CS 473: COMPILER DESIGN

Adapted from slides by Steve Zdancewic, UPenn
PRINCIPLES OF LEXING
Regular Expressions: Definition

• Regular expressions precisely describe sets of strings.

• A regular expression $R$ has one of the following forms:
  
  – $\varepsilon$  
  Epsilon stands for the empty string
  
  – 'a'  
  An ordinary character stands for itself
  
  – $R_1 \mid R_2$  
  Alternatives, stands for choice of $R_1$ or $R_2$
  
  – $R_1R_2$  
  Concatenation, stands for $R_1$ followed by $R_2$
  
  – $R^*$  
  Kleene star, stands for zero or more repetitions of $R$

• Useful extensions:
  
  – "foo"  
  Strings, equivalent to 'f' 'o' 'o'
  
  – $R^+$  
  One or more repetitions of $R$, equivalent to $RR^*$
  
  – $R?$  
  Zero or one occurrences of $R$, equivalent to ($\varepsilon \mid R$)
  
  – ['a'-'z']  
  One of a or b or c or ... z, equivalent to (a | b | ... | z)
  
  – [^'0'-'9']  
  Any character except 0 through 9
  
  – .  
  Any character
Example Regular Expressions

- Recognize the keyword "if": "if"
- Recognize a digit: ['0'–'9']
- Recognize an integer literal: '-'?['0'–'9']+ 
- Recognize an identifier: (['a'–'z']|['A'–'Z']) (['0'–'9']| _ | ['a'–'z']| ['A'–'Z'])*
Finite Automata

• Every regular expression can be recognized by a finite automaton
• Consider the regular expression: " " [ ^ ' " ' ] * ' " '
• An automaton (DFA) can be represented as:
  – A transition table:
    |     | "   | Non-" |
    |-----|-----|-------|
    | 0   | 1   | ERROR |
    | 1   | 2   | 1     |
    | 2   | ERROR | ERROR |
  – A graph:
• Every regular expression can be recognized by a finite automaton

• Strategy: consider every possible regular expression:

'a'

ε

R_1R_2

What about?

R_1 | R_2
Nondeterministic Finite Automata

- A finite set of states, a start state, and accepting state(s)
- Transition arrows connecting states
  - Labeled by input symbols
  - Or $\varepsilon$ (which does not consume input)
- **Nondeterministic**: two arrows leaving the same state may have the same label
• Converting regular expressions to NFAs is easy.
• Assume each NFA has one start state, unique accept state
• Sums and Kleene star are easy with NFAs
Exercise: RE to NFA

- Construct an NFA for the following regular expression:

  $$(a^*b^*) \mid (b^*a^*)$$
Deterministic Finite Automata

• An NFA accepts a string if there is any way to get to an accepting state
  – To implement, we either have to try all possibilities or get good at guessing!

• A deterministic finite automata never has to guess: two arrows leaving the same state must have different labels, and never $\varepsilon$

• This means that action for each input is fully determined!

• We can make a table for each state: “if you see symbol X, go to state Y”

• Fortunately, we can convert any NFA into a DFA!
NFA to DFA conversion (Intuition)

- Idea: Run all possible executions of the NFA “in parallel”
- Keep track of a set of possible states: “finite fingers”
- Consider: –? [0–9] +

- NFA representation:

- DFA representation:
Summary of Lexer Generator Behavior

• Take each regular expression $R_i$ and its action $A_i$

• Compute the NFA formed by $(R_1 \mid R_2 \mid \ldots \mid R_n)$
  – Remember the actions associated with the accepting states of the $R_i$

• Compute the DFA for this big NFA
  – There may be multiple accept states
  – A single accept state may correspond to one or more actions

• Compute the minimal equivalent DFA
  – There is a standard algorithm due to Myhill & Nerode

• Produce the transition table

• Implement longest match:
  – Start from initial state
  – Follow transitions, remember last accept state entered (if any)
  – Accept input until no transition is possible (i.e. next state is “ERROR”)
  – Perform the highest-priority action associated with the last accept state; if no accept state there is a lexing error
Questions

Top
Lex: Start States

• Sometimes we want to use different lexers for different parts of a program
• For instance, strings:
  
  ```c
  if (a == "\"if\" 0") return 0;
  ```

• **Start states** let us specify multiple sets of lexing rules and switch between them

  ```
  %s STRING // define a new ruleset for strings

  // INITIAL is the default lexer
  <INITIAL>[a-z]+ { return ID; }
  <INITIAL>[0-9]+ { return NUM; }

  // switch to the string lexer
  <INITIAL>" { BEGIN STRING; }
  <STRING>. { /*store characters*/; }

  // switch back when we're done
  <STRING>" { BEGIN INITIAL; }
  ```

• Demo: states.lex
Questions

Top