Implementation of Lexical Analysis

--Finite Automata

Outline

- Specifying lexical structure using regular expressions
- Finite automata
  - Deterministic Finite Automata (DFAs)
  - Non-deterministic Finite Automata (NFAs)
- Implementation of regular expressions

  RegExp => NFA => DFA => Tables

Regular Expressions => Lexical Spec. (1)

1. Select a set of tokens
   - Number, Keyword, Identifier, ...
2. Write a R.E. for the lexemes of each token
   - Number = $digit^+$
   - Keyword = 'if' | 'else' | ...
   - Identifier = letter (letter | digit)*
   - LeftPar = '('
   - ...

Regular Expressions => Lexical Spec. (2)

3. Construct R, matching all lexemes for all tokens

   R = Keyword | Identifier | Number | ...
   $= R_1 \quad | R_2 \quad | R_3 \quad | \ldots$

Facts: If $s \in L(R)$ then s is a lexeme

- Furthermore $s \in L(R)$ for some "R_i"
- This "R_i" determines the token that is reported

Regular Expressions => Lexical Spec. (3)

4. Let the input be $x_1 \ldots x_n$

   ($x_1 \ldots x_n$ are characters in the language alphabet)

   For $1 \le i \le n$ check $x_1 \ldots x_i \in L(R)$?

5. It must be that $x_1 \ldots x_i \in L(R_j)$ for some i and j

6. Remove $x_1 \ldots x_i$ from input and go to (4)

Lexing Example

R = Whitespace | Integer | Identifier | '+'

- Parse "f +3 +g"
  - "f" matches R, more precisely Identifier
  - "+" matches R, more precisely '+'
  - ...
  - The token-lexeme pairs are
    (Identifier, "f"), ('+', "+"), (Integer, "3")
    (Whitespace, " "), ('+', "+"), (Identifier, "g")
- We would like to drop the Whitespace tokens
  - after matching Whitespace, continue matching

Ambiguities (1)

- There are ambiguities in the algorithm
- Example:
  R = Whitespace | Integer | Identifier | '+'
- Parse "foo+3"
  - "f" matches R, more precisely Identifier
  - But also "fo" matches R, and "foo", but not "foo+"
- How much input is used? What if
  - $x_1 \ldots x_i \in L(R)$ and also $x_1 \ldots x_K \in L(R)$
  - "Maximal munch" rule: Pick the longest possible substring that matches R

7

More Ambiguities

R = Whitespace | 'new' | Integer | Identifier

- Parse "new foo"
  - "new" matches R, more precisely 'new'
  - but also Identifier, which one do we pick?
- In general, if $x_1 \ldots x_i \in L(R_j)$ and $x_1 \ldots x_i \in L(R_k)$
  - Rule: use rule listed first (j if j < k)
- We must list 'new' before Identifier

8

Error Handling

R = Whitespace | Integer | Identifier | '+'

- Parse "=56"
  - No prefix matches R: not "=", nor "=5", nor "=56"
- Problem: Can't just get stuck …
- Solution:
  - Add a rule matching all "bad" strings; and put it last
- Lexer tools allow the writing of:
  $R = R_1 \mid \ldots \mid R_n \mid$ Error
  - Token Error matches if nothing else matches

9

Summary

- Regular expressions provide a concise notation for string patterns
- Use in lexical analysis requires small extensions
  - To resolve ambiguities
  - To handle errors
- Good algorithms known (next)
  - Require only single pass over the input
  - Few operations per character (table lookup)

10

Finite Automata

- Regular expressions = specification
- Finite automata = implementation
- A finite automaton consists of
  - An input alphabet Σ
  - A set of states S
  - A start state n
  - A set of accepting states $F \subseteq S$
  - A set of transitions $state \to^{input} state$

11

Finite Automata

- Transition

  $$s_1 \to^{a} s_2$$

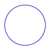
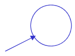
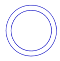
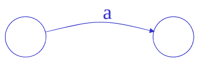
- Is read

  In state s_1 on input "a" go to state s_2

- If end of input (or no transition possible)
  - If in accepting state => accept
  - Otherwise => reject

12

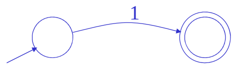
Finite Automata State Graphs

- A state 
- The start state 
- An accepting state 
- A transition 

13

A Simple Example

- A finite automaton that accepts only “1”

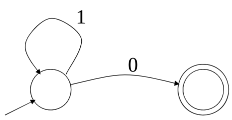


- A finite automaton accepts a string if we can follow transitions labeled with the characters in the string from the start to some accepting state

14

Another Simple Example

- A finite automaton accepting any number of 1’s followed by a single 0
- Alphabet: {0,1}

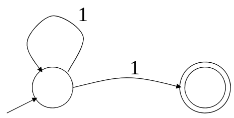


- Check that “1110” is accepted but “110…” is not

15

And Another Example

- Alphabet still { 0, 1 }

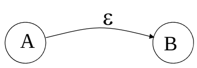


- The operation of the automaton is not completely defined by the input
  - On input “11” the automata could be in either state

16

Epsilon Moves

- Another kind of transition: ε-moves



- Machine can move from state A to state B without reading input

17

Deterministic and Nondeterministic Automata

- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No ε-moves
- Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves
- Finite automata have finite memory
  - Need only to encode the current state

18

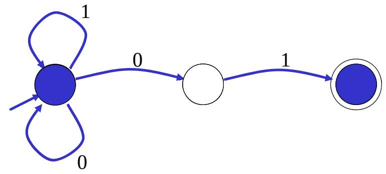
Execution of Finite Automata

- A DFA can take only one path through the state graph
  - Completely determined by input
- NFAs can choose
  - Whether to make ε-moves
  - Which of multiple transitions for a single input to take

19

Acceptance of NFAs

- An NFA can get into multiple states



- Input: 1 0 1
- Rule: NFA accepts if it <u>can</u> get in a final state

20

NFA vs. DFA (1)

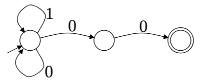
- NFAs and DFAs recognize the same set of languages (regular languages)
- DFAs are easier to implement but harder to be constructed
  - There are no choices to consider

21

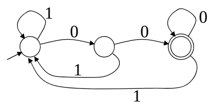
NFA vs. DFA (2)

- For a given language the NFA can be simpler than the DFA

NFA



DFA

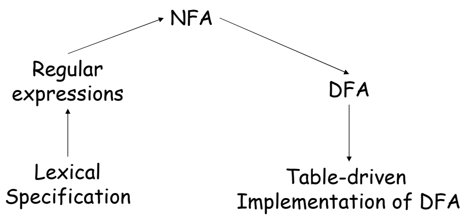


- DFA can be exponentially larger than NFA

22

Regular Expressions to Finite Automata

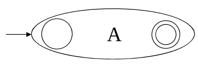
- High-level sketch



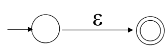
23

Regular Expressions to NFA (1)

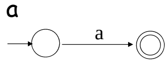
- For each kind of rexp, define an NFA
  - Notation: NFA for rexp A



- For ε



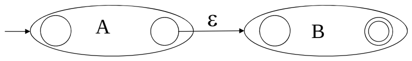
- For input a



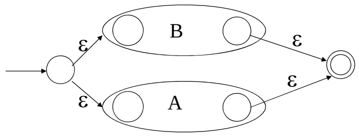
24

Regular Expressions to NFA (2)

- For AB

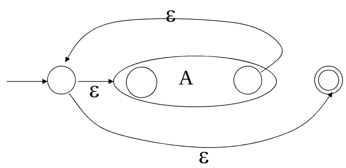


- For A | B



Regular Expressions to NFA (3)

- For A*

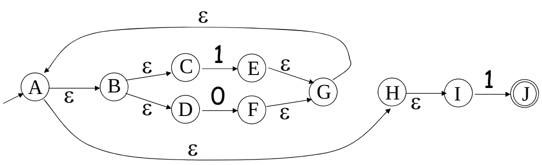


Example of RegExp -> NFA conversion

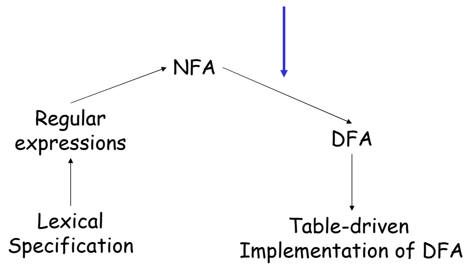
- Consider the regular expression

  (1 | 0)*1

- The NFA is



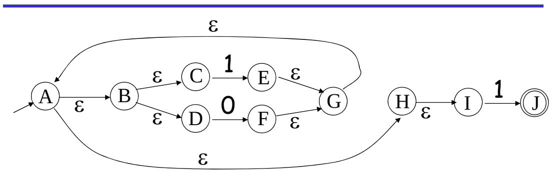
Next

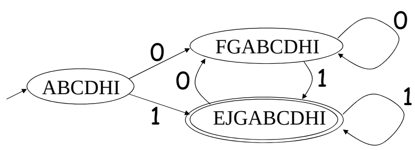


NFA to DFA. The Trick

- Simulate the NFA
- Each state of DFA
  = a non-empty subset of states of the NFA
- Start state
  = the set of NFA states reachable through ε-moves from NFA start state
- Add a transition $S \rightarrow^a S'$ to DFA iff
  - S' is the set of NFA states reachable from the states in S after seeing the input a
    - considering ε-moves as well

NFA -> DFA Example





NFA to DFA. Remark

- An NFA may be in many states at any time
- How many different states ?
- If there are N states, the NFA must be in some subset of those N states
- How many non-empty subsets are there?
  - 2^N - 1 = finitely many

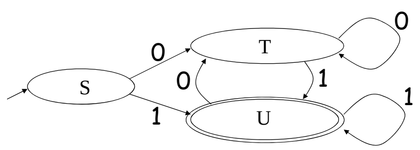
31

Implementation

- A DFA can be implemented by a 2D table T
  - One dimension is "states"
  - Other dimension is "input symbols"
  - For every transition $S_i \to^a S_k$ define T[i,a] = k
- DFA "execution"
  - If in state S_i and input a, read T[i,a] = k and skip to state S_k
  - Very efficient

32

Table Implementation of a DFA



	0	1
S	T	U
T	T	U
U	T	U

33

Implementation (Cont.)

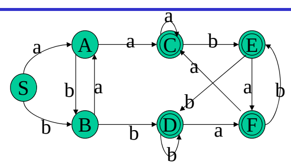
- NFA -> DFA conversion is at the heart of tools such as flex or jlex
- But, DFAs can be huge
- In practice, flex-like tools trade off speed for space in the choice of NFA and DFA representations

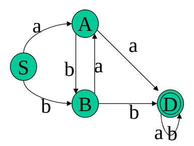
34

Minimize DFA state number

Π0: {S, A, B}
{C, D, E, F}

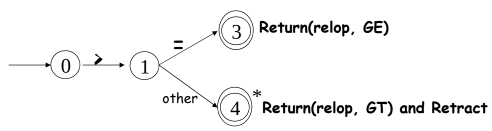
a {S, A, B}->{S, B} {A}
Π1: {S, B} {A} {C, D, E, F}
{
b {S, B} ->{S} {B}
Π2: {S} {A} {B} {C, D, E, F}

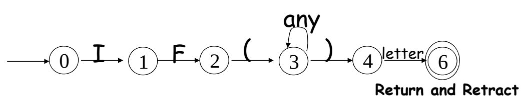




35

Use FA to build scanner

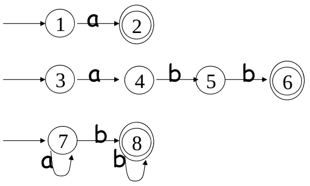




36

Example:

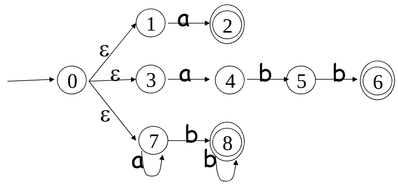
a | abb | a*b⁺



37

1. Max length match
2. Muti—match select

Input: aaba; abb

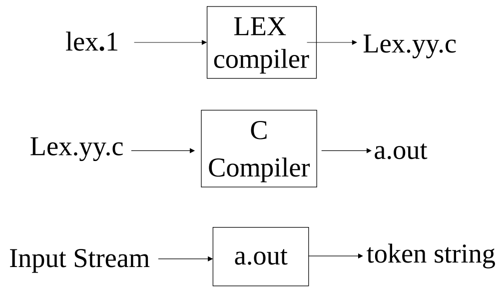


38

what does lex do?

- Input: patterns written by programmer, describing tokens in language
- Process:
  - Reads patterns as regular expressions
  - Builds finite automaton to accept valid tokens
- Output: C code implementation of FA
- Compile and link C code, you've got a scanner

39



40

scanner comparison

- Hand-coded scanner (like any ordinary program):
  - Programmer creates types, defines data & procedures, designs flow of control, implements in source language.
- Lex-generated scanner:
  - Programmer writes patterns
  - (Declarative, not procedural)
  - Lex implements flow of control
  - Must less hand-coding, but
  - code looks pretty alien, tricky to debugs

41

Summary: Automata theory⇒ Programming practice

- Regular expressions and automata theory prove you can write regular expressions, give them to a program like lex, which will generate a machine to accept exactly those expressions.

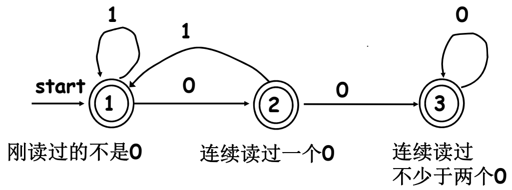
42

例 题 1

- 叙述下面的正规式描述的语言，并画出接受该语言的最简DFA的状态转换图

  (1|01)* 0*

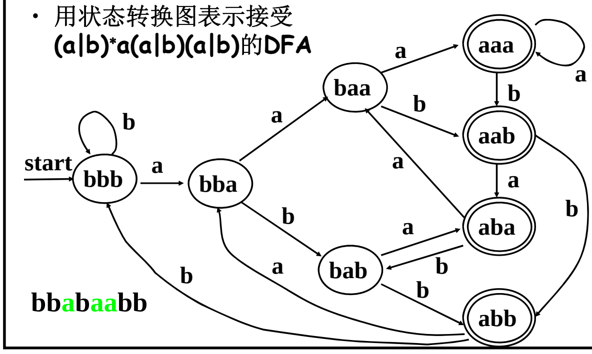
  - 描述的语言是，所有不含子串001的0和1的串



例 题 2

- 用状态转换图表示接受

  (a|b)*a(a|b)(a|b)的DFA



bbabaabb

例 题 3

- 写出语言“所有相邻数字都不相同的非空数字串”的正规定义

  12303135710679803579012 3

$answer \to (0 \mid no\_0\ \ 0)\ (no\_0\ \ 0)^*\ (no\_0 \mid \varepsilon) \mid no\_0$

$no\_0 \to (1 \mid no\_0\text{-}1\ 1)\ (no\_0\text{-}1\ 1)^*\ (no\_0\text{-}1 \mid \varepsilon) \mid no\_0\text{-}1$

. . .

$no\_0\text{-}8 \to 9$

将这些正规定义逆序排列就是答案

例 题 4

下面C语言编译器编译下面的函数时，报告

parse error before ‘else’

```
long gcd(p,q)
long p,q;
{
  if (p%q == 0)
      /* then part */
      return q            此处遗漏分号
  else
      /* else part */
      return gcd(q, p%q);
}
```

例 题 4

现在少了第一个注释的结束符号后，反而不报错了

```
long gcd(p,q)
long p,q;
{
  if (p%q == 0)
      /* then part
      return q
  else
      /* else part */
      return gcd(q, p%q);
}
```