Compilation Principle
编 译 原 理

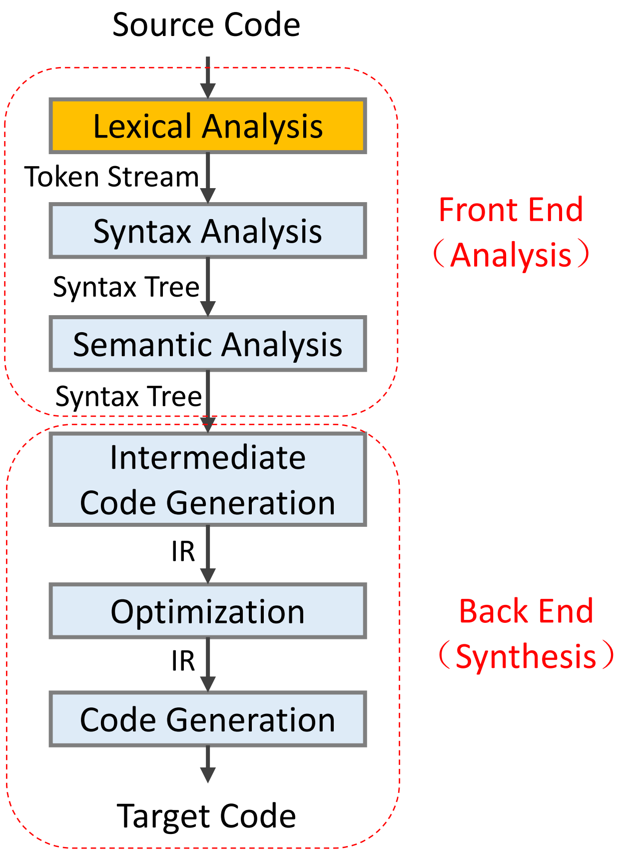
第1讲：词法分析

张献伟

xianweiz.github.io

DCS290, Spring 2021

Structure of a Typical Compiler



What is Lexical Analysis[词法分析]?

- Example:

```
/* simple example */
if (i == j)
  z = 0;
else
  z = 1;
```

- Input： a string of characters
  - “*if* (*i* == *j*)*n**t* *tz* = 0; *telse**n**tz* = 1; *n*”
- Goal: partition the string into a set of substrings
  - Those substrings are **tokens**
- Steps:
  - Remove comments
  - Identify substrings: ‘if’ ‘(‘ ‘i’ ‘==‘ ‘j’ ......
  - Identify **token classes**: (keyword, ‘if’), (LPAR, ‘(‘), (id, ‘i’) ......

What is a token[词]?

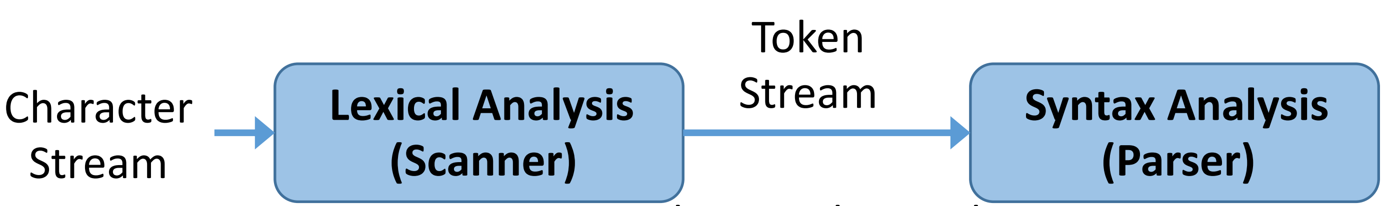
- **Token**: a "word" in language (smallest unit with meaning)
  - Categorized into classes according to its role in language
  - Token classes in English:
    - Noun, verb, adjective, …
  - Token classes in a programming language
    - Number, keyword, whitespace, identifier, …
- Each **token class** corresponds to a set of strings
  - Numbers: a non-empty string of 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 (Q: what are the rules in C language?)

Lexical Analysis: Tokenization[分词]？

- Lexical analysis is also called **Tokenization** (also called <u>Scanner</u>)
  - Partition input string into a sequence of tokens
  - Classify each token according to roles (token class)
    - **Lexeme**: an instance of the corresponding token class, e.g. ‘z’, ‘=‘, ‘1’
- Pass tokens to syntax analyzer (also called <u>Parser</u>)
  - Parser relies on token classes to identify roles (e.g., a keyword is treated differently than an identifier)

z = 1

<Id, ‘z’>
<Op, ‘=‘>
<Num, ‘1’>



Token = <class, value>

Lexical Analyzer: Design

- Define a finite set of token classes
  - Describe all items of interest
  - Depends on language, design of parser
  - “*if* (*i* == *j*)\n\t \t*z* = 0; \t*else*\n\t*z* = 1; \n”
    - Keyword, identifier, whitespace, integer

- Label which string belongs to which token class

```
if (i == j)
  z = 0;
else
  z = 1;
```

'==‘ or ‘=‘?

keyword or identifier?

Lexical Analyzer: Implementation

- An implementation must do two things
  - Recognize the token class the substring belongs to
  - Return the value or lexeme of the token
- A token is a tuple (class, lexeme)
- The lexer usually discards “non-interesting” tokens that don’t contribute to parsing, e.g., whitespace, comments
- If token classes are non-ambiguous, tokens can be recognized in a single left-to-right scan of input string
- Problem can occur when classes are ambiguous

Ambiguous Tokens in C++

- C++ template syntax
  - Foo<Bar>
- C++ stream syntax
  - cin >> var
- Ambiguity
  - Foo<Bar<Bar>>
  - cin >> var
  - Q: Is ‘>>’ a stream operator or two consecutive brackets?

```
Template <typename T>
T getMax(T x, T y) {
   return (x > y) ? x : y;
}

int main (int argc, char* argv[]) {
    getMax<int>(3, 7);
    getMax<double>(3.0, 2.0);
    getMax<char>(‘g’, ‘e’);

    return 0;
}
```

Look Ahead

- ”look ahead” may be required to resolve ambiguity
  - Extracting some tokens requires looking at the larger context or structure
  - Structure emerges only at parsing stage with parse tree
  - Hence, sometimes feedback from parser needed for lexing
    - This complicates the design of lexical analysis
    - Should minimize the amount of look ahead

- Usually tokens do not overlap
  - Tokenizing can be done in one pass w/o parser feedback
  - Clean division between lexical and syntax analyses

Summary: Lexer

- Lexical Analysis
  - Partition the input string to lexeme
  - Identify the token class of each lexeme
- Left-to-right scan => look ahead may be required
  - In reality, lookahead is always needed
  - The amount of lookahead should be minimized

```
/* simple example */
if (i == j)
  z = 0;
else
  z = 1;
```

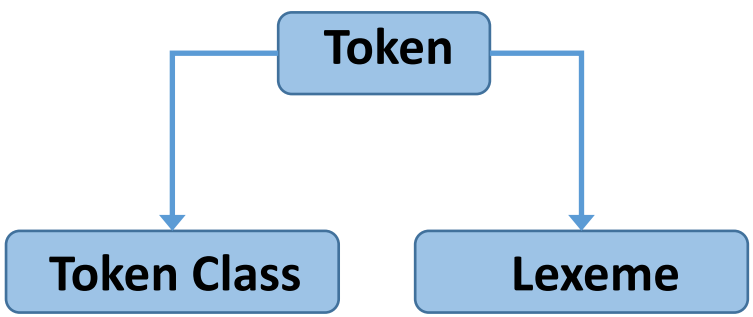
```
if (i == j)
  z = 0;
else
  z = 1;
```

‘if’ ‘(‘ ‘i’ ‘==‘ ‘j’ ‘)’ ‘\n’ ‘\t’ ‘z’ ‘=‘ ‘0’ ‘;’
‘\n’ ‘else’ ‘\n’ ‘\t’ ‘z’ ‘=‘ ‘1’

<keyword, if> <LPAR, (), <id, i>, <op, ==)
… …

Token Specification

- Recognizing token class: how to describe string patterns
  - 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 powerful (able to express patterns)
  - Tokenizer implementation can be generated automatically from specification (using a translation tool)
  - Resulting implementation is provably efficient



String patterns
describing the class

Language: Definition

- **Alphabet** ∑[字母表]: a finite set of symbols
  - Symbol: letter, digit, punctuation, …
  - Example: {0, 1}, {a, b, c}, ASCII
- **String** [串]: a finite sequence of symbols drawn from ∑
  - Example: aab (length = 3), ε (empty string, length = 0)
- **Language** [语言]: a set of strings of the characters drawn from ∑
  - ∑ = {0, 1}, then {}, {01, 10}, {1, 11, 1111, …} are all languages over ∑
  - {ε} is a language
  - Φ, empty set is also a language

Language: Example

- Examples:
  - Alphabet $\sum$ = (set of) English characters
  - Language L = (set of) English sentences
  - Alphabet $\sum$ = (set of) Digits, +, -
  - Language L = (set of) Integer numbers

- Languages are subsets of all possible strings
  - Not all strings of English characters are sentences
  - Not all sequences of digits and signs are integers

Regular Expressions and Languages

- Need a notion to specify strings in a particular language
  - More complex languages need more complex notations
- 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)
- Languages that can be expressed using regular expressions are called Regular Languages

- More complex languages and expressions will be covered later

Atomic REs[原子]

- Atomic
  - Smallest RE that cannot be broken down further

- **Epsilon or ε** character denotes a zero length string
  - ε = {“”}
- **Single character** denotes a set of one string
  - ‘c’ = {“c”}
- Empty set is { } = ϕ, not the same as ε
  - Size(ϕ) = 0
  - Size(ε) = 1
  - Length(ε) = 0

Compound REs[组合]

- **Union**[并]: if A and B are REs, then

  $A|B = \{ s \mid s \in A \text{ or } s \in B \}$

- **Concatenation**[连接] of sets/strings

  $AB = \{ ab \mid a \in A \text{ and } b \in B \}$

- **Iteration**[迭代] (Kleene closure)

  $A^* = \cup_{i \geq 0} A^i$ where $A^i = A...A$ (i times)

  in particular

  $A^* = \{\varepsilon\} + A + AA + AAA + ...$

  $A+ = A + AA + AAA + ... = AA^*$

- **(A) ≡ A**: A is a RE

SUN YAT-SEN UNIVERSITY 中山大學

RE and RL

- The regular expressions (REs) over ∑ are the total set of expressions that can be constructed using components:
  - ε
  - ‘c’ where c ∈ ∑
  - A|B where A, B are REs over ∑
  - AB where A, B are REs over ∑
  - A* where A is a RE over ∑
- The regular languages (RLs) over ∑ are the total set of languages that can be expressed using REs:
  - L(ε) = {“”}
  - L(’c’) = {“c”}
  - L(A|B) = L(A) ∪ L(B)
  - L(AB) = { ab | a ∈ L(A) and b ∈ L(B) }
  - $L(A^*)=\cup_{i\geq 0} L(A^i)$

SUN YAT-SEN UNIVERSITY 中山大學

Operator Precedence[优先级]

- RE operator precedence
  - (A)
  - A*
  - AB
  - A|B

- Example: ab*c|d
  - a<u>(b*)</u>c|d
  - <u>(a(b*))</u>c|d
  - <u>((a(b*))c)</u>|d

Common REs

- **At least one**: A+ ≡ AA*
- **Union**: A|B ≡ A+B
- **Option**: A? ≡ A + ε
- **Range**: ‘a’ + ‘b’ + … + ‘z’ ≡ [a-z]
- **Excluded range**: complement of [a-z] ≡ [^a-z]

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?

More Examples

- Keywords: ‘if’ or ‘else’ or ’then’ or ‘for’ …
  - RE = ‘i’’f’ + ‘e’’l’’s’’e’ + … = ‘if’ + ‘else’ + ‘then’ + …
- Numbers: a non-empty string of digits
  - digit = ‘0’ + ‘1’ + ‘2’ + ‘3’ + ‘4’ + ‘5’ + ‘6’ + ‘7’ + ‘8’ + ‘9’
  - integer = digit digit*
  - Q: is ‘000’ an integer?
- Identifier: strings of letters or digits, starting with a letter
  - letter = ‘a’ + ‘b’ + … ‘z’ + ‘A’ + ‘B’ + … + ‘Z’ = [a-zA-Z]
  - RE = letter(letter + digit)*
  - Q: is the RE valid for identifiers in C?
- 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 + … = R1 + R2 + R3 + …

- S2: let input be $x_q \ldots x_n$, for $1 \leq i \leq n$, check $x_1 \ldots x_i \in L(R)$
- 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

Lexical Specification of a Language

- How much input is used?
  - $x_1 \ldots x_i \in L(R)$, $x_1 \ldots x_j \in L(R)$, $i \neq j$
  - Which one do we want? (e.g., ‘==‘ or ‘=‘)
  - Maximal match: always choose the longer one
- Which token is used if more than one matches?
  - $x_1 \ldots x_i \in L(R)$ where $R = R_1 + R_2 + \ldots + R_n$
  - $x_1 \ldots x_i \in L(R_m)$, $x_1 \ldots x_i \in L(R_n)$, $m \neq n$
  - E.g., keywords = ‘if’, identifier = letter(letter+digit)*
  - Keyword has higher priority
  - Rule of thumb: choose the one listed first
- What if no rule matches?
  - $x_1 \ldots x_i \notin L(R) \rightarrow$ Error

Summary: RE

- We have learnt how to specify tokens for lexical analysis
  - Regular expressions
  - Concise notations for the string patterns
- Used in lexical analysis with some extensions
  - To resolve ambiguities
  - To handle errors
- REs 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**