CSE 311 Foundations of Computing I

Lecture 18
Recursive Definitions: Regular Expressions, Context-Free Grammars and Languages
Spring 2013

Announcements

- Reading assignments
  - 7th Edition, pp. 878-880 and pp. 851-855
  - 6th Edition, pp. 817-819 and pp. 789-793
- For Wednesday, May 15
  - 7th Edition, Section 9.1 and pp. 594-601
  - 6th Edition, Section 8.1 and pp. 541-548

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 Σ
  - Palindromes over Σ
  - 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 Σ
- Basis:
  - $\varnothing$, λ 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***

Each regular expression is a "pattern"

- λ 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

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***)*

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 hypertext scripting language PHP used for web programming

– Also in text processing programming language Perl

Regular Expressions in PHP

- int **preg_match** (string $pattern , string $subject,...)
- $pattern syntax:

  `[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** $\cup$ λ)

  a***** zero or more of a **A***

  a**+** one or more of a **AA***

- e.g. **`^[\-+]?[0-9]*(\.|\,)?[0-9]+$`**

  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

Fact: Not all sets of strings 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 Σ 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

  $\mathbf{A} \rightarrow w_1 \mid w_2 \mid ... \mid w_k$ where each w_i is a string of variables and terminals – that is $w_i \in (\mathbf{V} \cup \Sigma)^*$

How Context-Free Grammars 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's in the rules for **A**
  - Write this as $x\mathbf{A}y \Rightarrow xwy$
  - Repeat until no variables left
- The set of strings the CFG generates are all strings produced in this way that have no variables

Sample Context-Free Grammars

- Example: $\mathbf{S} \rightarrow 0\mathbf{S}0 \mid 1\mathbf{S}1 \mid 0 \mid 1 \mid \lambda$

- Example: $\mathbf{S} \rightarrow 0\mathbf{S} \mid \mathbf{S}1 \mid \lambda$

Sample Context-Free Grammars

- 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: $\mathbf{S} \rightarrow (\mathbf{S}) \mid \mathbf{SS} \mid \lambda$

Simple Arithmetic Expressions

$\mathbf{E} \rightarrow \mathbf{E}+\mathbf{E} \mid \mathbf{E}*\mathbf{E} \mid (\mathbf{E}) \mid x \mid y \mid z \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Generate $(2*x) + y$

Generate $x+y*z$ in two fundamentally different ways

Context-Free Grammars 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

Building in 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

Another name for CFGs

- 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 $\rightarrow$

BNF for C

```
statement:
  ((identifier | "case" constant-expression | "default") ":")*
  (expression? ";" |
   block |
   "if" "(" expression ")" statement |
   "if" "(" expression ")" statement "else" statement |
   "switch" "(" expression ")" statement |
   "while" "(" expression ")" statement |
   "do" statement "while" "(" expression ")" ";" |
   "for" "(" expression? ";" expression? ";" expression? ")" statement |
   "goto" identifier ";" |
   "continue" ";" |
   "break" ";" |
   "return" expression? ";"
  )

block: "{" declaration* statement* "}"

expression:
  assignment-expression%

assignment-expression: (
    unary-expression (
      "=" | "*=" | "/=" | "%=" | "+=" | "-=" | "<<=" | ">>=" | "&=" |
      "^=" | "|="
    )
  )* conditional-expression

conditional-expression:
  logical-OR-expression ( "?" expression ":" conditional-expression )?
```


Parse Trees

Back to middle school:

<sentence>::=<noun phrase><verb phrase>

<noun phrase>::=<article><adjective><noun>

<verb phrase>::=<verb><adverb>|<verb><object>

<object>::=<noun phrase>

Parse:

The yellow duck squeaked loudly

The red truck hit a parked car