Semantics of regular expressions

Semantics

The semantics of a regular expression over A is a language over A

- $\emptyset \rightsquigarrow$ the empty set
- $\epsilon \rightsquigarrow \{\epsilon\}$
- $\sigma \rightsquigarrow \{\texttt{"}\sigma\texttt{"}\}$, for all $\sigma \in A$
- $e_1|e_2 \rightsquigarrow$ union of the semantics of e_1 and e_2
- $e_1\ e_2 \rightsquigarrow$ concatenation of the semantics of e_1 and e_2

Concrete syntax of regular expressions

Precedence and associativity

- the Kleene star has higher precedence than concatenation and union
- concatenation has higher precedence than union
- concatenation and union are left associative

Derived operators and extended notation (Java API syntax)

- $e{+}{=}ee{\star}$ (one or more times e)
- ϵ is represented by the empty string: `a|`ϵ becomes `a|`
- e`?`=`|`e (e is optional, that is, once or not at all)
- `[a0B]` any of the characters between brackets (that is `a|0|B`)
- `[b-d]` any of the characters in the range between brackets (that is `b|c|d`)
- `[a0B]|[b-d]` can be written in the more compact way `[a0Bb-d]`
- `[^...]` any character except for . . .
  Example: `[^a0Bb-d]` any character except for `a`, `0`, `B`, `b`, `c`, `d`

Concrete syntax of reguar expressions

Special characters (Java API syntax)

- . means any character
- \ is the escape character to quote the next character(s)

Quoted characters

The \ character is used to give

- ordinary meaning to special characters
- special meaning to ordinary characters

Concrete syntax of reguar expressions

Special characters that have an ordinary meaning with

Examples: `\|`, `\*`, `\+`, `\?`, `\.`, `\\`

Special meaning

Examples:

- `\t`: tab
- `\n`: newline (=line feed)
- `\s`: any white space character
- `\S`: any non-white space character
- `\d`: any digit character (`[0-9]`)
- `\D`: any non-digit character (`[^0-9]`)
- `\w`: any word character (`[[a-zA-Z_0-9]]`)
- `\W`: any non-word character (`[^\w]`)

Simple examples of regular languages

Definition

A language is called *regular* if it can be defined by a regular expression.

Examples

- identifiers `(a|...|z|A|...Z)(a|...|z|A|...Z|0|...|9)*` compares with
  $L_{id} = \{ \texttt{"a"}, \ldots, \texttt{"z"} \} \cup \{ \texttt{"A"}, \ldots, \texttt{"Z"} \} \cdot$
  $(\{\text{'a'}, \ldots, \text{'z'}\} \cup \{\text{'A'}, \ldots, \text{'Z'}\} \cup \{\text{'0'}, \ldots, \text{'9'}\})^*$
- numbers (radix 10): `0|(1|...|9)(0|...|9)*`
- numbers (radix 8): `0(0|...|7)*`

Where are regular expressions used?

Main use cases

- definition of lexers/tokenizers (see the following slides)
- data validation (example: web forms)
- text manipulation (example: find & replace in text editors)

Lexical analysis

Lexeme

A substring which is considered a syntactic *unit*

Lexical analysis

The problem of decomposing a string in *lexemes*

Lexer (or scanner)

A program which performs lexical analysis and generates lexemes

Example in C

The string `"x2=042;"` is decomposed in the following lexemes:

- `"x2"`
- `"="`
- `"042"`
- `";"`

Lexical analysis

Token

- More abstract than the notion of lexeme
- A token corresponds to some kind of lexemes
- Example: identifiers, numbers, the assignment operator, . . .
- In some cases it can carry semantic information: numbers → their values

Tokenizer

A program which performs lexical analysis and generates tokens

Example in C

The string `"x2=042;"` is decomposed in the following tokens:

- `IDENTIFIER` with name `"x2"`
- `ASSIGN_OP`
- `INT_NUMBER` with value thirty-four
- `STATEMENT_TERMINATOR`