Lecture 20: Regular expressions and context-free grammars

Languages: sets of strings

Sets of strings that satisfy special properties are called languages.

Examples:
- English sentences
- Syntactically correct Java/C/C++ programs
- $\Sigma^* =$ All strings over alphabet $\Sigma$
- Palindromes over $\Sigma$
- Binary strings that don’t have a 0 after a 1
- Legal variable names, keywords in Java/C/C++
- Binary strings with an equal # of 0’s and 1’s

Regular expressions

Regular expressions over $\Sigma$

- Basis:
  - $\emptyset$ and $\varepsilon$ are regular expressions
  - $a$ is a regular expression for any $a \in \Sigma$

- Recursive step:
  - If $A$ and $B$ are regular expressions then so are:
    - $(A \cup B)$
    - $(AB)$
    - $A^*$

Examples

- $001^*$
- $0^*1^*$
- $(0 \cup 1)0(0 \cup 1)0$
- $(0^*1^*)^*$
- $(0 \cup 1)^*0110(0 \cup 1)^*$
- $(00 \cup 11)^*(01010 \cup 10001)(0 \cup 1)^*$

Each regular expression is a “pattern”

- $\varepsilon$ matches the empty string
- $a$ matches the one character string $a$
- $(A \cup B)$ matches all strings that either $A$ matches or $B$
  matches (or both)
- $(AB)$ matches all strings that have a first part that $A$
  matches followed by a second part that $B$
  matches
- $A^*$ matches all strings that have any number of strings (even
  0) that $A$ matches, one after another

Regular expressions in practice

- Used to define the “tokens”: e.g., legal variable names,
  keywords in programming languages and compilers
- Used in grep, a program that does pattern matching searches
  in UNIX/LINUX
- Pattern matching using regular expressions is an essential
  feature of PHP
- We can use regular expressions in programs to process
  strings!
regular expressions in Java

- Pattern p = Pattern.compile("a*b");
- Matcher m = p.matcher("aaaaab");
- boolean b = m.matches();

 matcher email addresses: RFC 822

\[01\]  a 0 or a 1  ^ start of string  $ end of string\]
\[0–9\]  any single digit  \. period  \, comma  \- minus
\ .  any single character
ab  a followed by b  (AB)
\(a|b\)  a or b  (A \cup B)
a?  zero or one of a  (A \cup \emptyset)
a*  zero or more of a  A*  
a+  one or more of a  AA*  
\ e.g.  ^\[\-+\]0–9\]0–9\]+S
General form of decimal number e.g. 9.12 or -9,8 (Europe)

more examples

- All binary strings that have an even # of 1’s
- All binary strings that don’t contain 101

limitations of regular expressions

- Not all languages can be specified by regular expressions
- Even some easy things like
  – Palindromes
  – Strings with equal number of 0’s and 1’s
- But also more complicated structures in programming languages
  – Matched parentheses
  – Properly formed arithmetic expressions
  – etc.

context-free grammars

- A Context-Free Grammar (CFG) is given by a finite set of substitution rules involving
  – A finite set \(V\) of variables that can be replaced
  – Alphabet \(\Sigma\) of terminal symbols that can’t be replaced
  – One variable, usually \(S\), is called the start symbol
- The rules involving a variable \(A\) are written as
  \[A \rightarrow w_1 \mid w_2 \mid \cdots \mid w_k\]
  where each \(w_i\) is a string of variables and terminals:
  \(w_i \in (V \cup \Sigma)^*\)

how CFGs generate strings

- Begin with start symbol \(S\)
- If there is some variable \(A\) in the current string you can replace it by one of the \(w_i\)’s in the rules for \(A\)
  \(- A \rightarrow w_1 \mid w_2 \mid \cdots \mid w_k\)
  \(- Write this as  xAy \rightarrow xw_iy\)
  \(- Repeat until no variables left\)
- The set of strings the CFG generates are all strings produced in this way that have no variables
Example:  \[ S \rightarrow 0S | 1S | 0 | 1 | \varepsilon \]

Example:  \[ S \rightarrow 0S | S1 | \varepsilon \]

---

**simple arithmetic expressions**

\[ E \rightarrow E+E | E+E | (E) | x | y | z | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \]

Generate \((2 \times x) + y\)

Generate \(x + y + z\) in two fundamentally different ways

---

**CFGs and recursively-defined sets of strings**

- A CFG with the start symbol \(S\) as its only variable recursively defines the set of strings of terminals that \(S\) can generate

- A CFG with more than one variable is a simultaneous recursive definition of the sets of strings generated by each of its variables
  
  - Sometimes necessary to use more than one

---

**Grammar for \(0^n1^n : n \geq 0\)**

(all strings with same \# of 0’s and 1’s with all 0’s before 1’s)

Example:  \[ S \rightarrow (S) | SS | \varepsilon \]

---

**parse trees**

Suppose that grammar \(G\) generates a string \(x\)

A parse tree of \(x\) for \(G\) has

- Root labeled \(S\) (start symbol of \(G\))
- The children of any node labeled \(A\) are labeled by symbols of \(w\) left-to-right for some rule \(A \rightarrow w\)
- The symbols of \(x\) label the leaves ordered left-to-right

\[ S \rightarrow 0S0 | 1S1 | 0 | 1 | \varepsilon \]

Parse tree of \(0110\):

---

**building precedence in simple arithmetic expressions**

- \(E\) – expression (start symbol)
- \(T\) – term
- \(F\) – factor
- \(I\) – identifier
- \(N\) – number

\[ E \rightarrow T | E+T \]

\[ T \rightarrow F | F+T \]

\[ F \rightarrow (E) | I | N \]

\[ I \rightarrow x | y | z \]

\[ N \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \]
Backus-Naur form (same as CFG)

BNF (Backus-Naur Form) grammars

- Originally used to define programming languages
- Variables denoted by long names in angle brackets, e.g.
  `<identifier>`, `<if-then-else-statement>`,
  `<assignment-statement>`, `<condition>`
  `::=` used instead of `→`

BNF for C

```
statement: (identifier | "case" constant-expression | "default") "="? expression "|" statement |
           "if" ? expression "then" ? statement |
           "if" ? expression "then" ? statement "else" ? statement |
           "do" expression "while" ? expression "|" statement |
           "for" ? expression "|" expression "|" expression "|" statement |
           "goto" identifier ";" |
           "continue" ";" |
           "return" expression ";"
block: "{" declaration ? statement ";"
expression: assignment-expression
assignment-expression: ( unary-expression | "="? expression 
                        | "="? expression "="? expression |
                                        | ";" |
                                        | "(" expression-expression ";" conditional-expression |
conditional-expression: logical-or-expression |
                        logical-and-expression |
                        conditional-expression |
logical-or-expression: "(" expression-expression "|" conditional-expression |
logical-and-expression: "(" expression-expression "" and-expression |
                        logical-and-expression |
                        conditional-expression |
and-expression: "(" expression-expression "|" conditional-expression |
                        logical-and-expression |
                        conditional-expression |
statement: (identifier | "case" constant-expression | "default") "="? expression "|" statement |
           "if" ? expression "then" ? statement |
           "if" ? expression "then" ? statement "else" ? statement |
           "do" expression "while" ? expression "|" statement |
           "for" ? expression "|" expression "|" expression "|" statement |
           "goto" identifier ";" |
           "continue" ";" |
           "return" expression ";"
```