Regular Expressions

Regular Expressions

One way to denote (often infinite) languages

Definition

A regular expression e is built from:

- $\varnothing$, corresponding to the empty language
- ε, corresponding to $\{\varepsilon\}$
- a, b, etc. corresponding to $\{\mathrm{a}\}, \{\mathrm{b}\}, \ldots$
- $e_1 \mid e_2$ corresponding to $L_{e_1} \cup L_{e_2}$
- $e_1 e_2$ corresponding to $L_{e_1} \cdot L_{e_2}$
- e^* corresponding to L_e^*

Example: $\mathit{letter}(\mathit{letter} \mid \mathit{digit})^*$
where $\mathit{letter} = \mathrm{a} \mid \mathrm{b} \mid \mathrm{c} \mid \ldots$ and $\mathit{digit} = 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Meaning of Regular Expressions

Regular expressions are just a notation for some operations on languages

$letter(letter \,|\, digit)^*$ denotes the set

$$L_{letter} \cdot (L_{letter} \,|\, L_{digit})^*$$

where $L_{letter} = \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \ldots\}$ and $L_{digit} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Meaning of Regular Expressions

Regular expressions are just a notation for some operations on languages

letter(*letter* | *digit*)* denotes the set

$$L_{letter} \cdot (L_{letter} \mid L_{digit})^*$$

where $L_{letter} = \{a, b, c, \ldots\}$ and $L_{digit} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Every finite language $\{w_1, \ldots, w_n\}$ can be described using a regexp $w_1 \mid \ldots \mid w_n$

As well as the class of *regular* languages, containing some infinite languages

Derived Regular Expression Operators

Some new operators can be defined in terms of previous ones.

- Shorthands for finite languages, such as $[\mathrm{a..z}] = \mathrm{a} \,|\, \mathrm{b} \,|\, \ldots \,|\, \mathrm{z}$
- Optional $e^? =$

Derived Regular Expression Operators

Some new operators can be defined in terms of previous ones.

- Shorthands for finite languages, such as $[\mathrm{a..z}] = \mathrm{a} \,|\, \mathrm{b} \,|\, \ldots \,|\, \mathrm{z}$
- Optional $e^? = e \,|\, \varepsilon$
- Repeat at least once $e^+ =$

Derived Regular Expression Operators

Some new operators can be defined in terms of previous ones.

- Shorthands for finite languages, such as $[\mathrm{a..z}] = \mathrm{a} \,|\, \mathrm{b} \,|\, \ldots \,|\, \mathrm{z}$
- Optional $e^? = e \,|\, \varepsilon$
- Repeat at least once $e^+ = ee^*$
- Other repetitions $e^{k..*} =$

Derived Regular Expression Operators

Some new operators can be defined in terms of previous ones.

- Shorthands for finite languages, such as $[\mathsf{a..z}] = \mathsf{a} \,|\, \mathsf{b} \,|\, \ldots \,|\, \mathsf{z}$
- Optional $e^? = e \,|\, \varepsilon$
- Repeat at least once $e^+ = ee^*$
- Other repetitions $e^{k..*} = e^k e^*$ and $e^{p..q} =$

Derived Regular Expression Operators

Some new operators can be defined in terms of previous ones.

- Shorthands for finite languages, such as [a..z] = a | b | ... | z
- Optional $e^? = e \mid \varepsilon$
- Repeat at least once $e^+ = ee^*$
- Other repetitions $e^{k..*} = e^k e^*$ and $e^{p..q} = e^p (e^?)^{q-p}$
- Complement: $!e$, denoting $A^* \setminus L_e$

Derived Regular Expression Operators

Some new operators can be defined in terms of previous ones.

- Shorthands for finite languages, such as [a..z] $=$ a | b | ... | z
- Optional $e^? = e \mid \varepsilon$
- Repeat at least once $e^+ = ee^*$
- Other repetitions $e^{k..*} = e^k e^*$ and $e^{p..q} = e^p (e^?)^{q-p}$
- Complement: $!e$, denoting $A^* \setminus L_e$ — non-obvious translation into base operators!
- Intersection $e_1 \,\&\, e_2 =$

Derived Regular Expression Operators

Some new operators can be defined in terms of previous ones.

- Shorthands for finite languages, such as $[\mathsf{a..z}] = \mathsf{a} \,|\, \mathsf{b} \,|\, \ldots \,|\, \mathsf{z}$
- Optional $e^? = e \,|\, \varepsilon$
- Repeat at least once $e^+ = ee^*$
- Other repetitions $e^{k..*} = e^k e^*$ and $e^{p..q} = e^p(e^?)^{q-p}$
- Complement: $!e$, denoting $A^* \setminus L_e$ — non-obvious translation into base operators!
- Intersection $e_1 \,\&\, e_2 = !(!e \,|\, !e)$, denoting $L_{e_1} \cap L_{e_2}$

An Interesting Connection

The language of regular expressions is equivalent to
monadic second-order logic of strings.

Meaning: a logic where the objects of study are *strings*, and which is *monadic* — may quantify only over single-argument predicates (or sets)

Example: $\{a, ab\}^* \;=\; \{\, w \in \{a, b\}^* \mid \forall i \in \mathbb{N}.\; w_{(i)} = b \implies i > 0 \land w_{(i-1)} = a \,\}$

Computability of Regular Expression Predicates

Nice property of regular expressions/languages:
most questions about them can be answered algorithmically

Emptiness $L_e = \varnothing$

Inclusion $L_{e_1} \subseteq L_{e_2}$

Disjointness $L_{e_1} \cap L_{e_2} = \varnothing$

etc.

We will see that these quickly become undecidable for more complex languages.

Lexical Analysis

Lexical Analysis

Input: character streams
  such as `res = 14 + arg * 3`

Lexer yields: "`res`", "`=`", "`14`", "`+` ","`arg`", "`*` ","`3`",

Lexical analyzer (lexer, scanner, tokenizer) usually specified
  through regular expressions for each kind of token

Groups characters and maps streams of characters to streams of tokens

Tokens could be strings, but are better represented as structured data

Lexical Analyzer – Key Ideas

Typically needs only *small* amounts of *memory*.

Not difficult to construct manually.

Typically use the first character to decide on the token class

$$\mathit{first}(L) = \{\, \mathsf{a} \mid \mathsf{a}w \in L \,\}$$

Use the *longest match* rule:

*Eagerly accept the **longest** token that can be recognized at each point, regardless of what follows.*

Automatic Derivation of Lexical Analyzers

Implementing lexical analyzers can be *automated*. Done either through:

- Conversion to finite-state automata.
- Usage of regular expression *derivation*.

Example "compiler compiler" tools to derive lexers: JavaCC, Lex, ocamllex

While Language – A Program

```
num = 13;
while (num > 1) {
  println("num = ", num);
  if (num % 2 == 0) {
    num = num / 2;
  } else {
    num = 3 * num + 1;
  }
}
```

Tokens (Words) of the *While* Language

Ident ::=

letter (letter | digit)*

integerConst ::= digit digit*

regular expressions

keywords

if else while println

special symbols

( ) && < == + - * / % ! - { } ; ,

letter ::= a | b | c | … | z | A | B | C | … | Z

digit ::= 0 | 1 | … | 8 | 9

Manually Constructing Lexers by example

Definition of Tokens

```
enum Token:
  case ID(content: String) // id3
  case IntConst(value: Int) // 10
  case object AssignEQ
  case CompareEQ
  case MUL // *
  case PLUS // +
  case LEQ // <=
  case OPAREN
  case CPAREN
  case IF
  case WHILE
  case EOF // End Of File
```

Definition of Lexer

```scala
enum Token:
  case ID(content: String)
  case IntConst(value: Int)
  case object AssignEQ
  case CompareEQ
  case MUL // *
  case PLUS // +
  case LEQ // <=
  case OPAREN
  case CPAREN
  case IF
  case WHILE
  case EOF // End Of File
```

```scala
class CharStream(fileName: String):
  val file = new BufferedReader(
    new FileReader(fileName))
  var current: Char = '\0x0'
  var eof: Boolean = false
  def next =
    if (eof) throw
      EndOfInput("reading" + file)
    val c = file.read()
    eof = (c == -1)
    current = c.toChar
  next // initialize first char

class Lexer(ch: CharStream):
  var current: Token
  def next: Unit =
    // lexer code goes here
```

Recognizing Identifiers and Keywords

```
if (isLetter) {
  b = new StringBuffer
  while (isLetter || isDigit) {
     b.append(ch.current)
     ch.next
  }
 keywords.lookup(b.toString) {
  case None=> token=ID(b.toString)
  case Some(kw) => token=kw
 }
}
```

regular expression for identifiers:

letter (letter|digit)*

Keywords look like identifiers, but are simply indicated as keywords in language definition. Introduce a constant Map from strings to keyword tokens. If not in map, then it is ordinary identifier.

Integer Constants and Their Value

regular expression for integers:

digit digit*

```
if (isDigit) {
  k = 0
  while (isDigit) {
    k = 10*k + toDigit(ch.current)
    ch.next
  }
  token = IntConst(k)
}
```

Deciding which Token is Coming

- How do we know when we are supposed to analyze string, when integer sequence etc?
- Manual construction: use **lookahead** (next symbol in stream) to decide on token class
- compute first(e) - symbols with which e can start
- check in which first(e) current token is
- If $L \subseteq A^*$ is a language, then first(L) is set of all alphabet symbols that start some word in L

$$\mathrm{first}(L) = \{a \in A \mid \exists v \in A^* .\ a\,v \in L\}$$

First Symbols of a Set of Words

first({a, bb, ab}) = {a,b}

first({a, ab}) = {a}

first({aaaaaaa}) = {a}

first({a}) = {a}

first({}) = {}

first({ε}) = {}

first({ε,ba}) = {b}

First Symbols of a Regexp

Examples:

- $\mathit{first}(\mathrm{ab}^*) = \{\mathrm{a}\}$
- $\mathit{first}(\mathrm{ab}^* \mid \mathrm{c}) = \{\mathrm{a}, \mathrm{c}\}$
- $\mathit{first}(\mathrm{a}^*\mathrm{b}^*\mathrm{c}) = \{\mathrm{a}, \mathrm{b}, \mathrm{c}\}$
- $\mathit{first}((\mathrm{cb} \mid \mathrm{a}^*\mathrm{c}^*)\mathrm{d}^*e))$

First Symbols of a Regexp

Examples:

- $\mathit{first}(\mathrm{ab}^*) = \{\mathrm{a}\}$
- $\mathit{first}(\mathrm{ab}^* \,|\, \mathrm{c}) = \{\mathrm{a}, \mathrm{c}\}$
- $\mathit{first}(\mathrm{a}^*\mathrm{b}^*\mathrm{c}) = \{\mathrm{a}, \mathrm{b}, \mathrm{c}\}$
- $\mathit{first}((\mathrm{cb} \,|\, \mathrm{a}^*\mathrm{c}^*)\mathrm{d}^* e)) = \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}\}$

How to compute these?

Needs a notion of nullability.

Can use automata, or can use regexp derivation.