Regular Expressions

• Define a formal language
  • The language is the set of strings accepted by the regex
  • This set is called a regular set, may be finite or infinite
• Easy to automate
• Used in many contexts:
  • Lexical scanning
  • UNIX grep, sed, awk, etc.
  • perl, python, Java, C++ (11), etc.
• However, not as powerful as context-free grammars

Regular Expressions: Formalism

• Regular expression operate on a finite character set, aka alphabet (denoted Σ)
  • ASCII
  • Unicode
  • Also define an empty (or null) string, λ
• Three operations for building up strings:
  • Catenation
  • Alternation
  • Kleene closure

Catenation

• Joining individual characters:
  • ab – accepts the string “ab”
• Joining strings:
  • abc – accepts the string “abc”
  • λa = aλ = a – accepts the string “a”
• Joining sets of strings:
  • If P and Q are sets of strings, then:
    • s₁ in P, s₂ in Q implies s₁s₂ in PQ
  • By extension, we can concatenate any regular expressions

Alternation

• Finite sets:
  • (a | b | c) – accepts any of the three strings “a”, “b”, “c”
• Strings:
  • (ab | λ) – accepts the string “ab”, or the empty string “”
• Sets:
  • (P | Q) – accepts any string from P or from Q
  • By extension, alternation on regular expressions

Kleene Closure

Operator * - postfix Kleene closure operator

For set S:
• S* represents all strings formable as catenation of zero or more strings from S.
• λ implicitly a part of any set represented by a Kleene closure
Regular Expressions

• $\emptyset$ is the empty set
• $\lambda$ denotes the set containing only the empty string
• $s$ denotes the set containing only the string “s”
• If $A, B$ are regular expressions, then so are:
  • $A | B$
  • $AB$
  • $A^*$

Extensions

• $P^*$ - positive closure: one or more string in $P$ catted together
  • $P^+$ is equivalent to $PP^*$
• $P?$ – zero or 1 string in $P$
  • Equivalent to $P | \lambda$
  • Common in regex packages to avoid need for $\lambda$
• $\text{Not}(A)$, where $A$ is a set of characters = all characters not in $A$ (notations vary)
  • Many more...

Examples

• int literal
• Fixed decimal literal
• C-style double literal
• Jack identifier

Finite Automata

A finite automata (aka finite state machine) consists of:
• A (finite) set of states
• A finite alphabet, $\Sigma$
• A set of transitions from one state to another, labeled with characters from $\Sigma$
• A distinguished start state
• A subset of the states known as accepting, or final states

Examples

A finite automata is a kind of graphical model; so we usually draw it:

- Denotes a state
- Denotes a transition on symbol s
- Denotes the start state
- Denotes an accepting state

A finite automata

- Denotes a state
- Denotes a transition on symbol s
- Denotes the start state
- Denotes an accepting state
More Example

Accepts (abc)|(xyz*)

Kinds of Finite Automata

• Basically, two flavors:
  • Deterministic (transitions are unique for a given state and symbol)
  • Non-deterministic (multiple possible transitions on a given symbol, or even on λ)
• DFAs are easily automated
  • Table representation – states by states – each non-empty entry represents a transition
  • No ambiguity
  • Basis for regex-based lexer tools
• NFAs are used primarily as a transition stage
  • Regular expressions easily converted to NFAs
  • NFAs convertible to DFAs (this is a bit involved)
  • All of this converting can be automated!

Tabular Representation

Note table size is quadratic in number of states, but very sparse. A sparse representation is thus generally preferred.

Scanner Generators

A variety of tools available for different languages:
• lex, flex – C
• PLY – python
• Jflex – java

Various approaches, but commonly driven by regular expressions:
• You supply regular expressions for each type of token
• The tool creates or provides a scanner based on your regular expressions

Writing a Scanner

Book suggests a certain structure for scanner ("JackTokenizer")
• Very similar to what you’ve seen previously for VM translator
• Key method is called “advance” – get the next token from the input

Q. How do you get the next token?
• Big if statement
• Lots of lookahead?
• Buffering (this is probably the most annoying bit)

Writing a Scanner Using Regular Expressions

Buffering:
• Buffer line-by-line
  – or –
• Just read the entire file into one big string!

Tokenizing:
• Create/store/compile regular expressions for every token type, whitespace, and comments
• At each “advance” step:
  • Try each regular expression in turn on current start of string
  • When a matching regular expression is found:
    • Move a pointer/index into your string to just past the matched string
    • Return the token corresponding to the matched regular expression
Regular Expressions: Python

# find an integer constant at the start of the string
import re
foo = "42+36"
expr = "[0-9]+"
pattern = re.compile(expr)
if pattern.match(foo):
    print(pattern.match(foo).group(0))

See [https://docs.python.org/3/library/re.html](https://docs.python.org/3/library/re.html) for full docs

Regular Expressions: Java

// find an integer constant at the start of the string
import java.util.regex.*;
public class Main {
    public static void main(String[] args) {
        String foo = "42+36";
        String expr = "[0-9]+";
        Pattern p = Pattern.compile(expr);
        Matcher m = p.matcher(foo);
        if (m.lookingAt()) {
            System.out.println(m.group(0));
        }
    }
}

See [http://docs.oracle.com/javase/7/docs/api/](http://docs.oracle.com/javase/7/docs/api/) for full docs

Regular Expressions: C++

#include <regex>
#include <string>
#include <iostream>
using namespace std;

int main() {
    string foo = "42+36";
    regex pattern("[0-9]+");
    regex_search(foo, m, pattern, regex_constants::match_continuous);
    if (!m.empty()) {
        cout << m.str() << endl;
    } else {
        cout << "No match." << endl;
    }
    return 0;
}

Note: requires g++ 4.9 on linux.