# Semantics of regular expressions

## Semantics

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

- pattern used to define the semantics:
  specific syntax case $\rightsquigarrow$ its semantics (= a language over $A$ = a set of strings over $A$)
- $\emptyset \rightsquigarrow$ the empty set
- $\epsilon \rightsquigarrow \{\epsilon\}$
- $\sigma \rightsquigarrow \{"\sigma"\}$, 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 $e_2$
- $e* \rightsquigarrow$ the Kleene star of the semantics of $e$
- $(e) \rightsquigarrow$ the semantics of $e$

# Syntax of regular expressions more in details

## Precedence and associativity of operators

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

  Examples:

  - $e_1 \mid e_2 \mid e_3$ corresponds to $(e_1 \mid e_2) \mid e_3$
  - $e_1\ e_2\ e_3$ corresponds to $(e_1\ e_2)\ e_3$

  Remark: in this case left associativity has an impact on the syntax, but not on the semantics, because union and concatenation of languages are associative
- as usual, parentheses can force the precedence rules.

  Example:

  $\texttt{a|bc} \rightsquigarrow \{\texttt{"a"}\} \cup (\{\texttt{"b"}\} \cdot \{\texttt{"c"}\}) = \{\texttt{"a"}, \texttt{"bc"}\}$
  $\texttt{(a|b)c} \rightsquigarrow (\{\texttt{"a"}\} \cup (\{\texttt{"b"}\}) \cdot \{\texttt{"c"}\} = \{\texttt{"ac"}, \texttt{"bc"}\}$

# Concrete syntax of regular expressions

## Concrete syntax and derived operators (Java API syntax)

- $e$+ means one or more times $e$, that is, $e+ = ee*$ (same precedence as $*$)
- $e$? means $e$ is optional, that is, $e? = \epsilon \mid e = e \mid \epsilon$ (same precedence as $*$)
- [...] means any of the strings of length 1 in square brackets

  **Example**: `[a4B] = a|4|B`
- [...-...] means any of the strings of length 1 in the range in square brackets

  **Example**: `[b-d] = b|c|d`
- single strings of length 1 and ranges can be mixed in square brackets
  **Example**: `[a4Bb-d] = [a4B]|[b-d] = a|4|B|b|c|d`
- [^...] means any string of lenght 1 not in square brackets

  **Example**: `[^a4Bb-d] = e|f|...|z|A|C|...|Z|0|1|2|3|5|...|9|...`

Remark: $e = e'$ means that the two expressions $e$ and $e'$ have the same semantics, that is, they represent the same set of strings

# Examples with the concrete syntax

## Examples

- identifiers: `[a-zA-Z][a-zA-Z0-9]* =`
  `(a|...|z|A|...|Z)(a|...|z|A|...|Z|0|...|9)*`

  Compare with the more verbose syntax based on set notation:

  ({ "a", ..., "z" } ∪ { "A", ..., "Z" }) ·
  ({"a",...,"z"} ∪ {"A",...,"Z"} ∪ {"0",...,"9"})*
- integers (radix 10): `0|[1-9][0-9]* = 0|(1|...|9)(0|...|9)*`
- integers (radix 8): `0[0-7]* = 0(0|...|7)*`

# Concrete syntax of regular expressions

## Special characters (Java API syntax)

- . (dot) means any string of length 1
- \ (backslash) is the escape character to quote the next character(s)

## Quoted characters

The \ character is used to assign

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

Remark: the meaning of some characters depend form the context

**Examples:**

`a*` (the Kleene star operator), `[*]` (the asterisk character)

`-` (the `-` character), `[a-z]` (the range separator)

Suggestion: if you need an ordinary character *c* and you are not sure about its possible special meaning, then use always *c*

# Concrete syntax of regular expressions

## Special characters that have an ordinary meaning with \

Examples: `\|`, `\*`, `\+`, `\?`, `\(`, `\)`, `\[`, `\]`, `\.`, `\\`, `\-`, `\^`

## Ordinary characters that have a special meaning with \

Examples:

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

# Examples with the concrete syntax

## Revisited examples

- identifiers (with underscore allowed): `[a-zA-Z_]\w*`
  Remark: `\w` includes also the underscore character _
- integers (radix 10): `0|[1-9]\d*`

# 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

## Definitions

- lexeme: a substring of a string 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 recognizes lexemes
- tokenizer: an abstraction of lexer, preferred in practice

## Lexical and syntactic analysis

- lexical analysis: first step of syntactic analysis
- lexer: a software component of the syntax analyzer
- two levels approach:
  - upper level:
    program = string which is the concatenation of a sequence lexemes
  - lower level:
    different types of lexems correspond to different sets of strings on characters
    Examples: the indentifiers, the integer numbers, the operators, . . .

# Lexical analysis

## Example 1 in C/Java/C++/C#

String `"x2=042;"`

- syntactically correct
- decomposed in the following lexemes:
  - `"x2"`
  - `"="`
  - `"042"`
  - `";"`

# Lexical analysis

## Example 2 in C/Java/C++/C#

Remark: most of the syntactic errors are not detected by the lexer

String `"=x2;042"`

- not syntactically correct, but the lexer does not detect any error
- decomposed in the following lexemes:
  - `"="`
  - `"x2"`
  - `";"`
  - `"042"`

# Lexical analysis

## Example 3 in C/Java/C++/C#

Remark: most of the syntactic errors are not detected by the lexer

String `"2x=042;"`

- not syntactically correct, but the lexer does not detect any error
- decomposed in the following lexemes:
  - `"2"`
  - `"x"`
  - `"="`
  - `"042"`
  - `";"`

# Lexical analysis

## Example 4 in C/Java/C++/C#

String `"x2=\;"`

- not syntactically correct, the lexer detects the error
- partially decomposed in the following lexemes:
  - `"x2"`
  - `"="`
  - lexer error: no lexeme can start with `\`

# Lexical analysis

## Token versus lexeme

- Token more abstract than lexeme
- A token is defined by the following information:
  - a token type
  - optionally, syntactic/semantic data
- Examples: identifiers, numbers, the assignment operator, . . .

## Tokenizer

A lexer which recognizes lexemes and emits corresponding tokens

## Example in C/Java/C++/C#

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

- `IDENTIFIER` with a syntactic data: the name `"x2"` (that is, the lexeme)
- `ASSIGN_OP` with no further data
- `INT_NUMBER` with semantic data: the value thirty-four
- `STATEMENT_TERMINATOR` with no further data