CS 344 - Regular expressions

Announcements
- Ch 2 this week
- Essay due Friday
Regular Expressions

Defined as:
- A character
- The empty string, ε
- 2 regular expressions concatenated
- 2 regular expressions separated by an "or" (written |)
- A regular expression followed by * (Kleene star - 0 or more occurrences)
Regular Languages

The class of languages described by a regular expression.

Ex: \(0^*10^* = L\)

Language of all strings that contain exactly one 1

\(1 \in L\)

\(01 \in L\)

\(000100 \in L\)

\(11 \notin L\)

\(0 \notin L\)
Ex: Give the regular expression for \( \exists w \mid w \) begins with a 1 and ends with a 0 if

\[ 1(0|1)^*0 \]

Ex: Give the regular expression for \( \exists w \mid w \) starts with 0 and has an odd length if

\[ 0((0|1)(0|1))^* \]
Example: Numbers in Pascal

\[
\begin{align*}
\text{digit} & \rightarrow [0-9] \\
\text{digit} & \rightarrow 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9 \\
\text{unsigned\_int} & \rightarrow \text{digit}\ \text{digit}\ * \\
\text{unsigned\_number} & \rightarrow \text{unsigned\_int} (3) \\
\text{unsigned\_int} & \rightarrow (3) \text{unsigned\_int} (3) \\
\text{unsigned\_int} & \rightarrow \text{unsigned\_int} (3) \\
\end{align*}
\]
Deterministic Finite Automata (DFA)

Regular languages are precisely the things recognized by DFAs.

- A set of states
- Input alphabet
- A start state
- A set of accept states
- A transition function: given a state and an input, output a new state
Example:

The arrows give transition from state to state. The start state is $s_1$, and the accept state is $s_2$. The language $L$ accepted is $0^*10^*$.
Ex:

Start

$S_1 \to [0-9]$ $S_2$

$[0,9]$
Non-deterministic Finite Automata

Note: No ambiguity is allowed in DFA's. So given a state & input, can't be multiple options.

Also - no ε-transitions.

If we allow several choices to exist, this is called an NFA.
L (0|1)* 0
Ex:
unsigned_number → unsigned_int (?) → unsigned_int

[0-9] → \[0-9\] → \[0-9\]

3 → 3

S
Essentially, we can think of an NFA as modeling a parallel set of possibilities (or a tree of them).

Thm: Every NFA has an equivalent DFA.

Both recognize regular languages.
Limitations of Regular Expressions

Certain languages are not regular.

Ex: \( Ew = \{\text{w has an equal number of } 0\text{s and } 1\text{s}\} \)

Somehow, this needs a type of memory which regular expressions do not have.
Why do we need this?

Need to "nest" expressions.

Ex:  expr → id | number | - expr
     | (expr) | expr op expr
     | op → + | - | / | *

Regular expressions can't quite do this.
Context Free Languages - CFLs

Described in terms of productions (called Backus-Naur Form, or BNF)

- A set of terminals $T$
- A set of non-terminals $N$
- A start symbol $S$ (a non-terminal)
- A set of productions
Ex: $\exists 0^n1^n \mid n > 0$

Nonterminals = $S, 0, 1$

$S \rightarrow 0S1 \bigcup \text{ productions}$

$S \rightarrow \epsilon$
Ex: \( \exists w \) w has an equal number of 0s \& 1's

\[
S \rightarrow 0S1 * \\
S \rightarrow 1S0 \\
S \rightarrow \epsilon
\]

Show \( 001011 \) is in this lang:

\[
S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 001S011 \\
\text{use } \epsilon
\]
Expression grammars: simple calculator

expr → term | expr add-op term
term → factor | term mult-op factor
factor → id | number | -factor | (expr)
add-op → + | -
mult-op → * | /

 terminals

x + y * z + a
Example: Show how rules can generate $3 + 4 * 5$.

\[
\begin{align*}
\text{expr} & \Rightarrow \text{expr} \text{ addop} \text{ term} \\
& \Rightarrow \text{term} \\
& \Rightarrow \text{factor} \\
& \Rightarrow \text{num} \\
\Rightarrow \text{number} + \text{term} \\
\Rightarrow \text{number} + (\text{term} \ast \text{factor}) \\
& \Rightarrow \text{factor} \\
& \Rightarrow \text{num} \\
\Rightarrow \text{num} + (\text{num} \ast \text{factor})
\end{align*}
\]
Parse Tree

\[ 6 \times 3 + 4 \times 5 \]
Another example: \( \text{slope} \times x + \text{intercept} \)

\[
\begin{aligned}
\text{expr} &\rightarrow \text{expr} \mid \text{add-op} \mid \text{term} \\
\text{term} &\rightarrow \text{term} \mid \text{factor} \\
\text{factor} &\rightarrow \text{factor} \mid \text{id} \\
\text{id} &\rightarrow \text{Slope} \mid \text{Slope} \\
\text{id} &\rightarrow \text{Slope} \mid \text{Slope} \\
\end{aligned}
\]
Question: Can these be ambiguous?
Ex: 10 - 4 - 3

Yes
Scanning

Recall that the scanner is responsible for

- tokenizing source code
- removing comments
- saving text of identifiers, #s, strings
- saving source locations for error messages
Ex: Calculator Scanner

```
expr → term / expr add-op term
term → factor / term1 mult-op factor
factor → id / number / - factor (expr)
add-op → + / -
mult-op → * / 
```

```
Add

id → letter (letter | digit)*

comment → */ (non-* | * non-/)*/ */ (non-newline)* newline

assign → :=

How to scan/recognize?
Ad Hoc Approach: Go char by char!

if current ∈ \{ "(, ), +, -, *, /" \}
    return that symbol

if current = "::
    read next
    if it is =, announce "assign"
    else announce error

if current = "/"
    read next
    if it is "*" or "/
        read until "*" or "newline" (resp.)
    else return divide
Another Way:

We are essentially running a DFA!

Accept states are just places we have reached a token.
Getting tokens (with DFA)
- Run the machine over & over to get our tokens
Rule: Always take longest possible token
Why? Ex: 3.14159
Ex: foo bar