Compilation Principle
编 译 原 理

第2讲：词法分析(2)

张献伟

xianweiz.github.io

DCS290, 3/5/2024

Review Questions

- Q1: input and output of lexical analysis?

  Input: source code/char stream, output: tokens

- Q2: lexical analysis of “while i>=1)”?

  (keyword, ‘while’), (id, ‘i’), (sym, ‘>=’), (num, ‘1’), (sym, ‘)’)

- Q3: $\Sigma = \{a, b\}$, $L_1 = \{aa\}$, $L_2=\{bbb\}$. What are $L_1 \mid L_2$ and L_1L_2?

  $L_3 = L_1 \mid L_2 = \{aa\} \mid \{bbb\} = \{aa, bbb\}$, $L_4 = L_1L_2 = \{aabbb\}$

- Q4: L_3^2?

  $L_3^2 = L_3L_3 = \{aa, bbb\}\{aa, bbb\} = \{aaaa, aabbb, bbbaa, bbbbbb\}$

- Q5: describe the meaning of $L_1^* \mid L_2^*$?

  A language composed of ‘*a*’s and ‘b’s of length 2N and 3N, respectively, including ε

- Q6: RE of identifiers in C language?

  (_letter)(_letter|digit)*

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

<table>
<tr><th>Regular Expression</th><th>Explanation</th></tr>
<tr><td>a^*</td><td>0 or more a’s (ε, a, aa, aaa, aaaa, …)</td></tr>
<tr><td>a+</td><td>1 or more a’s (a, aa, aaa, aaaa, …)</td></tr>
<tr><td>(a|b)(a|b)</td><td>(aa, ab, ba, bb)</td></tr>
<tr><td>$(a|b)^*$</td><td>all strings of a’s and b’s (including ε)</td></tr>
<tr><td>$(aa|ab|ba|bb)^*$</td><td>all strings of a’s and b’s of even length</td></tr>
<tr><td>[a-zA-Z]</td><td>shorthand for “a|b|…z|A|B|…|Z”</td></tr>
<tr><td>[0-9]</td><td>shorthand for “0|1|2|…|9”</td></tr>
<tr><td>$0([0-9])^*0$</td><td>numbers that start and end with 0</td></tr>
<tr><td>$1^*(0|\varepsilon)1^*$</td><td>binary strings that contain at most one zero</td></tr>
<tr><td>$(0|1)^*00(0|1)^*$</td><td>all binary strings that contain ‘00’ as substring</td></tr>
</table>

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

Different REs of the Same Language

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

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

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

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 ‘=’)
  - Maximal match: 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
  - Rule of thumb: 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]
  - 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]

Impl. of Lexical Analyzer[实现]

- How do we go from specification to implementation?
  - RE → finite automata (FA)
- **Solution 1**: to implement using a tool — Lex (for C), Flex (for C++), Jlex (for java)
  - Programmer specifies tokens using REs
  - The tool generates the source code from the given REs
    - The Lex tool essentially does the following translation: REs (specification) ⇒ FAs (implementation)
- **Solution 2**: to write the code yourself
  - More freedom; even tokens not expressible through REs
  - But difficult to verify; not self-documenting; not portable; usually not efficient
  - ~~Generally not encouraged~~

Transition Diagram[转换图]

- REs → transition diagrams
  - By hand
  - Automatic

- Node[节点]: state
  - Each state represents a condition that may occur in the process
  - Initial state (<u>Start</u>): only one, circle marked with ‘start →’
  - Final state (<u>Accepting</u>): may have multiple, double circle

- Edge[边]: directed, labeled with symbol(s)
  - From one state to another on the input

Finite Automata[有穷自动机]

- **Regular Expression** = specification[正则表达是定义]
- **Finite Automata** = implementation[自动机是实现]

- Automaton (pl. automata): a machine or program
- **Finite automaton** (FA): a program with a finite number of states

- Finite Automata are similar to transition diagrams
  - They have states and labelled edges
  - There are one unique start state and one or more than one final states

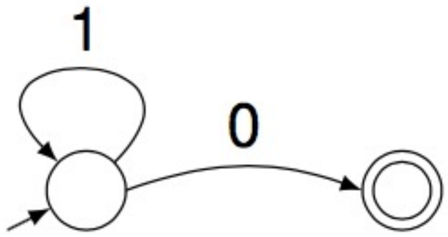
FA: Language

- An FA is a program for classifying strings (accept, reject)
  - In other words, a program for recognizing a language
  - The Lex tool essentially does the following translation: REs (specification) ⇒ FAs (implementation)
  - For a given string ‘x’, if there is transition sequence for ‘x’ to move from start state to certain accepting state, then we say ‘x’ is accepted by the FA
    - Otherwise, rejected

- Language of FA = set of strings accepted by that FA
  - L(FA) ≡ L(RE)

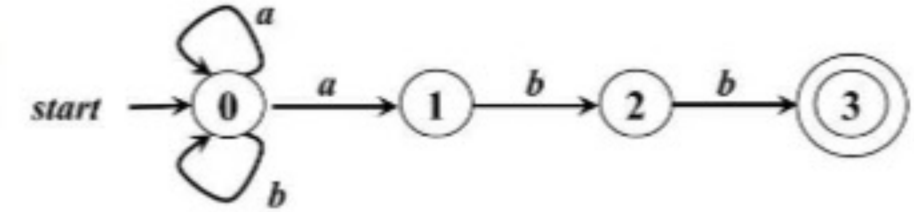
Example

- Are the following strings acceptable?
  - 0 ✓
  - 1 ✗
  - 11110 ✓
  - 11101 ✗
  - 11100 ✗
  - 1111110 ✓



- What language does the state graph recognize? ∑ = {0, 1}

Any number of ‘1’s followed by a single 0



L(FA): all strings of ∑{a, b}, ending with ‘abb’

L(RE) = (a|b)*abb

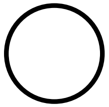
DFA and NFA

- Deterministic Finite Automata (**DFA**): the machine can exist in only one state at any given time[确定]
  - One transition per input per state
  - No ε-moves
  - Takes only one path through the state graph

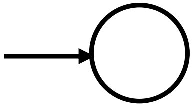
- Nondeterministic Finite Automata (**NFA**): the machine can exist in multiple states at the same time[非确定]
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves
  - Can choose which path to take
    - An NFA accepts if some of these paths lead to accepting state at the end of input

State Graph

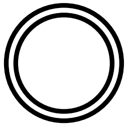
- 5 components （Σ, S, n, F, δ）
  - An input alphabet Σ
  - A set of states S



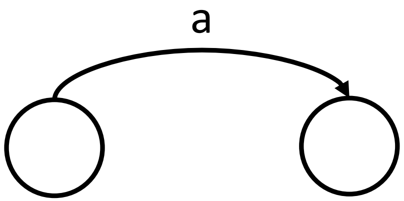
  - A start state $n \in S$



  - A set of accepting states $F \subseteq S$

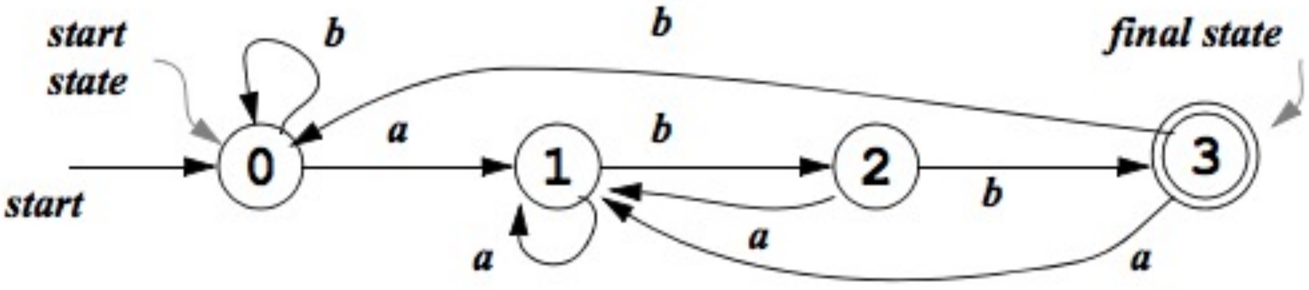


  - A set of transitions $\delta: S_a \xrightarrow{\text{input}} S_b$



Example: DFA

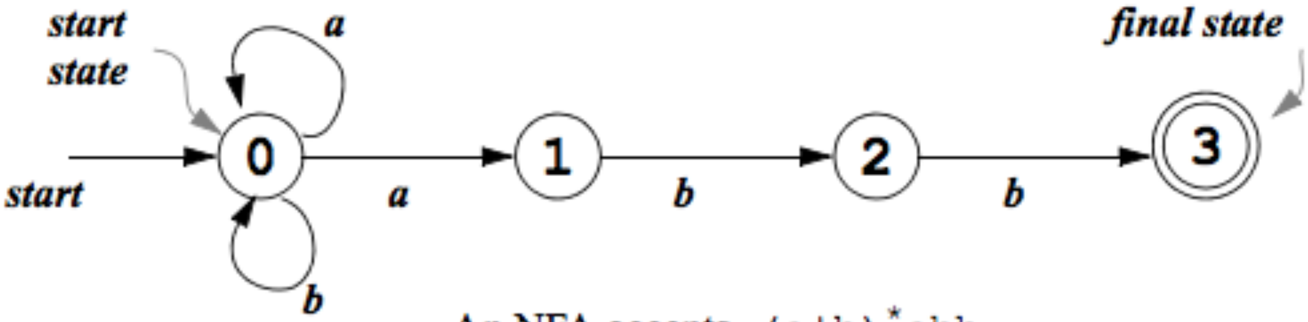
- There is **only one** possible sequence of moves --- either lead to a final state and accept or the input string is rejected
  - Input string: aabb
  - Successful sequence: $0 \xrightarrow{a} 1 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$



A DFA accepts (a|b)*abb

Example: NFA

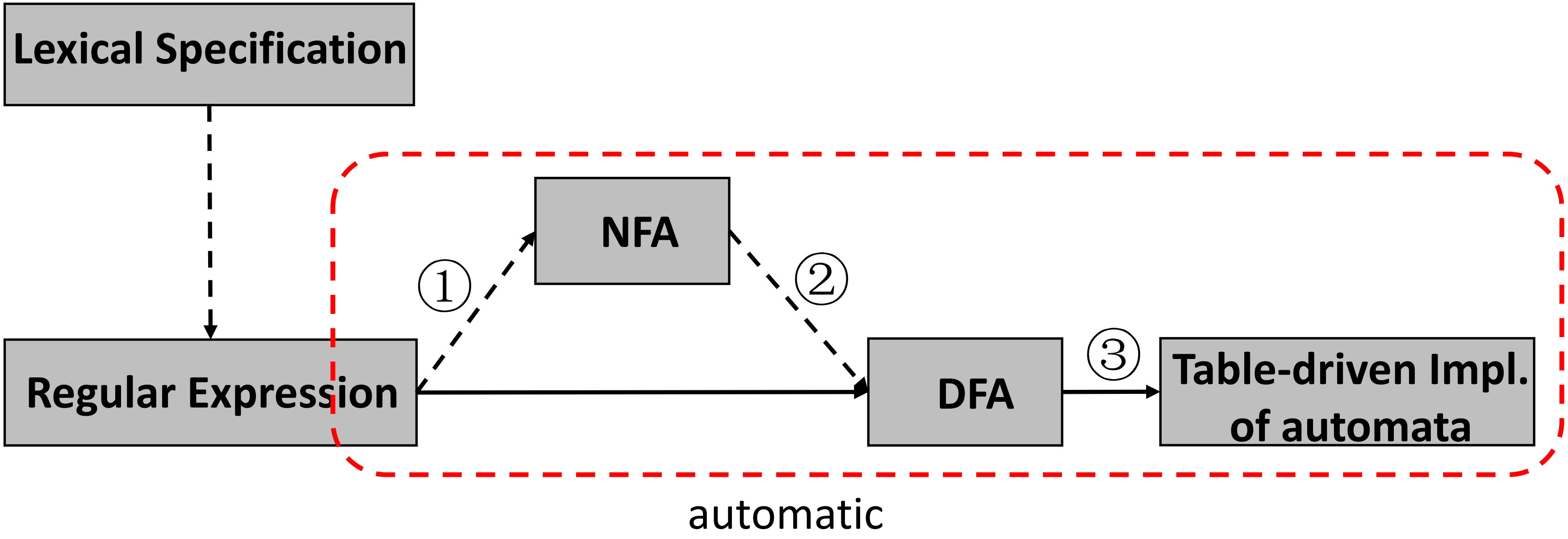
- There are **many possible** moves: to accept a string, we only need one sequence of moves that lead to a final state
  - Input string: aabb
  - Successful sequence: $0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$
  - Unsuccessful sequence: $0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$



An NFA accepts $(a|b)^*abb$

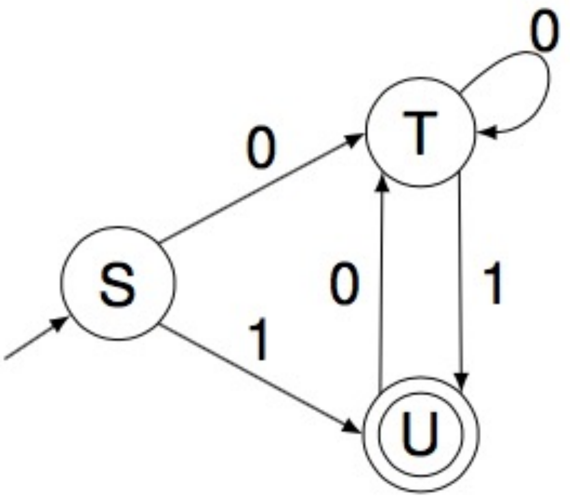
Conversion Flow[转换流程]

- Outline: RE → NFA → DFA → Table-drive Implementation
  - ③ Converting DFAs to table-driven implementations
  - ① Converting REs to NFAs
  - ② Converting NFAs to DFAs



DFA → Table

- FA can also be represented using transition table



alphabet →

state ↓

	0	1
S	**T**	**U**
T	**T**	**U**
U	**T**	**x**

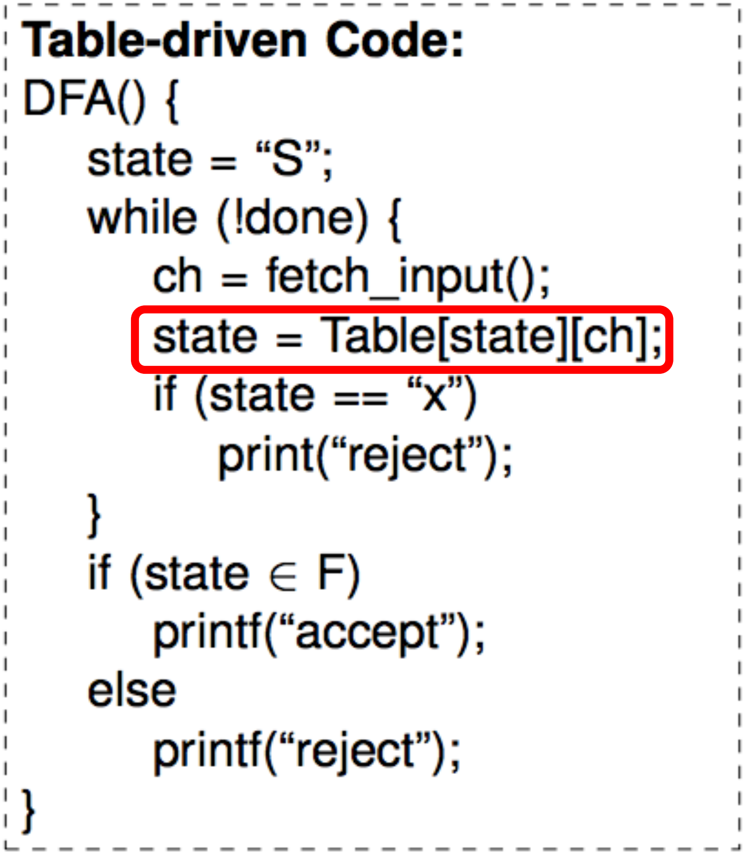


Table-driven Code:

```
DFA() {
    state = “S”;
    while (!done) {
        ch = fetch_input();
        state = Table[state][ch];
        if (state == “x”)
            print(“reject”);
    }
    if (state ∈ F)
        printf(“accept”);
    else
        printf(“reject”);
}
```

Q: which is/are accepted?
111
000
001

More on Table

- Implementation is efficient[表格是一种高效实现]
  - Table can be automatically generated
  - Need finite memory O(S x ∑)
    - Size of transition table
  - Need finite time O(input length)
    - Number of state transitions

- Pros and cons of table[表格实现的优劣]
  - Pro: can easily find the transitions on a given state and input
  - Con: takes a lot of space, when the input alphabet is large, yet most states do not have any moves on most of the input symbols