Compilation Principle
编 译 原 理

第4讲：词法分析(4)

张献伟

xianweiz.github.io

DCS290, 3/7/2023

Quiz Questions

- Q1: write RE for binary numbers that are multipliers of 4?

  (0|1)*00

- Q2: lexical analysis of 'if (a != b'?

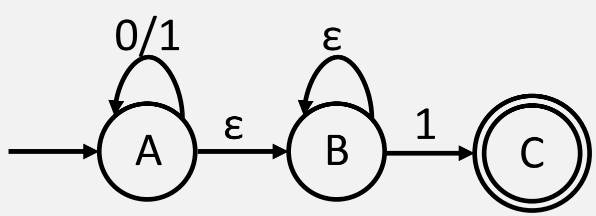
  (keyword, 'if'), (sym, '('), (id, 'a'), (sym, '!='), (id, 'b')

- Q3: regard lexer implementation, why NFA → DFA?

  Trade-off space for speed; DFA is more efficient

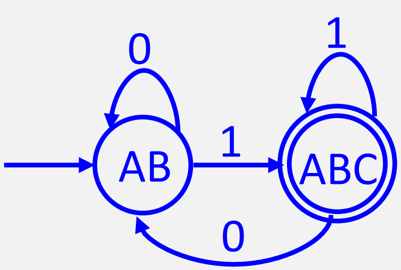
- Q4: RE of the FA?

  (0|1)*1



- Q5: start state of the equivalent DFA?

	0	1
AB	AB	ABC
ABC	AB	ABC



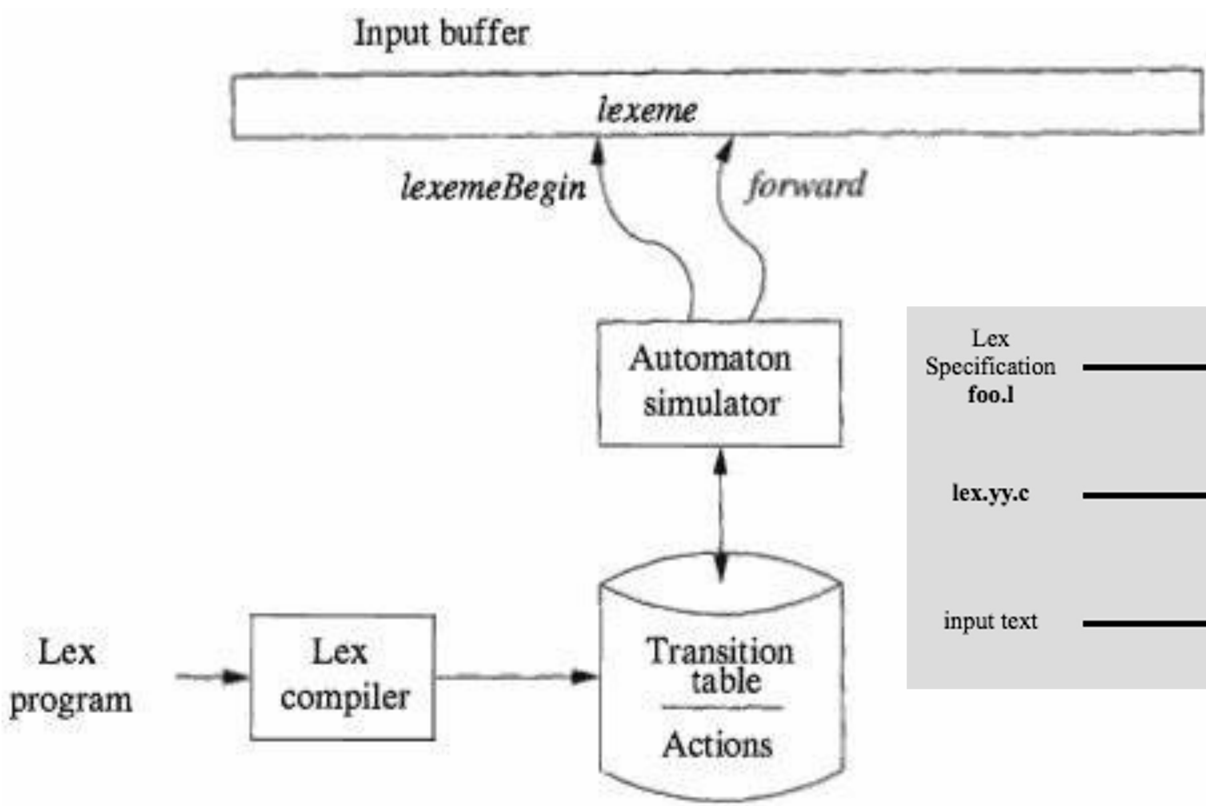
ε-closure(A) = {A, B}

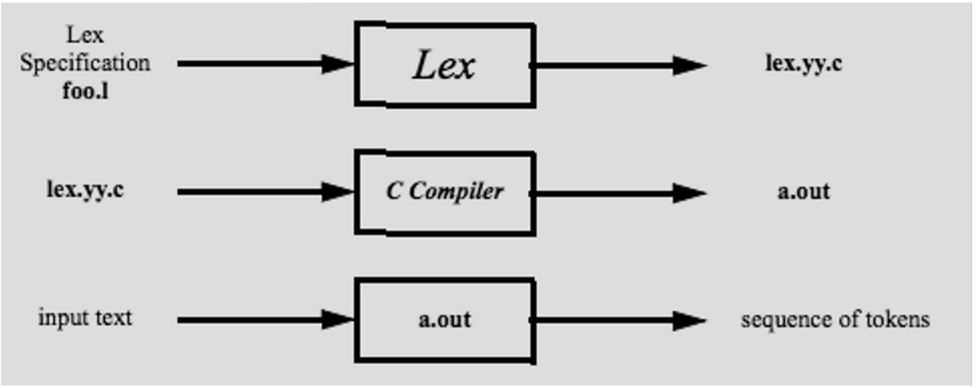
ε-closure(move({AB}, 0)) = ε-closure({A}) ⟹ {A, B}

ε-closure(move({AB}, 1)) = ε-closure({A,C}) ⟹ {A, B, C}

Lexical Analyzer Generated by Lex

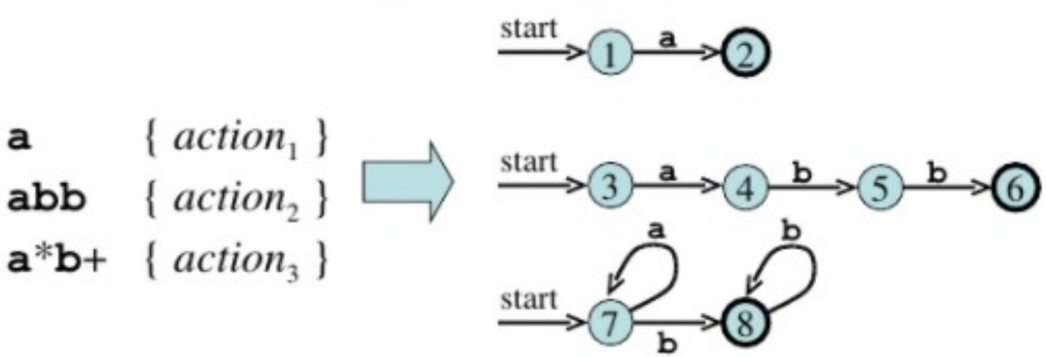
- A Lex program is turned into a transition table and actions, which are used by a FA simulator
- Automaton recognizes matching any of the patterns

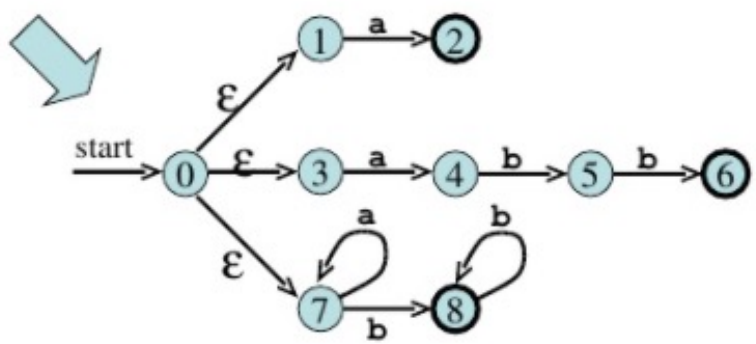




Lex: Example

- Three patterns, three NFAs
- Combine three NFAs into a single NFA
  - Add start state 0 and ε-transitions





Any one is possbile, if you haven’t read any input symbol ⟹

Lex: Example (cont.)

```
ptn1    a
ptn2    abb
ptn3    a*b+

%%

{ptn1}  { printf("\n<%s, %s>", "ptn1", yytext); }
{ptn2}  { printf("\n<%s, %s>", "ptn2", yytext); }
{ptn3}  { printf("\n<%s, %s>", "ptn3", yytext); }

%%

int main(){
  yylex();
  return 0;
}
```

$flex lex.l

$clang lex.yy.c -o mylex -ll

```
root@aa51dde06c76:~/test# echo "aaba" | ./mylex

<ptn3, aab>
<ptn1, a>
root@aa51dde06c76:~/test# echo "abba" | ./mylex

<ptn2, abb>
<ptn1, a>
```

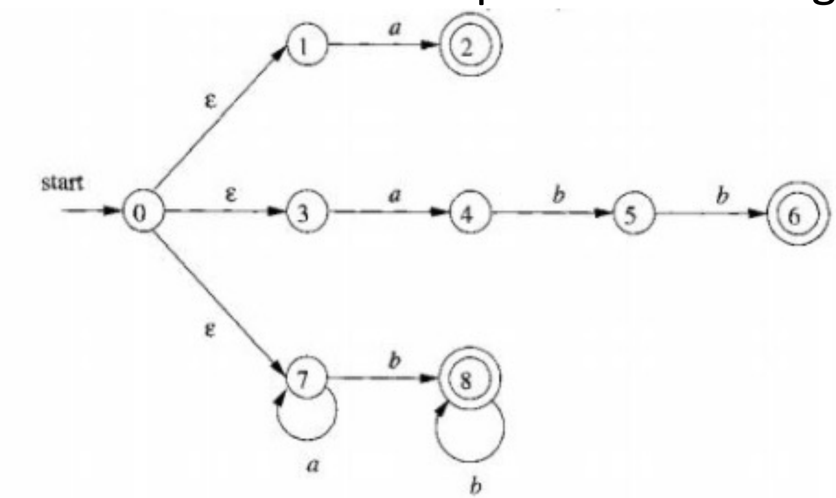
Lex: Example (cont.)

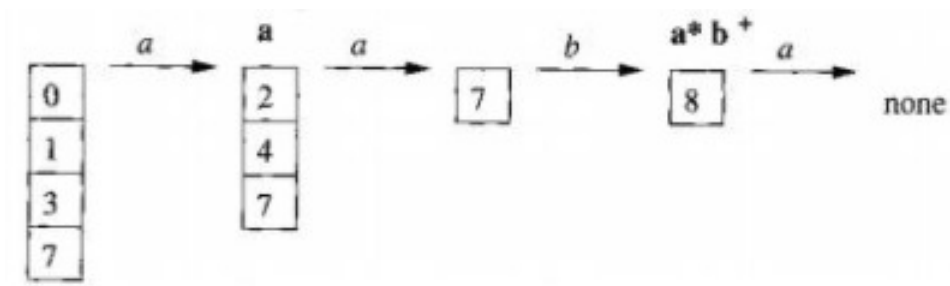
- NFA's for lexical analyzer
- Input: aaba
  - ε-closure(0) = {0, 1, 3, 7}
  - Empty states after reading the fourth input symbol
    - There are no transitions out of state 8
    - Back up, looking for a set of states that include an accepting state
  - State 8: a*b+ has been matched
    - Select aab as the lexeme, execute action_3
    - Return to parser indicating that token w/ pattern p_3=a*b+ has been found

a

abb

a*b+



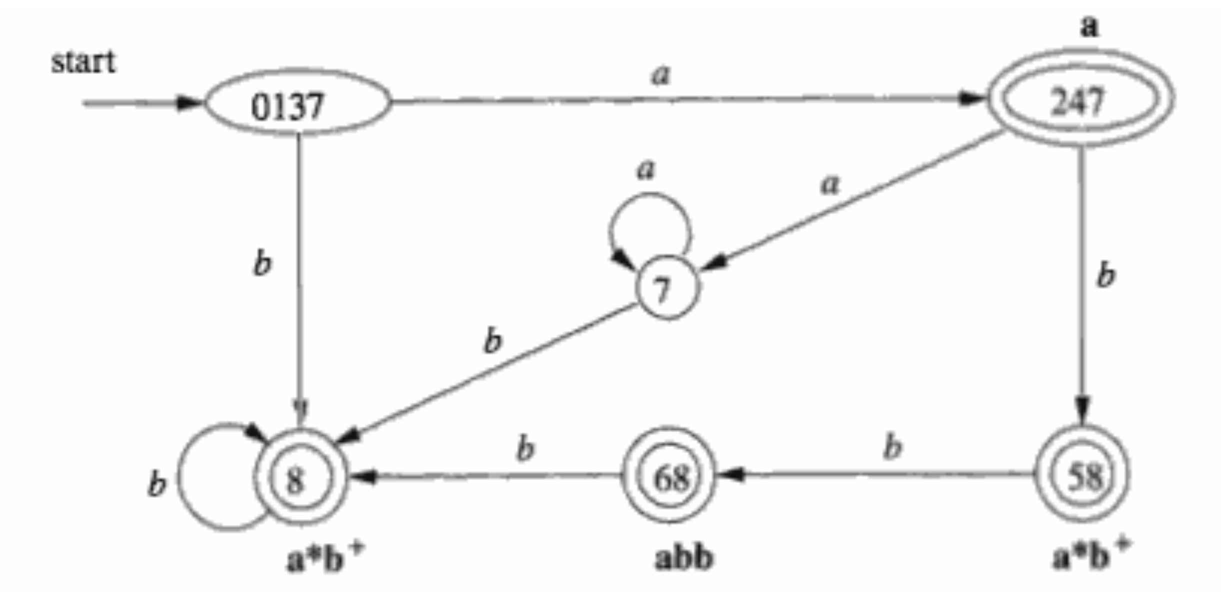


aaba → (a*b+, aab), (a, a)

Lex: Example (cont.)

- DFA's for lexical analyzer
- Input: abba
  - Sequence of states entered: 0137 → 247 → 58 → 68
  - At the final *a*, there is no transition out of state 68
    - 68 itself is an accepting state that reports pattern p_2 = abb

```
a
abb
a*b+
```



How Much Should We Match?[匹配多少]

- In general, find the longest match possible
  - We have seen examples
  - One more example: input string aabbb …
    - Have many prefixes that match the third pattern
    - Continue reading *b*’s until another *a* is met
    - Report the lexeme to be the intial *a*’s followed by as many *b*’s as there are

```
a      { action1 }
abb    { action2 }
a*b+   { action3 }
```

- If same length, rule appearing first takes precedence
  - String *abb* matches both the second and third
  - We consider it as a lexeme for p_2, since that pattern listed first

```
ptn1    a
ptn2    abb
ptn3    a*b+

%%

{ptn1}  { printf("\n<%s, %s>", "ptn1", yytext); }
{ptn2}  { printf("\n<%s, %s>", "ptn2", yytext); }
{ptn3}  { printf("\n<%s, %s>", "ptn3", yytext); }
```

<ptn2, abb>

```
ptn1    a
ptn2    abb
ptn3    a*b+

%%

{ptn1}  { printf("\n<%s, %s>", "ptn1", yytext); }
{ptn3}  { printf("\n<%s, %s>", "ptn3", yytext); }
{ptn2}  { printf("\n<%s, %s>", "ptn2", yytext); }
```

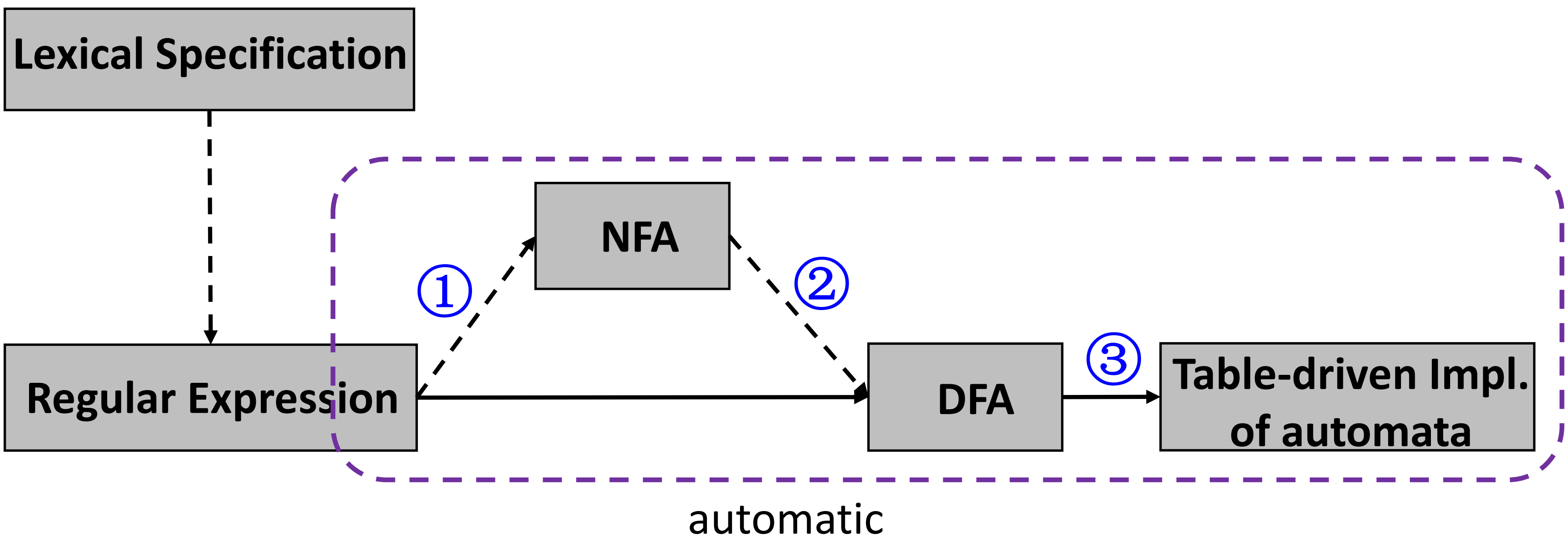
<ptn3, abb>

How to Match Keywords?[匹配关键字]

- Example: to recognize the following tokens
  - Identifiers: letter(letter|digit)*
  - Keywords: if, then, else
- **Approach 1**: make REs for keywords and place them before REs for identifiers so that they will take precedence
  - Will result in more bloated finite state machine
- **Approach 2**: recognize keywords and identifiers using same RE but differentiate using special keyword table
  - Will result in more streamlined finite state machine
  - But extra table lookup is required
- Usually approach 2 is more efficient than 1, but you can implement approach 1 in your projects for simplicity

The Conversion Flow[转换流程]

- Outline: RE → NFA → DFA → Table-driven Implementation
  - ③ Converting DFAs to table-driven implementations
  - ① Converting REs to NFAs (M-Y-T algorithm)
  - ② Converting NFAs to DFAs (subset construction)
  - ③' DFA minimization (partition algorithm)

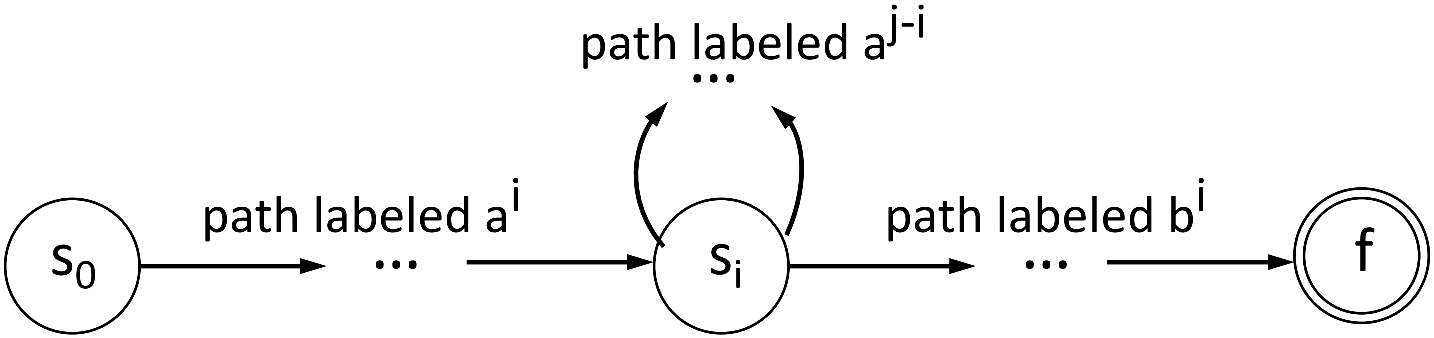


Beyond Regular Languages

- Regular languages are expressive enough for tokens
  - Can express identifiers, strings, comments, etc.
- However, it is the weakest (least expressive) language
  - Many languages are not regular
  - C programming language is not
    - The language matching braces "{{{...}}}" is also not
  - FA cannot count # of times char encountered
    - $L = \{a^n b^n \mid n \geq 1\}$
    - Crucial for analyzing languages with nested structures (e.g. nested for loop in C language)
- We need a more powerful language for parsing
  - Later, we will discuss context-free languages (CFGs)

RE/FA is NOT Powerful Enough

- $L = \{a^n b^n \mid n \geq 1\}$ is NOT a Regular Language
  - Suppose ***L*** were the language defined by regular expression
  - Then we could construct a DFA ***D*** with ***k*** states to accept ***L***
  - Since ***D*** has only ***k*** states, for an input beginning with more than ***k a***'s, ***D*** must enter some state twice, say s_i
  - Suppose that the path from s_i back to itself is labeled with a^{j-i}
  - Since $a^i b^i$ is in ***L***, there must be a path labeled b^i from s_i to an accepting state ***f***
  - But, there is also a path from s_0 through s_i to ***f*** labelled $a^j b^i$
  - Thus, ***D*** also accepts $a^j b^i$, which is not in ***L***, contradicting the assumption that ***L*** is the language accepted by ***D***



RE/FA is NOT Powerful Enough(cont.)

- *L* = $\{a^n b^n \mid n \geq 1\}$ is not a Regular Language
  - Proof → Pumping Lemma (泵引理)
  - FA does not have any memory (FA cannot count)
    - The above *L* requires to keep count of a's before seeing b's

- Matching parenthesis is not a RL
- Any language with nested structure is not a RL
  - if … if … else … else
- Regular Languages
  - Weakest formal languages that are widely used

中山大學 SUN YAT-SEN UNIVERSITY
国家超级计算广州中心 NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

Compilation Principle
编 译 原 理

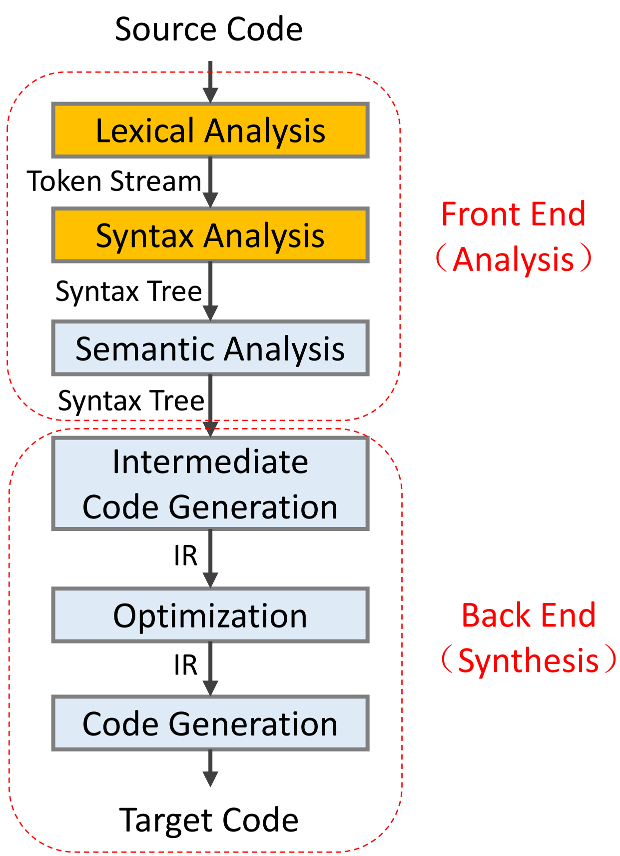
第4讲：语法分析(1)

张献伟

xianweiz.github.io

DCS290, 3/7/2023

Compilation Phases[编译阶段]



Example

- $vim test.c

```
void main() {
 int;
 int a,;
 int b, c;
}
```

- $clang -cc1 -dump-tokens ./test.c
- $clang -o test test.c

```
void 'void'
identifier 'main'
l_paren '('
r_paren ')'
l_brace '{'
int 'int'
semi ';'
int 'int'
identifier 'a'
comma ','
semi ';'
int 'int'
identifier 'b'
comma ','
identifier 'c'
semi ';'
r_brace '}'
eof ''
```

```
test.c:1:1: warning: return type of 'main' is not 'int' [-Wmain-return
void main() {
^
test.c:1:1: note: change return type to 'int'
void main() {
^~~~
int
test.c:2:3: warning: declaration does not declare anything [-Wmissing-
ns]
  int;
  ^~~
test.c:3:9: error: expected identifier or '('
  int a,;
        ^
2 warnings and 1 error generated.
```

Syntax Analysis[语法分析]

- Second phase of compilation[第二阶段]
  - Also called as **parser**
- Parser obtains a string of tokens from the lexical analyzer[以token作为输入]
  - **Lexical analyzer** reads the chars of the source program, groups them into lexically meaningful units called **lexemes**
  - and produces as output **tokens** representing these lexemes
    - Token: <token name, attribute value>
  - Token names are used by parser for syntax analysis
    - tokens → parse tree/AST
- Parse tree[分析树]
  - Graphically represent the syntax structure of the token stream

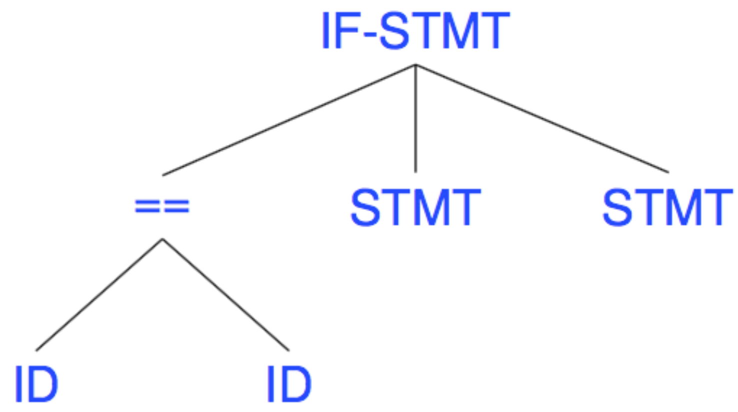
Parsing Example

- Input: if(x==y) … else …[源程序输入]

- Parser input (Lexical output)[语法分析输入]

  KEY(IF) '(' ID(x) OP('==') ID(y) ')' … KEY(ELSE) …

- Parser output[语法分析输出]



Parsing Example (cont.)

- Example: <id, x> <op, *> <op, %>
  - Is it a valid token stream in C language? **YES**
  - Is it a valid statement in C language (x *%)? **NO**

- Not every sequence of tokens are valid
  - Parser must distinguish between valid and invalid token sequence

- We need a method to describe what is valid sequence?
  - To specify the syntax of a programming language

How to Specify Syntax?

- How can we specify a syntax with nested structures?
  - Is it possible to use RE/FA?
  - L(Regular Expression) ≡ L(Finite Automata)

- RE/FA is not powerful enough
  - $L = \{a^n b^n \mid n \geq 1\}$ is not a Regular Language

- Example: matching parenthesis: # of '(' == # of ')'
  - (x+y)*z ✓
  - ((x+y)+y)*z ✓
  - (...(((x+y)+y)+y)...) ✓
  - ((x+y)+y)+y)*z ✗

What Language Do We Need?

- C-language syntax: **Context Free Language** (CFL)[上下文无关语言] e.g., ‘else’ is always ‘else’, wherever you place it
  - A broader category of languages that includes languages with nested structures
- Before discussing CFL, we need to learn a more general way of specifying languages than RE, called **Grammars**[文法]
  - Can specify both RL and CFL
  - and more ...
- Everything that can be described by a regular expression can also be described by a grammar
  - Grammars are most useful for describing nested structures

Concepts

- **Language**[语言]
  - Set of strings over alphabet
    - *String*: finite sequence of symbols
    - *Alphabet*: finite set of symbols
- **Grammar**[文法]
  - To systematically describe the syntax of programming language constructs like expressions and statements
- **Syntax**[语法]
  - Describes the proper form of the programs
  - Specified by grammar

Grammar[文法]

- Formal definition[形式化定义]: 4 components **{T, N, s, δ}**
- ***T***: set of terminal symbols[终结符]
  - Basic symbols from which strings are formed
  - Essentially tokens - leaves in the parse tree
- ***N***: set of non-terminal symbols[非终结符]
  - Each represents a set of strings of terminals – internal nodes
  - E.g.: declaration, statement, loop, ...
- **s**: start symbol[开始符号]
  - One of the non-terminals
- ***σ***: set of productions[产生式]
  - Specify the manner in which the terminals and non-terminals can be combined to to form strings
  - "LHS → RHS": left-hand-side produces right-hand-side

中山大學 SUN YAT-SEN UNIVERSITY

NSCC GZ

Grammar (cont.)

- Usually, we can just write the σ[简写]

- Merge rules sharing the same LHS[规则合并]
  - $\alpha \rightarrow \beta_1, \alpha \rightarrow \beta_2, \ldots, \alpha \rightarrow \beta_n$
  - $\alpha \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

G = ({id, +, *, (,)} , {E}, E, P)
P = { E → E + E,
E → E * E,
E → (E),
E → id }

→

G: E → E + E,
E → E * E,
E → (E),
E → id }

→

E → E + E | E * E | (E) | id

Syntax Analysis[语法分析]

- Informal description of variable declarations in C[变量声明]
  - Starts with *int* or *float* as the first token[类型]
  - Followed by one or more *identifier* tokens, separated by token *comma*[逗号分隔的标识符]
  - Followed by token *semicolon*[分号]
- To *check whether a program is well-formed* requires a specification of *what is a well-formed program*[语法定义]
  - The specification be precise[正确]
  - The specification be complete[完备]
    - Must cover all the syntactic details of the language
  - The specification must be convenient[便捷] to use by both language designer and the implementer
- A **context free grammar** meets these requirements