Compilation Principle
编 译 原 理

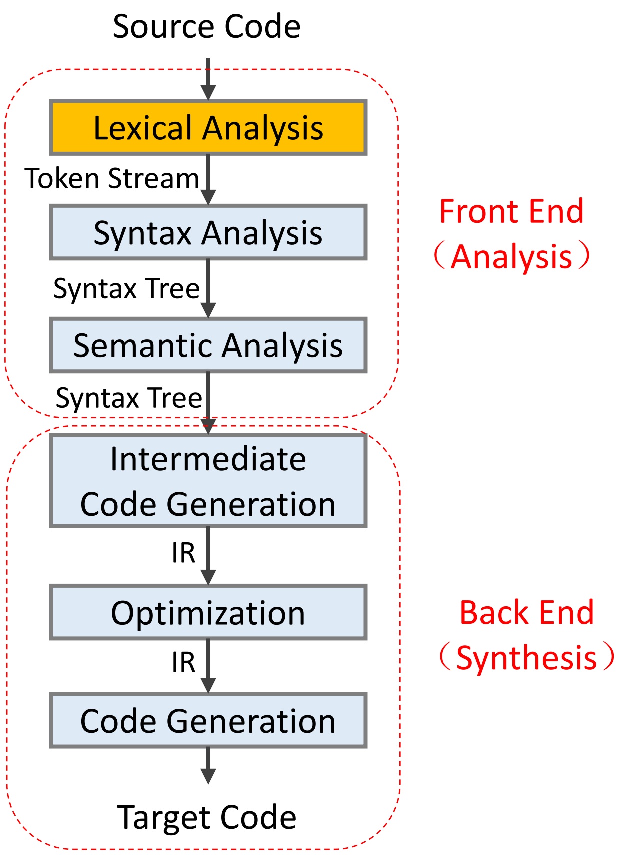
第1讲：词法分析(1)

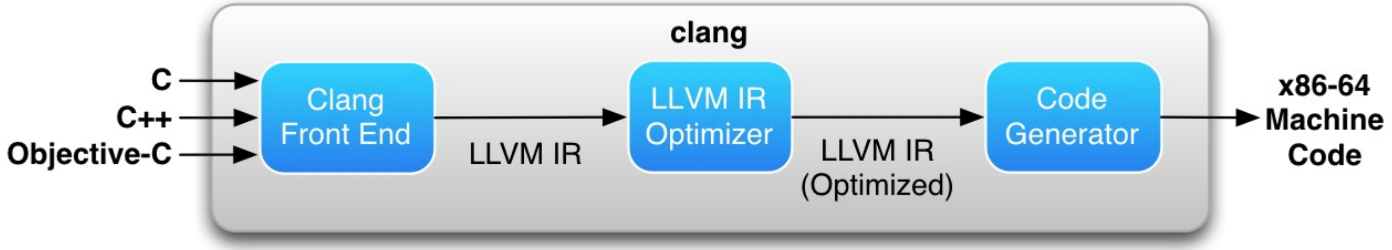
张献伟

xianweiz.github.io

DCS290, 2/23/2023

Structure of a Typical Compiler[结构]





What is Lexical Analysis[词法分析]？

- Example:

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

- Input[输入]: a string of characters
  - “*if* (*i* == *j*)*n**tz* = 0; *nelse**n**tz* = 1; *n*”
- Goal[目标]: partition the string into a set of substrings
  - Those substrings are **tokens**
- Steps[步骤]
  - Remove comments: ~~/* simple example */~~
  - Identify substrings: ‘if’ ‘(‘ ‘i’ ‘==‘ ‘j’ ......
  - Identify **token classes**: (keyword, ‘if’), (LPAR, ‘(‘), (id, ‘i’) ......

What is a token[词]?

- **Token**: a “word” in language (<u>smallest unit with meaning</u>)
  - Categorized into classes according to its role in language
  - Token classes in English[自然语言]
    - Noun, verb, adjective, …
  - Token classes in a programming language[编程语言]
    - Number, keyword, whitespace, identifier, …

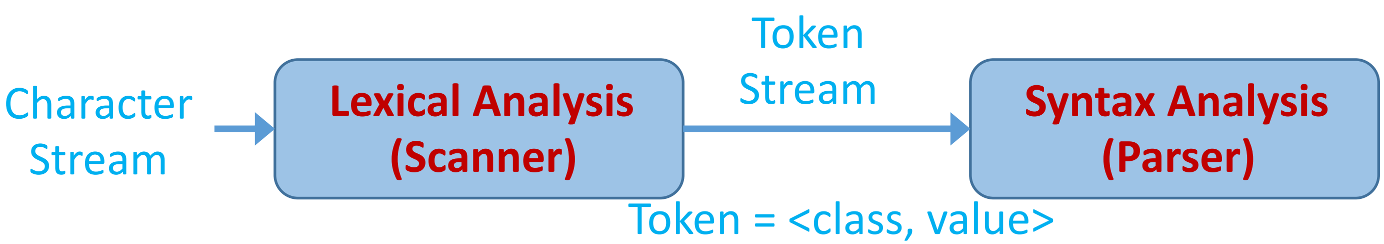
- Each **token class** corresponds to a set of strings[类: 集合]
  - Number: a non-empty string of digits
  - Keyword: a fixed set of reserved words (“for”, “if”, “else”, …)
  - Whitespace: a non-empty sequence of blanks, tabs, newlines
  - Identifier: user-defined name of an entity to identify
    - Q: what are the rules in C language?

Lexical Analysis: Tokenization[分词]

- Lexical analysis is also called **Tokenization** (also called Scanner)[扫描器]
  - Partition input string into a sequence of tokens
  - Classify each token according to its role (token class)
    - **Lexeme**[词素]: an instance of the token class, e.g. ‘z’, ‘=‘, ‘1’
- Pass tokens to syntax analyzer (also called Parser)[分析器]
  - Parser relies on token classes to identify roles (e.g., a *keyword* is treated differently from an *identifier*)

z = 1

<Id, ‘z’>
<Op, ‘=‘>
<Num, ‘1’>



Lexical Analyzer: Design[设计]

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

- Label which string belongs to which token class[识别]

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

'==‘ or ‘=‘?

keyword or identifier?

Lexical Analyzer: Implementation[实现]

- An implementation must do two things
  - Recognize the token class the substring belongs to[识别分类]
  - Return the value or lexeme of the token[返回数值]
- A token is a tuple (class, lexeme)[二元组]

- The lexer usually discards “non-interesting” tokens that don’t contribute to parsing[丢弃无意义词]
  - 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[歧义]

Ambiguous Tokens in C++

- C++ template syntax
  - Foo<Bar>
- C++ stream syntax
  - cin >> var

```
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;
}
```

> Template: a blueprint or formula for creating a generic class or a function.
> Templates are expanded at compiler time, similar to macros.

- Ambiguity
  - Foo<Bar<Bar>>
  - cin >> var
  - Q: Is '>>' a stream operator or two consecutive brackets?

Look Ahead[展望]

- ”look ahead” may be required to resolve ambiguity[展望消除歧义]
  - Extracting some tokens requires looking at the larger context or structure[需要上下文或语法结构]
  - Structure emerges only at parsing stage with parse tree[后一阶段才有]
  - 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[通常无重叠]
  - 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;
```

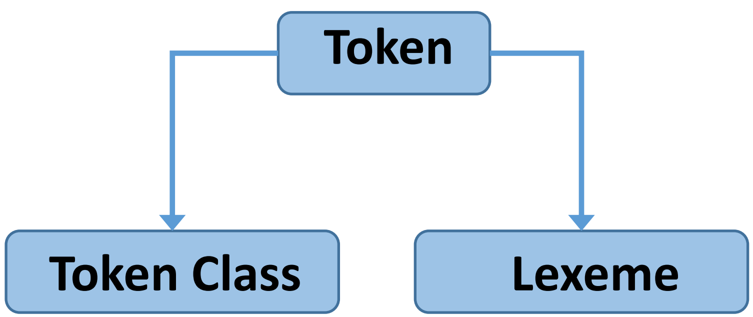
‘if’ ‘(‘ ‘i’ ‘==‘ ‘j’ ‘)’ ‘\n’ ‘\t’ ‘z’ ‘=‘ ‘0’ ‘;’ ‘\n’ ‘else’ ‘\n’ ‘\t’ ‘z’ ‘=‘ ‘1’

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

… …

Token Specification[定义]

- 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 ∑
  - i.e., sentence or word
  - 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

SUN YAT-SEN UNIVERSITY 中山大學

NSCC GZ

Language: Example

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

- Languages are <u>subsets of all</u> possible strings
  - Not all strings of English characters are (valid) sentences
    - aaa bbb ccc
  - Not all sequences of digits and signs are integers
    - 125+, 1-25

Language: Operations[语言运算]

- In lexical analysis, the most important operations on languages are <u>union</u>, <u>concatenation</u> and <u>closure</u>
- **Union**[并]: similar operation on sets
- **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, and concatenating them
- **Kleene closure**[闭包]: L^*, where L is the language, is the set of strings you get by concatenating L zero or more times
  - $L^0 = \{\varepsilon\}$, $L^i = L^{i-1}L$
  - L^+: the same as Kleene closure, but without L^0
    - $L \cup L^2 \cup L^3 \cup \ldots$
    - ε won't be in L^+ unless it is in L itself

Example

- $\sum_1 = \{0, 1\}$, $\sum_2 = \{a, b\}$ ==> $L_1 = \{0, 1\}$, $L_2 = \{a, b\}$
- $L_1 \cup L_2$
  $= \{0, 1\} \cup \{a, b\} = \{0, 1, a, b\}$
- L_1L_2
  $= \{0, 1\}\{a, b\} = \{0a, 0b, 1a, 1b\}$
- L_1^3
  $= \{0, 1\}^3 = \{0, 1\}\{0, 1\}\{0, 1\} = \{000, 001, 010, 011, 100, 101, 110, 111\}$
- L_1^*
  $= L_1^0 \cup L_1^1 \cup L_1^2 \cup L_1^3 \cup \ldots = \{\varepsilon\} \cup \{0, 1\} \cup \{0, 1\}^2 \cup \{0, 1\}^3$
  $= \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, \ldots\}$
- L_1^+
  - $= L_1^* - L_1^0 = \{0, 1\} \cup \{0, 1\}^2 \cup \{0, 1\}^3$
  - $= \{0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, \ldots\}$

Language: 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

Identifiers can be described by giving names to sets of letters and digits and using the language operators

SUN YAT-SEN UNIVERSITY

Regular Expressions & Languages[正则]

- **Regular expressions** are to describe all the languages that can be built from the operators applied to the symbols of some alphabet
- Regular Expression is a <u>simple</u> 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 need more complex notations
  - More complex languages and expressions[非正则] will be covered later

Atomic REs[原子表达式]

- 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[组合表达式]

- **Compound**
  - 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)
    - We can add additional () around expressions without changing the language they denote
  - (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 $(L(r))^*$
- REs often contain unnecessary (), which could be dropped
  - **(A) ≡ A**: A is a RE
  - $(a)|((b)^*(c)) \equiv a|b^*c$

Operator Precedence[优先级]

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

- Example: $ab^*c|d$
  - $a\underline{(b^*)}c|d$
  - $\underline{(a(b^*))}c|d$
  - $\underline{((a(b^*))c)}|d$

Common REs[常用表达]

- **At least one**: $A+ \equiv AA^*$
- **Option**: $A? \equiv A \mid \varepsilon$
- **Characters**: $[a_1a_2\ldots a_n] \equiv a_1|a_2|\ldots|a_n$
- **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
$1^*(0\|\varepsilon)1^*$	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?

SUN YAT-SEN UNIVERSITY

Different REs of the Same Language

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

- $(a^*b^*)^* = ?$
  - $(L(a^*b^*))^* = (L(a^*)L(b^*))^*$
  - $= L(\{\varepsilon, a, aa, \ldots\}\{\varepsilon, b, bb, \ldots\})^*$
  - $= L(\{\varepsilon, a, b, aa, ab, bb, \ldots\})^*$
  - $= \varepsilon + \{\varepsilon, a, b, aa, ab, bb, \ldots\} + \{\varepsilon, a, b, aa, ab, bb, \ldots\}^2 + \{\varepsilon, a, b, aa, ab, bb, \ldots\}^3 + \ldots$

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')+

中山大學 SUN YAT-SEN UNIVERSITY

NSCC GZ

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 $x_q \ldots x_n$, for $1 \le i \le n$, 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
  - E.g., an identifier or a number …
- **S4**: remove $x_1 \ldots x_i$ from input and go to step S2

SUN YAT-SEN UNIVERSITY

Lexical Spec. of a Language(cont.)

- 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 '=')
  - <u>Maximal match</u>: always choose the longer one[最长匹配]
- 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
  - <u>Rule of thumb</u>: choose the one listed first[次序]
- 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[定义token]
  - Regular expressions
    - Concise notations for the string patterns
- Used in lexical analysis with some extensions[适度扩展]
  - To resolve ambiguities
  - To handle errors
- RE is only a language specification[只是定义了语言]
  - An implementation is still needed
  - Next: to construct a token recognizer for languages given by regular expressions – by using **finite automata**[有穷自动机]