CS 301: Regular Expressions

Definitions

Equivalence to Finite Automata

RE's: Introduction

- *Regular expressions*
  - An algebraic way to describe languages

- Describes exactly the regular languages

Regular Expressions

- Convenient and most popular way to describe tokens of a programming language
- A regular expression is built up of simpler regular expressions (using defining rules)
- Each regular expression denotes a language
- **regular set :** Language denoted by a regular expression

RE's: Definition

- Basis 1: If *a* is any symbol, then **a** is a RE, and L(**a**) = {a}
  - Note: {a} is the language containing one string, and that string is of length 1
- Basis 2: ϵ is a RE, and L(ϵ) = {ϵ}
- Basis 3: $\emptyset$ is a RE, and L($\emptyset$) = $\emptyset$.

RE's: Definition – (2)

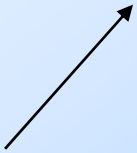
- Induction 1: If E_1 and E_2 are regular expressions, then E_1+E_2 is a regular expression, and $L(E_1+E_2) = L(E_1) \cup L(E_2)$

- Induction 2: If E_1 and E_2 are regular expressions, then E_1E_2 is a regular expression, and $L(E_1E_2) = L(E_1)L(E_2)$

Concatenation : the set of strings wx such that w is in $L(E_1)$ and x is in $L(E_2)$

RE's: Definition – (3)

- Induction 3: If E is a RE, then E* is a RE, and L(E*) = (L(E))*



Closure, or "Kleene closure" = set of strings $w_1w_2...w_n$, for some $n \geq 0$, where each w_i is in L(E)
Note: when n=0, the string is ϵ

Regular Expressions (Rules)

Regular expressions over alphabet Σ

Reg. Expr	Language it denotes
ε	$\{\varepsilon\}$
$a \in \Sigma$	$\{a\}$
$(r_1) \mid (r_2)$	$L(r_1) \cup L(r_2)$
$(r_1)\ (r_2)$	$L(r_1)\ L(r_2)$
$(r)^*$	$(L(r))^*$
(r)	$L(r)$

- $(r)^+ = (r)(r)^*$
- $(r)? = (r) \mid \varepsilon$

Regular Expressions (cont.)

- We may remove parentheses by using precedence rules
  - * highest
  - concatenation next
  - | lowest
- $ab^*|c$ means $(a(b)^*)|(c)$
- Ex:
  - $\Sigma = \{0,1\}$
  - $0|1 \Rightarrow \{0,1\}$
  - $(0|1)(0|1) \Rightarrow \{00,01,10,11\}$
  - $0^* \Rightarrow \{\varepsilon,0,00,000,0000,....\}$
  - $(0|1)^*$ => all strings with 0 and 1, including the empty string

Regular Definitions

- Writing regular expression for some languages may be difficult
- Alternative: *regular definitions*
- Assign names to regular expressions and we can use these names as symbols to define other regular expressions

- A ***regular definition*** is a sequence of the definitions of the form:

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

.

$d_n \rightarrow r_n$

where d_i is a distinct name and r_i is a regular expression over symbols in

$$\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$$

basic symbols previously defined names

Regular Definitions (cont.)

- Ex: Identifiers in Pascal

  letter $\rightarrow$ A | B | ... | Z | a | b | ... | z

  digit $\rightarrow$ 0 | 1 | ... | 9

  id $\rightarrow$ letter (letter | digit) *

  - If we try to write the regular expression representing identifiers without using regular definitions, that regular expression will be complex.

    (A|...|Z|a|...|z) ((A|...|Z|a|...|z) | (0|...|9)) *

- Ex: Unsigned numbers in Pascal

  digit $\rightarrow$ 0 | 1 | ... | 9

  digits $\rightarrow$ digit $^+$

  opt-fraction $\rightarrow$ (. digits) ?

  opt-exponent $\rightarrow$ (E (+|-)? digits) ?

  unsigned-num $\rightarrow$ digits opt-fraction opt-exponent

Examples: RE's

- L(**01**) = {01}
- L(**01**+**0**) = {01, 0}
- L(**0**(**1**+**0**)) = {01, 00}
  - Note order of precedence of operators
- L(**0***) = {ϵ, 0, 00, 000,… }
- L((**0**+**10**)*(ϵ+**1**)) = all strings of 0's and 1's without two consecutive 1's

Equivalence of RE's and Automata

- We need to show that for every RE, there is an automaton that accepts the same language
  - Pick the most powerful automaton type: the ε-NFA
- And we need to show that for every automaton, there is a RE defining its language
  - Pick the most restrictive type: the DFA