xkcd—a webcomic of romance, sarcasm, math, and language by Randall Munroe

Regular Expressions

> *Some people, when confronted with a problem think "I know, I'll use regular expressions." Now they have two problems.*

From a post by Jamie Zawinski to Usenet newsgroup `alt.religion.emacs` in 1997.

Regular Expressions (Syntax)

- What *are* they? (syntax)
- What do they *mean*? (semantics)

- Can a langauge denoted by a regular expression be effectvely recognized? (Implementation)
- Can tokens in a programing language be effectively recognized? (Scanning)

Regular Expressions (Syntax)

1. Empty. $\emptyset$
2. Atom. Any single symbol of $a \in \Sigma$ is a regular expression.
3. Alternation. If r_1 is a regular expression and r_2 is a regular expression, then $(r_1 + r_2)$ is a regular expression.
4. Concatenation. If r_1 and r_2 are regular expressions, then $(r_1 \cdot r_2)$ is a regular expression.
5. Closure. If r is a regular expression, then $(r)^*$ is a regular expression.

Regular Expressions (Syntax)

We omit parentheses following well-known rules:

- The outermost pair of parentheses is distracting.
- Alternation and concatenation can be considered left associative. (Semantically they are associative operators, so it does not make much difference.)
- Closure binds more tightly than concatenation which binds more tightly than alternation. (Similar to exponentiation, multiplication, and addition in traditional arithmetic expressions.)

Also, we sometimes omit the concatenation operator $\cdot$ using mere juxtaposition to indicate concatenation.

Regular Expressions (Haskell)

```
data Regex a = Empty | Sym a | Star (Regex a)
  Alt (Regex a) (Regex a) | Concat (Regex a) (Reg
    deriving (Eq, Ord)

Alt (Sym 'a', Concat (Sym 'b', Sym 'c')) -- a+(bc
Concat (Alt (Sym 'a', Sym 'b'), Sym 'c') -- (a+b)
```

Regular Expressions (Examples)

$$a \cdot b$$
$$a + b$$
$$a \cdot (b + c)$$
$$a + b \cdot c$$
$$a^*$$
$$a + (b \cdot c)^*$$
$$(a + (b \cdot c))^*$$

(using the alphabet $\Sigma = \{a, b, c\}$)

Regular Expressions (Examples)

(Omitting the "centered dot" and using the $|$ for alternation.)

$$ab$$
$$a \mid b$$
$$a(b \mid c)$$
$$a \mid bc$$
$$a^*$$
$$a \mid (bc)^*$$
$$(a \mid (bc))^*$$

(using the alphabet $\Sigma = \{a, b, c\}$)

Regular Expressions (Examples)

$$01$$
$$101$$
$$1+0$$
$$1(0+1)$$
$$0+10$$
$$0^*$$
$$1+(01)^*$$
$$(0+(10)^*$$
$$(0+(10)^*)^*$$

(using the alphabet $\Sigma = \{0, 1\}$ and omitting the $\cdot$ symbol)

Regular Expressions (Semantics)

That is what regular expressions look like (syntax). What do regular expressions *mean* (semantics)?

Each regular expression denotes a formal language (a set of strings).

There are an infinite number of regular expressions. (Each one is finitely constructed.) Just like programs in programming languages.

We must find a way to define the meaning or denotation of each regular expression.

So, we define a (recursive) function from regular expressions (as pieces of syntax) to sets of strings (languages).

Regular Expressions (Informal Semantics)

1. Empty. The language with no strings.
2. Atom. The language with one string of length one–a itself. Note that the notation a is ambiguous. It stands for both the symbol and the language.
3. Alternation. $(r_1 + r_2)$ is the union of two languages.
4. Concatenation. $(r_1 \cdot r_2)$ is the set all strings beginning in one language and then followed by a string in the second.
5. Closure. $(r)^*$ is zero, one, or more strings.

Regular Expressions (Examples)

$$
\begin{aligned}
a \cdot b &= \{ab\} \\
a + b &= \{a, b\} \\
a \cdot (b + c) &= \{ab, ac\} \\
a + (b \cdot c) &= \{a, bc\} \\
a^* &= \{\epsilon, a, aa, aaa, \ldots\} \\
a + (b \cdot c)^* &= \{\epsilon, a, bc, bcbc, \ldots\} \\
(a + (b \cdot c))^* &= \{\epsilon, a, bc, aa, abc, bcbc, \ldots\}
\end{aligned}
$$

(using the alphabet $\Sigma = \{a, b, c\}$)

Other Definitions

Some authors replace one of the five cases of the definition

1. Empty. $\emptyset$ denoting the language with no strings

with another case

1. Epsilon. ϵ denoting the set consisting of the single string ""

Do you see the difference?

Notice the ϵ is superfluous as $\{\epsilon\}$ is represented by $\emptyset^*$. Can you prove this?

Regular Expressions (Formal Semantics)

A regular expression denotes a formal language (set of strings) over an alphabet A by means of a function $\mathcal{D}$. The function $\mathcal{D}$ takes a regular expression and associates with it a particular formal language. The function is defined recursively over the five cases of the inductive definition of the set of regular expressions.

1. Empty.

$$\mathcal{D}[\![\emptyset]\!] \quad = \quad \{\}$$

2. Atom. For each $a \in \Sigma$,

$$\mathcal{D}[\![a]\!] \quad = \quad \{a\}$$

3. Alternation.

$$\mathcal{D}[\![(r_1 + r_2)]\!] \quad = \quad \mathcal{D}[\![r_1]\!] \cup \mathcal{D}[\![r_2]\!]$$

Regular Expressions (Formal Semantics)

4. Concatenation.

$$\mathcal{D}[\![(r_1 \cdot r_2)]\!] = \{x \cdot y \mid x \in \mathcal{D}[\![r_1]\!],\, y \in \mathcal{D}[\![r_2]\!]\}$$

   where $x \cdot y$ is string concatenation.

5. Closure.

$$\mathcal{D}[\![(r)^*]\!] = \bigcup_i (\mathcal{D}[\![r]\!])^i$$

   where S^i is defined recursively as follows:

$$\begin{array}{rcl} S^0 & = & \{\epsilon\} \\ S^{i+1} & = & \left\{x \cdot y \mid x \in S,\, y \in S^i\right\} \end{array}$$

$*$Using the Definitions

$$\begin{aligned}
\mathcal{D}[\![(a \cdot b)]\!] &= \{x \cdot y \mid x \in \mathcal{D}[\![a]\!],\ y \in \mathcal{D}[\![b]\!]\} \\
&= \{x \cdot y \mid x \in \{a\},\ y \in \{b\}\} \\
&= \{a \cdot b\} = \{ab\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{D}[\![(a + (a \cdot b))]\!] &= \mathcal{D}[\![a]\!] \cup \mathcal{D}[\![(a \cdot b)]\!] \\
&= \{a\} \cup \{ab\} \\
&= \{a,\ ab\}
\end{aligned}$$

$$\mathcal{D}[\![(a + (b \cdot c))^*]\!] = \bigcup_i (\mathcal{D}[\![(a + (b \cdot c))]\!])^i$$

$$= \{\epsilon\} \cup \{a,\ bc\} \cup \mathcal{D}[\![(a + (b \cdot c))]\!]^2 \cup \ldots$$

$$= \{\epsilon,\ a,\ bc\} \cup \{aa,\ abc,\ bca,\ bcbc\} \cup \ldots$$

* Digression: Induction

Is the recursively defined function D well-defined? Yes.
When are such recursively defined functions well-defined? Over free-generated sets.

$*$ Digression: Induction

Consider the following inductive definition of a subset M of Nat, the natural numbers, using integer multiplication $\cdot$:

- $1 \in M$,
- if $n \in M$, then $9 \cdot n \in M$,
- if $n \in M$, then $23 \cdot n \in M$.

Suppose we define the function $g : M \to Nat$ inductively by:

- $g(1) = 1$,
- $g(9 \cdot n) = 9$,
- $g(23 \cdot n) = 23$.

Prove that 0=1!

More Regular Expressions

To make regular expressions more convenient, regular expressions are almost always extended with new notation. Here are some additional meta-symbols commonly seen (perhaps with different syntax). Regular expressions with these new meta-symbols can be defined in terms of the original definitions. Let r be a regular expression over the alphabet Σ.

1. Optional. $r? = (r + \emptyset^*)$
2. One or more. (This is a second and conflicting use of the meta-character $+$.) $r^+ = (r \cdot r^*)$
3. Any. $. = (a + b + \ldots + y + z)$ where $\Sigma = a, b, \ldots, y, z$.
4. Range. $[a - z] = (a + b + \ldots + y + z)$. (Assumes that Σ is ordered.)
5. Range complement. $[\neg c - x] = (a + b + y + z)$. (Assumes that Σ is ordered.)

Where are
regular expressions
used?

Grep

The program `grep` is a command-line utility for searching text originally written for Unix. The `grep` command searches text files for lines matching a given regular expression and prints matching lines to the programs standard output.

Grep

Suppose the file `preamble.txt` has the following lines:

```
We the People of the United States, in Order to form a more perfect
Union, establish Justice, insure domestic Tranquility, provide for the
common defence, promote the general Welfare, and secure the Blessings
of Liberty to ourselves and our Posterity, do ordain and establish
this Constitution for the United States of America.
```

```
> grep and preamble.txt
common defence, promote the general Welfare, and secure the Blessings
of Liberty to ourselves and our Posterity, do ordain and establish
```

```
> grep -i people preamble.txt
We the People of the United States, in Order to form a more perfect
```

```
> grep -w do preamble.txt
of Liberty to ourselves and our Posterity, do ordain and establish
```

(But does not match "domestic.")

```
> grep -E -w "(defense)|(defence)" preamble.txt
common defence, promote the general Welfare, and secure the Blessings
```

Practical Regular Expressions

`grep` *options regexp filename*

Options:

```
-E extended regular expression
-i ignore case
-x match whole line
-w match word in line
```

Syntax of regular expressions for `grep`

`|`	or	`$`	end of line
`.`	any	`?`	optional
`[]`	set	`{n,m}`	n through m times
`[^ ]`	set compliment	`()`	grouping

grep

```
grep -E -w -i '[a-f]{3,4}'     /usr/dict/words
```

grep

```
grep -E -w -i '[a-f]{3,4}'     /usr/dict/words
```

```
beef
dead
deaf
fade
feed
```

among others.

grep

```
grep -E '.{5,}'            /usr/dict/words
```

grep

```
grep -E '.{5,}'          /usr/dict/words
```

```
Aarhus
Aaron
Ababa
aback
abacus
...
zombie
zoology
Zoroastrian
zucchini
Zurich
zygote
```

grep

```
grep -i -E '[aeiou]{4,}' /usr/dict/words
```

grep

```
grep -i -E '[aeiou]{4,}' /usr/dict/words
```

```
aqueous
Hawaiian
IEEE
obsequious
onomatopoeia
pharmacopoeia
prosopopoeia
queue
Sequoia
```

grep

```
grep -i -E '(q[^u]|q$)'  /usr/dict/words
```

grep

```
grep -i -E '(q[^u]|q$)'  /usr/dict/words
```

```
CEQ
Colloq
IQ
Iraq
q
Qatar
QED
q's
seq
```

grep

Is there a regular expression that matches those words whose letters appear in alphabetical order?

```
grep -x -E <regular expression> /usr/dict/words
```

grep

Is there a regular expression that matches those words whose letters appear in alphabetical order?

```
grep -x -E <regular expression> /usr/dict/words
```

```
grep -x -E
'a?b?c?d?e?f?g?h?i?j?k?l?m?n?o?p?q?r?s?t?u?v?w?x?y?z?'
/usr/dict/words
```

grep

Is there a regular expression that matches those words whose letters appear in alphabetical order?

```
grep -x -E <regular expression> /usr/dict/words
```

```
grep -x -E
'a?b?c?d?e?f?g?h?i?j?k?l?m?n?o?p?q?r?s?t?u?v?w?x?y?z?'
/usr/dict/words
```

```
almost
begin
below
biopsy
dirty
empty
first
glory
```

(Some of the longer words.)

grep

Can we modify the previous example to allow double letters?

```
grep -x -E
'a*b*c*d*e*f*g*h*i*j*k*l*m*n*o*p*q*r*s*t*u*v*w*x*y*z*'
/usr/dict/words
```

```
accent
almost
biopsy
choosy
effort
floppy
glossy
knotty
```

(Some of the longer words.)

grep

```
grep -E -e '(y.*){3,}' /usr/dict/wordsA
```

grep

```
grep -E -e '(y.*){3,}' /usr/dict/wordsA
```

The words with at least three y's.

```
polytypy      chromosomal variation between populations
psychophysiology  the way mind and body interact
synonymy  the state of being synonymous
syzygy    straight-line alignment of 3 celestial bodies
```

Back references

```
grep -E -e '(.)\1\1' /usr/dict/words
```