{2} ∪ {a, b}^{3}…

  = {ε, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, …}
- $\boldsymbol{L^+}$ = $\bigcup_{i=1}^{\infty} L^i$ =

  = {a, b} ∪ {a, b}^{2} ∪ {a, b}^{3}…

  = {a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, …}

Example Cont.

- **$L = \{A, B, \ldots, Z, a, b, \ldots, z\}$, $D = \{0, 1, \ldots, 9\}$**
  - 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
- $L4$: the set of all 4-letter strings
- L^*: the set of all strings of letters, including ε, the empty string
- $L(L \cup D)^*$: the set of all strings of letters and digits beginning with a letter
- D^+: the set of all strings of one or more digits

Regular Expressions & Languages

- **Regular expressions** [正则表达式] are to describe all the 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**
- 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(ε) = {“”}
  - 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
    - size(ϕ) = 0; size(ε) = 1; length(ε) = 0;

Build Regular Expressions[构建正则表达式]

- **Compound Regular Expressions**
  - Large REs built from smaller ones
- Suppose r and s are REs denoting languages L(r) and L(s)
  - (r)|(s) is a RE denoting the language L(r) ∪ L(s)
  - (r)(s) is a RE denoting the language L(r)L(s)
  - (r)* is a RE denoting the language (L(r))*
  - (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.
- REs often contain unnecessary (), which could be dropped
  - (A) ≡ A: A is a RE
  - (a)|((b)*(c)) ≡ 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: $[\mathrm{a}_1 a_2 \ldots a_n] \equiv \mathrm{a}_1 \mid a_2 \mid \ldots \mid a_n$

- Range: 'a' | 'b' | … | 'z' ≡ [a-z]

Common REs[常用表达] Cont.

- Excluded range: complement[补集] of [a-z] ≡ [^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
  - 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 | b)(a | b)	(aa, ab, ba, bb)
(a | b)*	all strings of a's and b's (including ε)
(aa | ab | ba | bb)*	all strings of a's and b's of even length
[a-zA-Z]	shorthand for "a|b|…z|A|B|…|Z"
[0-9]	shorthand for "0|1|2|…|9"
0([0-9])*0	numbers that start and end with 0
1*(0|ε)1*	binary strings that contain at most one zero
(0|1)*00(0|1)*	all binary strings that contain '00' as substring

Q : are (a|b)* and (a*b*)* equivalent?

Different REs of the Same Language

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

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

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

$$(L(a^*b^*))^* = (L(a^*)L(b^*))^*$$
$$= L(\{\varepsilon, a, aa, \dots\}\{\varepsilon, b, bb, \dots\})^*$$
$$= L(\{\varepsilon, a, b, aa, ab, bb, \dots\})^*$$
$$= \varepsilon \cup \{\varepsilon, a, b, aa, ab, bb, \dots\}^1 \cup \{\varepsilon, a, b, aa, ab, bb, \dots\}^2 \cup \{\varepsilon, a, b, aa, ab, bb, \dots\}^3 \cup \dots$$

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		
\d	Any decimal digit, i.e. [0-9]		
\D	Any non-digit char, i.e., [^0-9]		
\s	Any whitespace char, i.e., [\t\n\r\f\v]		
\S	Any non-whitespace char, i.e., [^ \t\n\r\f\v]		
\w	Any alphanumeric char, i.e., [a-zA-Z0-9_]		
\W	Any non-alphanumeric char, i.e., [^a-zA-Z0-9_]		
.	Any char	\\.	Matching "."
[a-f]	Char range	[^a-f]	Exclude range
^	Matching string start	$	Matching string end
(…)	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 = letter(letter | digit)*
- S1: construct R, matching all lexemes for all tokens

  R = numbers + keywords + identifiers + … = $R_1 + R_2 + R_3 + \ldots$
- S2: let input be $x_1 \ldots x_n$, for 1 ≤ i ≤ n, check $x_1 \ldots x_i$ ∈ L(R)
- S3: if successful, then we know $x_1 \ldots x_i \in L(R_j)$ for some j
- S4: remove $x_1 \ldots x_i$ from input and go to step S2

Ambiguity[二义性]

- 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?
    - $x_1 \ldots x_i \not\subset L(R)$ → Error

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

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

Summary

- 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.

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.