Algorithm Design: String Searching II

Ștefan Ciobâcă, Dorel Lucanu

Faculty of Computer Science
Alexandru Ioan Cuza University, Iași, Romania
dlucanu@info.uaic.ro

PA 2019/2020

1. Regular Expressions
   - Recap
   - String Searching with Regular Expressions
   - Building the Abstract Syntax Tree, AST (parsing)
   - Language Defined by a Regular Expression

2. Automaton associated to a regular expression
   - Building the automaton
   - Searching Algorithm

Plan

1. Regular Expressions
   - Recap
   - String Searching with Regular Expressions
   - Building the Abstract Syntax Tree, AST (parsing)
   - Language Defined by a Regular Expression

2. Automaton associated to a regular expression
   - Building the automaton
   - Searching Algorithm

Plan

1. Regular Expressions
   - Recap
   - String Searching with Regular Expressions
   - Building the Abstract Syntax Tree, AST (parsing)
   - Language Defined by a Regular Expression

2. Automaton associated to a regular expression
   - Building the automaton
   - Searching Algorithm

Motivation: patterns in many text editors (e.g., Emacs)

From documentation (Emacs):

Pattern	Matches
.	Any single character except newline ("\n").
\\.	One period
[0-9]+	One or more digits
[^ 0-9]+	One or more non-digit characters
[A-Za-z]+	one or more letters
[-A-Za-z0-9]+	one or more letter, digit, hyphen
[_A-Za-z0-9]+	one or more letter, digit, underscore
[-_A-Za-z0-9]+	one or more letter, digit, hyphen, underscore
[[:ascii:]]+	one or more ASCII chars. (codepoint 0 to 127, inclusive)
[[:nonascii:]]+	one or more none-ASCII characters (For example, Unicode charac
[\n\t]+	one or more {newline character, tab, space}.

Demo cu Emacs

(Mathematical) Definition

Definition

The set of *regular expressions* over the alphabet Σ is is recursively defined as follows:

(Mathematical) Definition

Definition

The set of *regular expressions* over the alphabet Σ is is recursively defined as follows:

- ε, *empty* are regular expressions

(Mathematical) Definition

Definition

The set of *regular expressions* over the alphabet Σ is is recursively defined as follows:

- ε, *empty* are regular expressions
- any character in Σ is a regular expression;

(Mathematical) Definition

Definition

The set of *regular expressions* over the alphabet Σ is is recursively defined as follows:

- ε, *empty* are regular expressions
- any character in Σ is a regular expression;
- if e_1, e_2 are regular expressions, then $e_1 e_2$ is a regular expression;

(Mathematical) Definition

Definition

The set of *regular expressions* over the alphabet Σ is is recursively defined as follows:

- ε, *empty* are regular expressions
- any character in Σ is a regular expression;
- if e_1, e_2 are regular expressions, then $e_1 e_2$ is a regular expression;
- if e_1, e_2 are regular expressions, then $e_1 + e_2$ is a regular expression;

(Mathematical) Definition

Definition

The set of *regular expressions* over the alphabet Σ is is recursively defined as follows:

- ε, *empty* are regular expressions
- any character in Σ is a regular expression;
- if e_1, e_2 are regular expressions, then $e_1 e_2$ is a regular expression;
- if e_1, e_2 are regular expressions, then $e_1 + e_2$ is a regular expression;
- if e is a regular expression, then e^* is a regular expressions.

Often we use parentheses to show how the above rules were applied; e.g., $(a + b)^*$

The language defined by a regular expression 1/2

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

The language defined by a regular expression 1/2

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) =$

The language defined by a regular expression 1/2

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$

The language defined by a regular expression 1/2

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),

The language defined by a regular expression 1/2

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\mathtt{empty}) =$

The language defined by a regular expression 1/2

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\mathtt{empty}) = \emptyset$

The language defined by a regular expression 1/2

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\mathtt{empty}) = \emptyset$
- if e is un character

The language defined by a regular expression 1/2

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\texttt{empty}) = \emptyset$
- if e is un character then $L(e) = \{e\}$;

The language defined by a regular expression 1/2

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\mathtt{empty}) = \emptyset$
- if e is un character then $L(e) = \{e\}$;
- if $e = e_1 e_2$

The language defined by a regular expression 1/2

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\texttt{empty}) = \emptyset$
- if e is un character then $L(e) = \{e\}$;
- if $e = e_1 e_2$ then $L(e) = L(e_1)L(e_2) = \{w_1 w_2 \mid w_1 \in L(e_1), w_2 \in L(e_2)\}$;

The language defined by a regular expression 1/2

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\mathtt{empty}) = \emptyset$
- if e is un character then $L(e) = \{e\}$;
- if $e = e_1 e_2$ then $L(e) = L(e_1)L(e_2) = \{w_1 w_2 \mid w_1 \in L(e_1), w_2 \in L(e_2)\}$;
- if $e = e_1 + e_2$

The language defined by a regular expression 1/2

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\texttt{empty}) = \emptyset$
- if e is un character then $L(e) = \{e\}$;
- if $e = e_1 e_2$ then $L(e) = L(e_1)L(e_2) = \{w_1 w_2 \mid w_1 \in L(e_1), w_2 \in L(e_2)\}$;
- if $e = e_1 + e_2$ then $L(e) = L(e_1) \cup L(e_2)$;

The language defined by a regular expression 1/2

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\texttt{empty}) = \emptyset$
- if e is un character then $L(e) = \{e\}$;
- if $e = e_1 e_2$ then $L(e) = L(e_1)L(e_2) = \{w_1 w_2 \mid w_1 \in L(e_1), w_2 \in L(e_2)\}$;
- if $e = e_1 + e_2$ then $L(e) = L(e_1) \cup L(e_2)$;
- if $e = e_1^*$

The language defined by a regular expression 1/2

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\mathtt{empty}) = \emptyset$
- if e is un character then $L(e) = \{e\}$;
- if $e = e_1 e_2$ then $L(e) = L(e_1)L(e_2) = \{w_1 w_2 \mid w_1 \in L(e_1), w_2 \in L(e_2)\}$;
- if $e = e_1 + e_2$ then $L(e) = L(e_1) \cup L(e_2)$;
- if $e = e_1^*$ then $L(e) = \cup_{k \geq 0} L(e_1^k)$, where $L(e_1^0) = \{\varepsilon\}, L(e_1^{k+1}) = L(e_1)L(e_1^k)$;

The language defined by a regular expression 1/2

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\texttt{empty}) = \emptyset$
- if e is un character then $L(e) = \{e\}$;
- if $e = e_1 e_2$ then $L(e) = L(e_1)L(e_2) = \{w_1 w_2 \mid w_1 \in L(e_1), w_2 \in L(e_2)\}$;
- if $e = e_1 + e_2$ then $L(e) = L(e_1) \cup L(e_2)$;
- if $e = e_1^*$ then $L(e) = \cup_{k \geq 0} L(e_1^k)$, where $L(e_1^0) = \{\varepsilon\}, L(e_1^{k+1}) = L(e_1)L(e_1^k)$;
- if $e = (e_1)$

The language defined by a regular expression 1/2

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\texttt{empty}) = \emptyset$
- if e is un character then $L(e) = \{e\}$;
- if $e = e_1 e_2$ then $L(e) = L(e_1)L(e_2) = \{w_1 w_2 \mid w_1 \in L(e_1), w_2 \in L(e_2)\}$;
- if $e = e_1 + e_2$ then $L(e) = L(e_1) \cup L(e_2)$;
- if $e = e_1^*$ then $L(e) = \cup_{k \geq 0} L(e_1^k)$, where $L(e_1^0) = \{\varepsilon\}, L(e_1^{k+1}) = L(e_1)L(e_1^k)$;
- if $e = (e_1)$ then $L(e) = L(e_1)$.

Remark. The operator $\_^*$ is called Kleene star or Kleene closure.

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b + a)c) =$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b + a)c) = \{abc, aac\}$ and
$L((ab)^*) =$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b + a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b + a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b + a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$
$L(b) = \{b\}$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b+a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$
$L(b) = \{b\}$
$L(b+a) = L(b) \cup L(a) = \{b, a\}$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b + a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$
$L(b) = \{b\}$
$L(b + a) = L(b) \cup L(a) = \{b, a\}$
$L(a(b + a)) = L(a)L(b + a) = \{ab, aa\}$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b + a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$
$L(b) = \{b\}$
$L(b + a) = L(b) \cup L(a) = \{b, a\}$
$L(a(b + a)) = L(a)L(b + a) = \{ab, aa\}$
$L(c) = \{c\}$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b+a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$
$L(b) = \{b\}$
$L(b+a) = L(b) \cup L(a) = \{b, a\}$
$L(a(b+a)) = L(a)L(b+a) = \{ab, aa\}$
$L(c) = \{c\}$
$L(a(b+a)c) = L(a(b+a))L(c) = \{abc, aac\}$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b + a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$
$L(b) = \{b\}$
$L(b + a) = L(b) \cup L(a) = \{b, a\}$
$L(a(b + a)) = L(a)L(b + a) = \{ab, aa\}$
$L(c) = \{c\}$
$L(a(b + a)c) = L(a(b + a))L(c) = \{abc, aac\}$
$L((ab)^0) = \{\varepsilon\}$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b + a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$
$L(b) = \{b\}$
$L(b + a) = L(b) \cup L(a) = \{b, a\}$
$L(a(b + a)) = L(a)L(b + a) = \{ab, aa\}$
$L(c) = \{c\}$
$L(a(b + a)c) = L(a(b + a))L(c) = \{abc, aac\}$
$L((ab)^0) = \{\varepsilon\}$
$L((ab)^1) = L(ab)L((ab)^0) = L(a)L(b)\{\varepsilon\} = \{ab\}\{\varepsilon\} = \{ab\}$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b+a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$
$L(b) = \{b\}$
$L(b+a) = L(b) \cup L(a) = \{b, a\}$
$L(a(b+a)) = L(a)L(b+a) = \{ab, aa\}$
$L(c) = \{c\}$
$L(a(b+a)c) = L(a(b+a))L(c) = \{abc, aac\}$
$L((ab)^0) = \{\varepsilon\}$
$L((ab)^1) = L(ab)L((ab)^0) = L(a)L(b)\{\varepsilon\} = \{ab\}\{\varepsilon\} = \{ab\}$
$L((ab)^2) = L(ab)L(ab) = \{abab\}$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b + a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$
$L(b) = \{b\}$
$L(b + a) = L(b) \cup L(a) = \{b, a\}$
$L(a(b + a)) = L(a)L(b + a) = \{ab, aa\}$
$L(c) = \{c\}$
$L(a(b + a)c) = L(a(b + a))L(c) = \{abc, aac\}$
$L((ab)^0) = \{\varepsilon\}$
$L((ab)^1) = L(ab)L((ab)^0) = L(a)L(b)\{\varepsilon\} = \{ab\}\{\varepsilon\} = \{ab\}$
$L((ab)^2) = L(ab)L(ab) = \{abab\}$
$L((ab)^3) = L(ab)L((ab)^2) = \{ababab\}$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b+a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$
$L(b) = \{b\}$
$L(b+a) = L(b) \cup L(a) = \{b, a\}$
$L(a(b+a)) = L(a)L(b+a) = \{ab, aa\}$
$L(c) = \{c\}$
$L(a(b+a)c) = L(a(b+a))L(c) = \{abc, aac\}$
$L((ab)^0) = \{\varepsilon\}$
$L((ab)^1) = L(ab)L((ab)^0) = L(a)L(b)\{\varepsilon\} = \{ab\}\{\varepsilon\} = \{ab\}$
$L((ab)^2) = L(ab)L(ab) = \{abab\}$
$L((ab)^3) = L(ab)L((ab)^2) = \{ababab\}$
...
$L((ab)^*) = \bigcup_{k \geq 0} L((ab)^k) = \{\varepsilon, ab, abab, ababab, \ldots\}$

Plan

1. Regular Expressions
   - Recap
   - String Searching with Regular Expressions
   - Building the Abstract Syntax Tree, AST (parsing)
   - Language Defined by a Regular Expression

2. Automaton associated to a regular expression
   - Building the automaton
   - Searching Algorithm

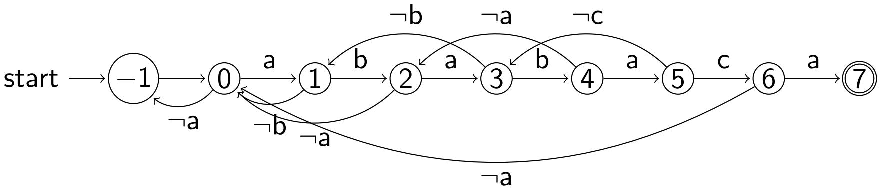
The Problem

Input A text s, a pattern p described by a regular expression e

Output: The first occurrence of a string belonging to the language defined by e

Example:
$s =$ "It will have blood, they say; blood will have blood."
$e =$ b[a-z]*d
Result: "It will have blood, they say; blood will have blood."

Recap: KMP – failure function as an automaton



Solution: Main steps

1. specify the regular expression as an data structure (abstract syntax tree, AST)
2. build the automaton associated to the regular expression
   the language defined by the expression = the language accepted by the automaton
3. use the automaton in the search process (naive algorithm, for now)

Plan

1. Regular Expressions
   - Recap
   - String Searching with Regular Expressions
   - Building the Abstract Syntax Tree, AST (parsing)
   - Language Defined by a Regular Expression
2. Automaton associated to a regular expression
   - Building the automaton
   - Searching Algorithm

Definition AST 1/2

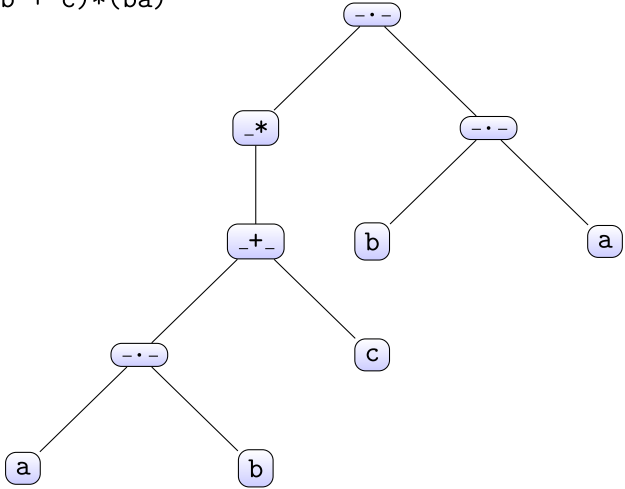
- data structure representing how the expression is built applying the rules from the definition
- node: the operation corresponding to the rule
  the subtrees of a node: thr ASTs corresponding to the subexpressions
  parentheses are not required to be explicitly represented, they give the structure of the tree
- an alternative definition for regular expressions using BNF[1] notation:

```
Exp   ::=  empty
         | ε
         | Sigma
         | Exp Exp          [_._]
         | Exp "+" Exp      [_+_]
         | "(" Exp ")"
         | Exp "*"          [_*]
```

[1] Backus–Naur form

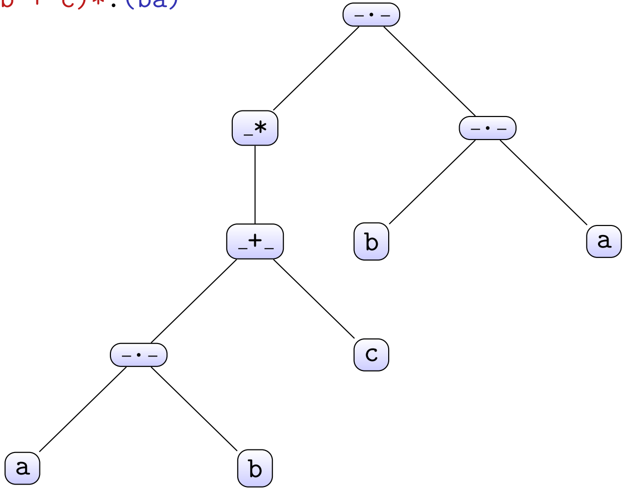
Definition AST 2/2

Exemplu: `(ab + c)*(ba)`



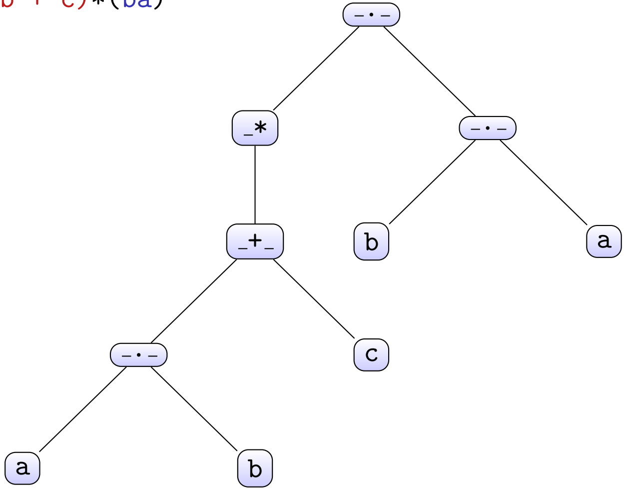
Definition AST 2/2

Exemplu: `(ab + c)*.(ba)`



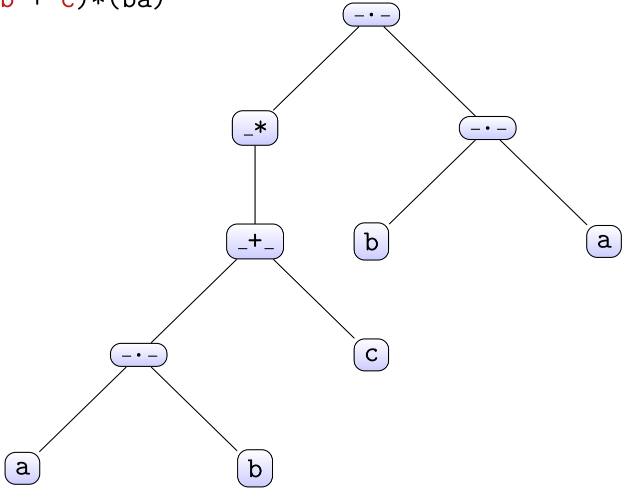
Definition AST 2/2

Exemplu: (ab + c)*(ba)



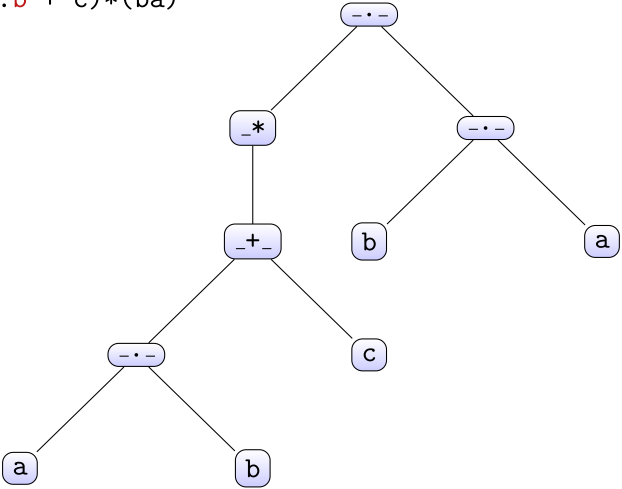
Definition AST 2/2

Exemplu: `(ab + c)*(ba)`



Definition AST 2/2

Exemplu: `(a.b + c)*(ba)`



Representation as Alk data structure

ast(e)

- *empty*: `[]`

Representation as Alk data structure

$ast(e)$

- *empty*: `[]`
- ε: `["", <>]`

Representation as Alk data structure

$ast(e)$

- *empty*: `[]`
- ε: `["", <>]`
- $a \in \Sigma$: `["a", <>]`

Representation as Alk data structure

$ast(e)$

- $empty$: []
- ε: ["", <>]
- $a \in \Sigma$: ["a", <>]
- $e_1 e_2$: ["_._", <$ast(e_1)$,$ast(e_2)$>]

Representation as Alk data structure

$ast(e)$

- $empty$: `[]`
- ε: `["", <>]`
- $a \in \Sigma$: `["a", <>]`
- $e_1 e_2$: `["_._", <ast(e1),ast(e2)>]`
- $e_1 + e_2$: `["_+_", <ast(e1),ast(e2)>]`

Representation as Alk data structure

ast(e)

- *empty*: []
- ε: ["", <>]
- $a \in \Sigma$: ["a", <>]
- $e_1 e_2$: ["_._", <$ast(e_1)$,$ast(e_2)$>]
- $e_1 + e_2$: ["_+_", <$ast(e_1)$,$ast(e_2)$>]
- $e*$: ["_*", <$ast(e)$>]

Representation as Alk data structure

$ast(e)$

- $empty$: `[]`
- ε: `["", <>]`
- $a \in \Sigma$: `["a", <>]`
- $e_1 e_2$: `["_._", <`$ast(e_1)$,$ast(e_2)$`>]`
- $e_1 + e_2$: `["_+_", <`$ast(e_1)$,$ast(e_2)$`>]`
- $e*$: `["_*", <`$ast(e)$`>]`

Example: `(ab + b)*(ba)`

```
["_._", <["_*", <["_+_", <["_._", <["a", <>], ["b", <>]>],
["b", <>]>]>], ["_._", <["b", <>], ["a", <>]>>]
```

An equivalent definition

```
expression ::= term ("+" term)*
term ::= maybeStar
       | maybeStar ("." maybeStar)*
maybeStar ::= factor ["*"] /* equiv to  factor | factor "*"  */
factor ::=
    Sigma
    | "(" expression ")"
```

An equivalent definition

```
expression ::= term ("+" term)*
term ::= maybeStar
       | maybeStar ("." maybeStar)*
maybeStar ::= factor ["*"] /* equiv to  factor | factor "*"  */
factor ::=
    Sigma
    | "(" expression ")"
```

Now the expression `(ab+c)*(ba)` is written as `(a.b+c)*.(b.a)` or as `(a.b+c)*.b.a`
(note the explicit representation fo the concatenation operator and that the parentheses expressing associativity are missing).

An equivalent definition

```
expression ::= term ("+" term)*
term ::= maybeStar
       | maybeStar ("." maybeStar)*
maybeStar ::= factor ["*"] /* equiv to  factor | factor "*"  */
factor ::=
    Sigma
    | "(" expression ")"
```

Now the expression `(ab+c)*(ba)` is written as `(a.b+c)*.(b.a)` or as `(a.b+c)*.b.a`
(note the explicit representation fo the concatenation operator and that the parentheses expressing associativity are missing).
NB A specification included in square brackets , [spec], is optional.

An equivalent definition

```
expression ::= term ("+" term)*
term ::= maybeStar
      | maybeStar ("." maybeStar)*
maybeStar ::= factor ["*"] /* equiv to  factor | factor "*"  */
factor ::=
    Sigma
    | "(" expression ")"
```

Now the expression `(ab+c)*(ba)` is written as `(a.b+c)*.(b.a)` or as `(a.b+c)*.b.a`
(note the explicit representation fo the concatenation operator and that the parentheses expressing associativity are missing).
NB A specification included in square brackets , [spec], is optional.

Exercise

Write the recursive definition described by the above BNF notation.

Some functions from AST domain 1/2

These methods define a kind of "domain specific language' for ASTs.

```
// number of children
chldNo(ast) {
  if (ast.size() > 0) return ast[1].size();
  return 0;
}
```

Some functions from AST domain 1/2

These methods define a kind of "domain specific language' for ASTs.

```
// number of children
chldNo(ast) {
  if (ast.size() > 0) return ast[1].size();
  return 0;
}


// the i-th child of an AST
chld(ast, i) {
  if (ast.size() > 0 && i < ast[1].size()) {
    return ast[1].at(i);
  }
}
```

`ast2string(ast)` - details in the file `ast.alk`

Some functions from AST domain 2/2

```
// updates a child
updatedChld(ast, i, newchld) {
  if (ast.size() > 0 && ast[1].size() > 0)
    if (i >= 0 && i < ast[1].size()) {
      ast[1].update(i, newchld);
      return ast;
    }
}
```

Some functions from AST domain 2/2

```
// updates a child
updatedChld(ast, i, newchld) {
  if (ast.size() > 0 && ast[1].size() > 0)
    if (i >= 0 && i < ast[1].size()) {
      ast[1].update(i, newchld);
      return ast;
    }
}

// removes a child
removedChld(ast, i) {
  if (ast.size() > 0 && ast[1].size() > 0)
    if (i >= 0 && i < ast[1].size()) {
      ast[1].removeAt(i);
      return ast;
    }
}
```

Parsing: helping functions 1/4

Global variables:

`input` - the expression given as input

`sigma` - the alphabet

`index` - the current position in the input

- display an error message

```
error(msg) modifies index {
  print(msg + " at position ");
  print(index);
  print("");
}
```

Parsing: helping functions 2/4

Functions that operate over input:

- curent symbol

```
sym() modifies index, input {
  if (index < input.size())
    return input.at(index);
  return "\0";
}
```

Parsing: helping functions 2/4

Functions that operate over input:

- curent symbol

```
sym() modifies index, input {
  if (index < input.size())
    return input.at(index);
  return "\0";
}
```

- next symbol

```
nextSym() modifies index, input {
  if (index < input.size()) {
    index++;
  } else
    error("nextsym: expected a symbol");
}
```

Parsing: helping functions 3/4

- tests if the current symbol is accepted

```
accept(s) {
    if (sym() == s) {
        nextSym();
        return true;
    }
    return false;
}
```

Parsing: helping functions 3/4

- tests if the current symbol is accepted

```
accept(s) {
    if (sym() == s) {
        nextSym();
        return true;
    }
    return false;
}
```

- tests if the current symbol is a character (an element in the alphabet)

```
acceptSigma() modifies sigma {
  for (i = 0; i < sigma.size(); ++i)
      if (accept(sigma[i])) {
        return true;
      }
  return false;
}
```

Parsing: helping functions 4/4

- tests if the current symbol is an expected one

```
expect(s) {
    if (accept(s))
        return true;
    error("expect: unexpected symbol");
    return false;
}
```

From the recursive definition to the algorithm 1/4

Definition:

```
factor ::=
    Sigma
    | "(" expression ")"
```

From the recursive definition to the algorithm 1/4

Definition:

```
factor ::=
    Sigma
    | "(" expression ")"
```

Algorithm:

```
factor() {
    s = sym();
    if (acceptSigma()) {
      return [s,<>];
    } else if (accept("(")) {
        ast = expression();
        expect(")");
    }
        return ast;
}
```

From the recursive definition to the algorithm 2/4

Definition:

```
maybeStar ::= factor ["*"] // equiv to factor | factor "*"
```

From the recursive definition to the algorithm 2/4

Definition:

```
maybeStar ::= factor ["*"] // equiv to factor | factor "*"
```

Algorithm:

```
maybeStar() {
    ast = factor();
    if (accept("*"))
      return ["_*", < ast >] ;
    else
      return ast;
}
```

From the recursive definition to the algorithm 3/4

Definition:

```
term ::= maybeStar
       | maybeStar ("." maybeStar)*
```

which is equivalent to

```
term ::= maybeStar ("." maybeStar)*
```

From the recursive definition to the algorithm 3/4

Definition:

```
term ::= maybeStar
       | maybeStar ("." maybeStar)*
```

which is equivalent to

```
term ::= maybeStar ("." maybeStar)*
```

Algorithm:

```
term() {
    ast = maybeStar();
    list = < ast >;
    while (accept(".")) {
        ast1 = maybeStar();
        list.pushBack(ast1);
    }
    if (list.size() > 1)
      return ["_._", list];
    else
      return list.at(0);
}
```

From the recursive definition to the algorithm 4/4

Definition:

```
expression ::= term ("+" term)*
```

From the recursive definition to the algorithm 4/4

Definition:

```
expression ::= term ("+" term)*
```

Algorithm:

```
expression() {
    ast = term();
    list = < ast >;
    while (accept("+")) {
        ast = term();
        list.pushBack(ast);
    }
    if (list.size() > 1)
      return ["_+_", list];
    else
      return list.at(0);
}
```

Test

```
// the alphabet
sigma = ["a","b","c"];
// the expression
input = "(a.b+c)*.(b.a)";
print(expression());
```

Test

```
// the alphabet
sigma = ["a","b","c"];
// the expression
input = "(a.b+c)*.(b.a)";
print(expression());

$ alki.sh -a parser.alk
[_._, <[_*, <[_+_, <[_._, <[a, <>], [b, <>]>], [c, <>]>]>],
       [_._, <[b, <>], [a, <>]>]>]
sigma |-> [a, b, c]
input |-> (a.b+c)*.(b.a)
index |-> 14
```

Plan

1. Regular Expressions
   - Recap
   - String Searching with Regular Expressions
   - Building the Abstract Syntax Tree, AST (parsing)
   - Language Defined by a Regular Expression
2. Automaton associated to a regular expression
   - Building the automaton
   - Searching Algorithm

Definition

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

Definition

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) =$

Definition

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$

Definition

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),

Definition

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\texttt{empty}) =$

Definition

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\texttt{empty}) = \emptyset$

Definition

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\mathtt{empty}) = \emptyset$
- if e is un character

Definition

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\texttt{empty}) = \emptyset$
- if e is un character then $L(e) = \{e\}$;

Definition

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\texttt{empty}) = \emptyset$
- if e is un character then $L(e) = \{e\}$;
- if $e = e_1 e_2$

Definition

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\texttt{empty}) = \emptyset$
- if e is un character then $L(e) = \{e\}$;
- if $e = e_1 e_2$ then $L(e) = L(e_1)L(e_2) = \{w_1 w_2 \mid w_1 \in L(e_1), w_2 \in L(e_2)\}$;

Definition

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\mathtt{empty}) = \emptyset$
- if e is un character then $L(e) = \{e\}$;
- if $e = e_1 e_2$ then $L(e) = L(e_1)L(e_2) = \{w_1 w_2 \mid w_1 \in L(e_1), w_2 \in L(e_2)\}$;
- if $e = e_1 + e_2$

Definition

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\mathtt{empty}) = \emptyset$
- if e is un character then $L(e) = \{e\}$;
- if $e = e_1 e_2$ then $L(e) = L(e_1)L(e_2) = \{w_1 w_2 \mid w_1 \in L(e_1), w_2 \in L(e_2)\}$;
- if $e = e_1 + e_2$ then $L(e) = L(e_1) \cup L(e_2)$;

Definition

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\texttt{empty}) = \emptyset$
- if e is un character then $L(e) = \{e\}$;
- if $e = e_1 e_2$ then $L(e) = L(e_1)L(e_2) = \{w_1 w_2 \mid w_1 \in L(e_1), w_2 \in L(e_2)\}$;
- if $e = e_1 + e_2$ then $L(e) = L(e_1) \cup L(e_2)$;
- if $e = e_1^*$

Definition

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\mathtt{empty}) = \emptyset$
- if e is un character then $L(e) = \{e\}$;
- if $e = e_1 e_2$ then $L(e) = L(e_1)L(e_2) = \{w_1 w_2 \mid w_1 \in L(e_1), w_2 \in L(e_2)\}$;
- if $e = e_1 + e_2$ then $L(e) = L(e_1) \cup L(e_2)$;
- if $e = e_1^*$ then $L(e) = \cup_k L(e_1^k)$, where $L(e_1^0) = \{\varepsilon\}, L(e_1^{k+1}) = L(e_1)L(e_1^k)$;

Definition

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\mathtt{empty}) = \emptyset$
- if e is un character then $L(e) = \{e\}$;
- if $e = e_1 e_2$ then $L(e) = L(e_1)L(e_2) = \{w_1 w_2 \mid w_1 \in L(e_1), w_2 \in L(e_2)\}$;
- if $e = e_1 + e_2$ then $L(e) = L(e_1) \cup L(e_2)$;
- if $e = e_1^*$ then $L(e) = \cup_k L(e_1^k)$, where $L(e_1^0) = \{\varepsilon\}, L(e_1^{k+1}) = L(e_1)L(e_1^k)$;
- if $e = (e_1)$

Definition

Definition

The set of strings (language) $L(e)$ described by a regular expression is recursively defined as follows:

- $L(\varepsilon) = \{\varepsilon\}$ (ε is the empty string (of size zero)),
- $L(\texttt{empty}) = \emptyset$
- if e is un character then $L(e) = \{e\}$;
- if $e = e_1 e_2$ then $L(e) = L(e_1)L(e_2) = \{w_1 w_2 \mid w_1 \in L(e_1), w_2 \in L(e_2)\}$;
- if $e = e_1 + e_2$ then $L(e) = L(e_1) \cup L(e_2)$;
- if $e = e_1^*$ then $L(e) = \cup_k L(e_1^k)$, where $L(e_1^0) = \{\varepsilon\}, L(e_1^{k+1}) = L(e_1)L(e_1^k)$;
- if $e = (e_1)$ then $L(e) = L(e_1)$.

Remark. The operator $\_^*$ is called Kleene star or Kleene closure.

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b + a)c) =$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b + a)c) = \{abc, aac\}$ and
$L((ab)^*) =$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b + a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b + a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b+a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$
$L(b) = \{b\}$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b + a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$
$L(b) = \{b\}$
$L(b + a) = L(b) \cup L(a) = \{b, a\}$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b+a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$
$L(b) = \{b\}$
$L(b+a) = L(b) \cup L(a) = \{b, a\}$
$L(a(b+a)) = L(a)L(b+a) = \{ab, aa\}$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b + a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$
$L(b) = \{b\}$
$L(b + a) = L(b) \cup L(a) = \{b, a\}$
$L(a(b + a)) = L(a)L(b + a) = \{ab, aa\}$
$L(c) = \{c\}$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b+a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$
$L(b) = \{b\}$
$L(b+a) = L(b) \cup L(a) = \{b, a\}$
$L(a(b+a)) = L(a)L(b+a) = \{ab, aa\}$
$L(c) = \{c\}$
$L(a(b+a)c) = L(a(b+a))L(c) = \{abc, aac\}$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b+a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$
$L(b) = \{b\}$
$L(b+a) = L(b) \cup L(a) = \{b, a\}$
$L(a(b+a)) = L(a)L(b+a) = \{ab, aa\}$
$L(c) = \{c\}$
$L(a(b+a)c) = L(a(b+a))L(c) = \{abc, aac\}$
$L((ab)^0) = \{\varepsilon\}$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b+a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$
$L(b) = \{b\}$
$L(b+a) = L(b) \cup L(a) = \{b, a\}$
$L(a(b+a)) = L(a)L(b+a) = \{ab, aa\}$
$L(c) = \{c\}$
$L(a(b+a)c) = L(a(b+a))L(c) = \{abc, aac\}$
$L((ab)^0) = \{\varepsilon\}$
$L((ab)^1) = L(ab) = \{ab\}$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b + a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$
$L(b) = \{b\}$
$L(b + a) = L(b) \cup L(a) = \{b, a\}$
$L(a(b + a)) = L(a)L(b + a) = \{ab, aa\}$
$L(c) = \{c\}$
$L(a(b + a)c) = L(a(b + a))L(c) = \{abc, aac\}$
$L((ab)^0) = \{\varepsilon\}$
$L((ab)^1) = L(ab) = \{ab\}$
$L((ab)^2) = L(ab)L(ab) = \{abab\}$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b + a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$
$L(b) = \{b\}$
$L(b + a) = L(b) \cup L(a) = \{b, a\}$
$L(a(b + a)) = L(a)L(b + a) = \{ab, aa\}$
$L(c) = \{c\}$
$L(a(b + a)c) = L(a(b + a))L(c) = \{abc, aac\}$
$L((ab)^0) = \{\varepsilon\}$
$L((ab)^1) = L(ab) = \{ab\}$
$L((ab)^2) = L(ab)L(ab) = \{abab\}$
$L((ab)^3) = L(ab)L((ab)^2) = \{ababab\}$

The language defined by a regular expression 2/2

Example: Consider $A = \{a, b, c\}$. We have
$L(a(b + a)c) = \{abc, aac\}$ and
$L((ab)^*) = \{\varepsilon, ab, abab, ababab, \ldots\} = \{(ab)^k \mid k \geq 0\}$.
Justification:
$L(a) = \{a\}$
$L(b) = \{b\}$
$L(b + a) = L(b) \cup L(a) = \{b, a\}$
$L(a(b + a)) = L(a)L(b + a) = \{ab, aa\}$
$L(c) = \{c\}$
$L(a(b + a)c) = L(a(b + a))L(c) = \{abc, aac\}$
$L((ab)^0) = \{\varepsilon\}$
$L((ab)^1) = L(ab) = \{ab\}$
$L((ab)^2) = L(ab)L(ab) = \{abab\}$
$L((ab)^3) = L(ab)L((ab)^2) = \{ababab\}$
...
$L((ab)^*) = \bigcup_{k \geq 0} L((ab)^k) = \{\varepsilon, ab, abab, ababab, \ldots\}$

Language Product

```
prod(L1, L2) {
  L = {};
  forall str1 in L1
    forall str2 in L2
        L = L U { str1 + str2 };
  return L;
}
```

Algorithm for a Bounded language 1/3

```
/*
  Returns the strings from L(ast), where Kleene iteration is
  applied k-times
*/
lang(ast, k) {
  ...
  return L;
}
```

- $L(\text{empty}) = \emptyset$ (L([]) = { })

```
    if (ast == []) L = {};
```

Algorithm for a Bounded language 1/3

```
/*
  Returns the strings from L(ast), where Kleene iteration is
  applied k-times
*/
lang(ast, k) {
  ...
  return L;
}
```

- $L(\texttt{empty}) = \emptyset$ (L([]) $= \{\,\}$)
```
    if (ast == []) L = {};
```
- $L(\varepsilon) = \{\varepsilon\}$ (L(["", <>]) $= \{""\}$)
- if e is a character then $L(e) = \{e\}$ (L(["a", <>]) $= \{"a"\}$)
```
    if (chldNo(ast) == 0)
      L = {root(ast)};
```

Algorithm for a Bounded language 2/3

- if $e = e_1 e_2 \ldots$ then
  $L(e) = L(e_1)L(e_2)\ldots = \{w_1 w_2 \ldots \mid w_1 \in L(e_1), w_2 \in L(e_2), \ldots\}$

```
if (root(ast) == "_._") {
  L = {""};
  for (i = 0; i < chldNo(ast); ++i)
    L = prod(L, lang(chld(ast, i),k));
}
```

Algorithm for a Bounded language 2/3

- if $e = e_1 e_2 \ldots$ then
  $L(e) = L(e_1)L(e_2)\ldots = \{w_1 w_2 \ldots \mid w_1 \in L(e_1), w_2 \in L(e_2), \ldots\}$

```
if (root(ast) == "_._") {
  L = {""};
  for (i = 0; i < chldNo(ast); ++i)
    L = prod(L, lang(chld(ast, i),k));
}
```

- if $e = e_1 + e_2 + \cdots$ then $L(e) = L(e_1) \cup L(e_2) \cup \cdots$

```
if (root(ast) == "_+_") {
  L = {};
  for (i = 0; i < chldNo(ast); ++i)
    L = L U lang(chld(ast, i),k);
}
```

Algorithm for a Bounded language 3/3

- if $e = e_1^*$ then $L(e) = \cup_i L(e_1^i)$, where $L(e_1^0) = \{\varepsilon\}, L(e_1^{i+1}) = L(e_1^i)L(e_1)$

```
if (root(ast) == "_*") {
  L = {""};
  Li = {""};
  L1 = lang(chld(ast, 0), k);
  for (i = 0; i < k; ++i) {
    Li = prod(Li, L1);
    L = L U Li;
  }
}
```

- if $e = (e_1)$ the $L(e) = L(e_1)$ - not needed

Assembling into an algorithm

```
lang(ast, k) {
  if (ast == []) L = {};
  else if (chldNo(ast) == 0)
    L = {ast[0]};
  else if (root(ast) == "_+_") {
    L = {};
    for (i = 0; i < chldNo(ast); ++i)
      L = L U lang(chld(ast, i),k);
    }
  else if (root(ast) == "_._") {
    L = {""};
    for (i = 0; i < chldNo(ast); ++i)
      L = prod(L, lang(chld(ast, i),k));
    }
  else if (root(ast) == "_*") {
    if (k == 0) L = {""};
    else {
      L1 = lang(ast, k-1);
      L  = L1 U prod(lang(chld(ast, 0), k), L1);
    }
  }
  else return "undefined";
  return L;
}
```

Test

```
// input = (a+b)*
print(lang(["_*", <["_+_", <["a", <>], ["b", <>]>]>], 3));
```

```
$ alki.sh -a lang.alk
{, a, aa, aaa, aab, ab, aba, abb, b, ba, baa, bab, bb, bba, bbb}
```

Plan

Definition 1/5

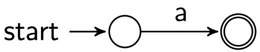
– base case

e is a character (a symbol) $a \in \Sigma$

Definition 1/5

– base case

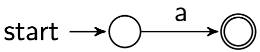
e is a character (a symbol) $a \in \Sigma$



Definition 1/5

– base case

e is a character (a symbol) $a \in \Sigma$

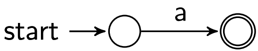


e is ε

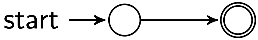
Definition 1/5

– base case

e is a character (a symbol) $a \in \Sigma$



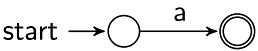
e is ε



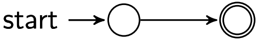
Definition 1/5

– base case

e is a character (a symbol) $a \in \Sigma$



e is ε

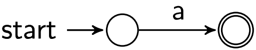


e is *empty*

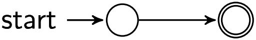
Definition 1/5

– base case

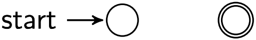
e is a character (a symbol) $a \in \Sigma$



e is ε

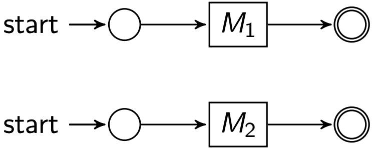


e is *empty*



Definition 2/5

for the recursive (inductive) case consider that e_1 and e_2 have the following corresponding automata:

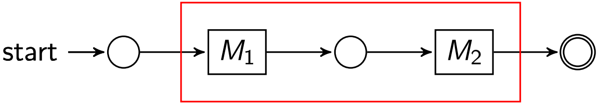


Definition 3/5

$e = e_1 e_2$:

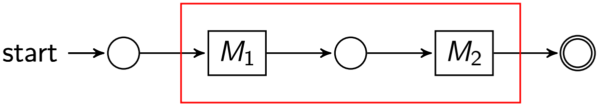
Definition 3/5

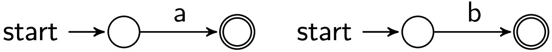
$e = e_1 e_2$:



Definition 3/5

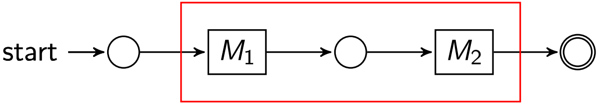
$e = e_1 e_2$:

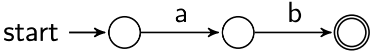




Definition 3/5

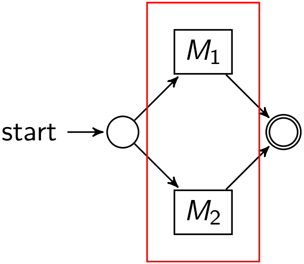
$e = e_1 e_2$:





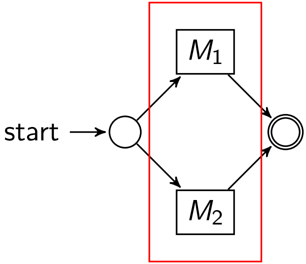
Definition 4/5

$e = e_1 + e_2$:



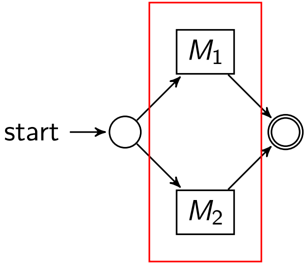
Definition 4/5

$e = e_1 + e_2$:

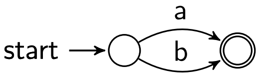


Definition 4/5

$e = e_1 + e_2$:



$a + b$:

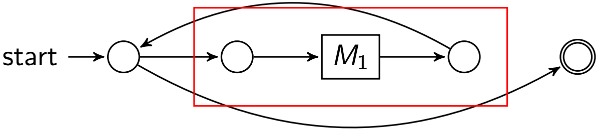


Definition 5/5

$e = e_1^*$:

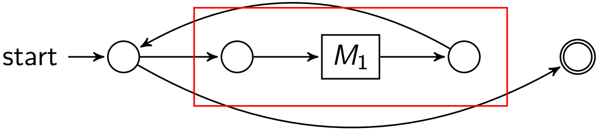
Definition 5/5

$e = e_1^*$:

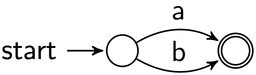


Definition 5/5

$e = e_1^*$:

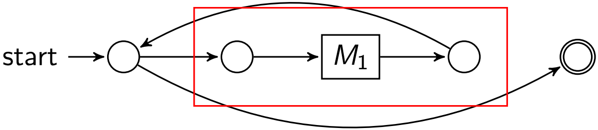


$a + b$:

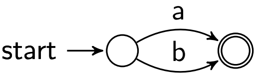


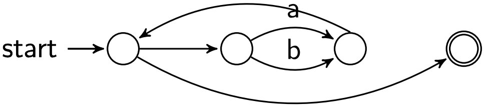
Definition 5/5

$e = e_1^*$:

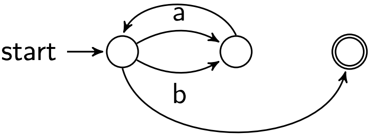


$a + b$:





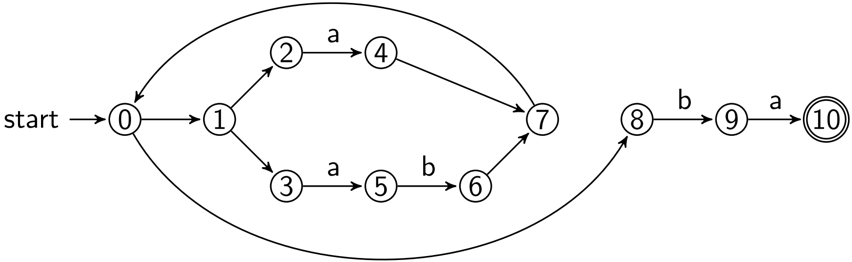
$(a + b)^*$



Example

$e = (a + ab)^* ba$

$M(e)$:



$L(e) = \{ba, aba, abba, aabba, ababa, \ldots\}$
Each string in $L(e)$ describe a route imn $M(e)$ from the initial state to the accepting state.

Nondeterministic automata

- The automata associated to regular expressions are particular cases of finite automata:

Nondeterministic automata

- The automata associated to regular expressions are particular cases of finite automata: $M = (Q, \Sigma, \delta, q_0, Q_f)$,

Nondeterministic automata

- The automata associated to regular expressions are particular cases of finite automata: $M = (Q, \Sigma, \delta, q_0, Q_f)$, where Q the set of states,

Nondeterministic automata

- The automata associated to regular expressions are particular cases of finite automata: $M = (Q, \Sigma, \delta, q_0, Q_f)$, where Q the set of states, Σ the alphabet,

Nondeterministic automata

- The automata associated to regular expressions are particular cases of finite automata: $M = (Q, \Sigma, \delta, q_0, Q_f)$, where Q the set of states, Σ the alphabet, $\delta \subset (Q \times A \times Q) \cup (Q \times Q)$ the transitions,

Nondeterministic automata

- The automata associated to regular expressions are particular cases of finite automata: $M = (Q, \Sigma, \delta, q_0, Q_f)$, where Q the set of states, Σ the alphabet, $\delta \subset (Q \times A \times Q) \cup (Q \times Q)$ the transitions, $q_0 \in Q$ initial state,

Nondeterministic automata

- The automata associated to regular expressions are particular cases of finite automata: $M = (Q, \Sigma, \delta, q_0, Q_f)$, where Q the set of states, Σ the alphabet, $\delta \subset (Q \times A \times Q) \cup (Q \times Q)$ the transitions, $q_0 \in Q$ initial state, $Q_f \subseteq Q$ accepting (final) states

Nondeterministic automata

- The automata associated to regular expressions are particular cases of finite automata: $M = (Q, \Sigma, \delta, q_0, Q_f)$, where Q the set of states, Σ the alphabet, $\delta \subset (Q \times A \times Q) \cup (Q \times Q)$ the transitions, $q_0 \in Q$ initial state, $Q_f \subseteq Q$ accepting (final) states
- the accepted language $L(M)$ is the set of strings describing routes from the initial state to an accepting state

Nondeterministic automata

- The automata associated to regular expressions are particular cases of finite automata: $M = (Q, \Sigma, \delta, q_0, Q_f)$, where Q the set of states, Σ the alphabet, $\delta \subset (Q \times A \times Q) \cup (Q \times Q)$ the transitions, $q_0 \in Q$ initial state, $Q_f \subseteq Q$ accepting (final) states
- the accepted language $L(M)$ is the set of strings describing routes from the initial state to an accepting state
- if $M(e)$ is the automaton associated to e, then $L(M(e)) = L(e)$

Nondeterministic automata

- The automata associated to regular expressions are particular cases of finite automata: $M = (Q, \Sigma, \delta, q_0, Q_f)$, where Q the set of states, Σ the alphabet, $\delta \subset (Q \times A \times Q) \cup (Q \times Q)$ the transitions, $q_0 \in Q$ initial state, $Q_f \subseteq Q$ accepting (final) states
- the accepted language $L(M)$ is the set of strings describing routes from the initial state to an accepting state
- if $M(e)$ is the automaton associated to e, then $L(M(e)) = L(e)$
- unlabeled transitions are called ε-transitions (internal transitions)

Nondeterministic automata

- The automata associated to regular expressions are particular cases of finite automata: $M = (Q, \Sigma, \delta, q_0, Q_f)$, where Q the set of states, Σ the alphabet, $\delta \subset (Q \times A \times Q) \cup (Q \times Q)$ the transitions, $q_0 \in Q$ initial state, $Q_f \subseteq Q$ accepting (final) states
- the accepted language $L(M)$ is the set of strings describing routes from the initial state to an accepting state
- if $M(e)$ is the automaton associated to e, then $L(M(e)) = L(e)$
- unlabeled transitions are called ε-transitions (internal transitions)
- the automaton built directly from definitions is nondeterministic (and not minimal)

Nondeterministic automata

- The automata associated to regular expressions are particular cases of finite automata: $M = (Q, \Sigma, \delta, q_0, Q_f)$, where Q the set of states, Σ the alphabet, $\delta \subset (Q \times A \times Q) \cup (Q \times Q)$ the transitions, $q_0 \in Q$ initial state, $Q_f \subseteq Q$ accepting (final) states
- the accepted language $L(M)$ is the set of strings describing routes from the initial state to an accepting state
- if $M(e)$ is the automaton associated to e, then $L(M(e)) = L(e)$
- unlabeled transitions are called ε-transitions (internal transitions)
- the automaton built directly from definitions is nondeterministic (and not minimal)
- not efficient in practice

Nondeterministic automata

- The automata associated to regular expressions are particular cases of finite automata: $M = (Q, \Sigma, \delta, q_0, Q_f)$, where Q the set of states, Σ the alphabet, $\delta \subset (Q \times A \times Q) \cup (Q \times Q)$ the transitions, $q_0 \in Q$ initial state, $Q_f \subseteq Q$ accepting (final) states
- the accepted language $L(M)$ is the set of strings describing routes from the initial state to an accepting state
- if $M(e)$ is the automaton associated to e, then $L(M(e)) = L(e)$
- unlabeled transitions are called ε-transitions (internal transitions)
- the automaton built directly from definitions is nondeterministic (and not minimal)
- not efficient in practice
- can we have an equivalent deterministic one?

Nondeterministic automata

- The automata associated to regular expressions are particular cases of finite automata: $M = (Q, \Sigma, \delta, q_0, Q_f)$, where Q the set of states, Σ the alphabet, $\delta \subset (Q \times A \times Q) \cup (Q \times Q)$ the transitions, $q_0 \in Q$ initial state, $Q_f \subseteq Q$ accepting (final) states
- the accepted language $L(M)$ is the set of strings describing routes from the initial state to an accepting state
- if $M(e)$ is the automaton associated to e, then $L(M(e)) = L(e)$
- unlabeled transitions are called ε-transitions (internal transitions)
- the automaton built directly from definitions is nondeterministic (and not minimal)
- not efficient in practice
- can we have an equivalent deterministic one?
- the answer is YES

Building an equivalent deterministic automaton - a general approach

Let N be an automaton with the states Q. The deterministic automaton D (where $\delta : Q \times \Sigma \to Q$) is defined as follows:

Building an equivalent deterministic automaton - a general approach

Let N be an automaton with the states Q. The deterministic automaton D (where $\delta : Q \times \Sigma \to Q$) is defined as follows:

- the set of states $\mathcal{P}(Q)$ (exponential number of states!!!)

Building an equivalent deterministic automaton - a general approach

Let N be an automaton with the states Q. The deterministic automaton D (where $\delta : Q \times \Sigma \to Q$) is defined as follows:

- the set of states $\mathcal{P}(Q)$ (exponential number of states!!!)
- there is a transition labeled by a from Q_1 to Q_2 iff $Q_2 = \bigcup\{q_2 \mid q_1 \xrightarrow{a} q_2, q_1 \in Q_1\}$

Building an equivalent deterministic automaton - a general approach

Let N be an automaton with the states Q. The deterministic automaton D (where $\delta : Q \times \Sigma \to Q$) is defined as follows:

- the set of states $\mathcal{P}(Q)$ (exponential number of states!!!)
- there is a transition labeled by a from Q_1 to Q_2 iff $Q_2 = \bigcup\{q_2 \mid q_1 \xrightarrow{a} q_2, q_1 \in Q_1\}$
- the initial state of D is $\{q_0\}$, where q_0 is the initial state of N

Building an equivalent deterministic automaton - a general approach

Let N be an automaton with the states Q. The deterministic automaton D (where $\delta : Q \times \Sigma \to Q$) is defined as follows:

- the set of states $\mathcal{P}(Q)$ (exponential number of states!!!)
- there is a transition labeled by a from Q_1 to Q_2 iff $Q_2 = \bigcup\{q_2 \mid q_1 \xrightarrow{a} q_2, q_1 \in Q_1\}$
- the initial state of D is $\{q_0\}$, where q_0 is the initial state of N
- a subset Q_f is accepting (final) if it includes an accepting state q_f of N

Building an equivalent deterministic automaton - a general approach

Let N be an automaton with the states Q. The deterministic automaton D (where $\delta : Q \times \Sigma \to Q$) is defined as follows:

- the set of states $\mathcal{P}(Q)$ (exponential number of states!!!)
- there is a transition labeled by a from Q_1 to Q_2 iff $Q_2 = \bigcup\{q_2 \mid q_1 \xrightarrow{a} q_2, q_1 \in Q_1\}$
- the initial state of D is $\{q_0\}$, where q_0 is the initial state of N
- a subset Q_f is accepting (final) if it includes an accepting state q_f of N

The above construction can be improved using Brzozowski derivatives.

Brzozowski derivatives

The derivatives of a regular expression (Brzozowski, 1964):

$$\delta_a(empty) = empty \qquad \varepsilon?(empty) = false$$

$$\delta_a(\varepsilon) = empty \qquad \varepsilon?(\varepsilon) = true$$

$$\delta_a(b) = \begin{cases} \varepsilon & , b = a \\ empty & , b \neq a \end{cases} \qquad \varepsilon?(b) = false$$

$$\delta_a(e_1 e_2) = \delta_a(e_1)e_2 + \varepsilon?(e_1)\delta_a(e_2) \qquad \varepsilon?(e_1 e_2) = \varepsilon?(e_1) \wedge \varepsilon?(e_2)$$

$$\delta_a(e_1 + e_2) = \delta_a(e_1) + \delta_a(e_2) \qquad \varepsilon?(e_1 + e_2) = \varepsilon?(e_1) \vee \varepsilon?(e_2)$$

$$\delta_a(e^*) = \delta_a(e)e^* \qquad \varepsilon?(e^*) = true$$

where $\varepsilon?(e_1)\delta_a(e_2)$ is a short notation for $\varepsilon?(e_1)\,?\,\delta_a(e_2) : empty$.

Brzozowski derivatives

The derivatives of a regular expression (Brzozowski, 1964):

$$\delta_a(empty) = empty \qquad \varepsilon?(empty) = false$$

$$\delta_a(\varepsilon) = empty \qquad \varepsilon?(\varepsilon) = true$$

$$\delta_a(b) = \begin{cases} \varepsilon & , b = a \\ empty & , b \neq a \end{cases} \qquad \varepsilon?(b) = false$$

$$\delta_a(e_1 e_2) = \delta_a(e_1)e_2 + \varepsilon?(e_1)\delta_a(e_2) \qquad \varepsilon?(e_1 e_2) = \varepsilon?(e_1) \wedge \varepsilon?(e_2)$$

$$\delta_a(e_1 + e_2) = \delta_a(e_1) + \delta_a(e_2) \qquad \varepsilon?(e_1 + e_2) = \varepsilon?(e_1) \vee \varepsilon?(e_2)$$

$$\delta_a(e^*) = \delta_a(e)e^* \qquad \varepsilon?(e^*) = true$$

where $\varepsilon?(e_1)\delta_a(e_2)$ is a short notation for $\varepsilon?(e_1)\,?\,\delta_a(e_2) : empty$.

Semantics: $L(\delta_a(e)) = \{w \mid aw \in L(e)\}$
$\varepsilon?(e) = true$ iff $\varepsilon \in L(e)$

Brzozowski derivatives

The derivatives of a regular expression (Brzozowski, 1964):

$$
\begin{aligned}
&\delta_a(empty) = empty && \varepsilon?(empty) = false \\
&\delta_a(\varepsilon) = empty && \varepsilon?(\varepsilon) = true \\
&\delta_a(b) = \begin{cases} \varepsilon & , b = a \\ empty & , b \neq a \end{cases} && \varepsilon?(b) = false \\
&\delta_a(e_1 e_2) = \delta_a(e_1)e_2 + \varepsilon?(e_1)\delta_a(e_2) && \varepsilon?(e_1 e_2) = \varepsilon?(e_1) \wedge \varepsilon?(e_2) \\
&\delta_a(e_1 + e_2) = \delta_a(e_1) + \delta_a(e_2) && \varepsilon?(e_1 + e_2) = \varepsilon?(e_1) \vee \varepsilon?(e_2) \\
&\delta_a(e^*) = \delta_a(e)e^* && \varepsilon?(e^*) = true
\end{aligned}
$$

where $\varepsilon?(e_1)\delta_a(e_2)$ is a short notation for $\varepsilon?(e_1)\,?\,\delta_a(e_2) : empty$.

Semantics: $L(\delta_a(e)) = \{w \mid aw \in L(e)\}$
$\varepsilon?(e) = true$ iff $\varepsilon \in L(e)$

Extension to strings: $\delta_\varepsilon(e) = e$, $\delta_{wa}(e) = \delta_a(\delta_w(e))$

A fundamental property

Theorem (Brzozowski)

The se of Brzozowski derivatives $\{\delta_w(e) \mid w \in A^*\}$*is finite.*

Example:
$\{\delta_w(\texttt{(ab + b)}^* \texttt{ b a}) \mid w \in A^*\} =$
$\{\texttt{(a b + b)}^* \texttt{ b a}, \texttt{b (a b + b)}^* \texttt{ b a}, (\texttt{(a b + b)}^* \texttt{ b a + a}),$
$(\texttt{b (a b + b)}^* \texttt{ b a} + \varepsilon\}$

the derivatives can be used to define the states of the automaton!

Computing ε? 1/2

- $\varepsilon?(empty) = false$

```
if (ast == []) return false;
```

- $\varepsilon?(\varepsilon) = true$

```
if (root(ast) == "")  // eps
  return true;
```

- $\varepsilon?(e_1 + e_2 + \cdots) = \varepsilon?(e_1) \vee \varepsilon?(e_2) \vee \cdots$

```
if (root(ast) == "_+_") {
  answ = epsIn(chld(ast, 0));
  for (i = 1; i < chldNo(ast); ++i)
    asw = answ || epsIn(chld(ast, i));
  return answ;
}
```

Computing ε? 2/2

- $\varepsilon?(e_1 e_2 \ldots) = \varepsilon?(e_1) \wedge \varepsilon?(e_2) \wedge \cdots$

```
if (root(ast) == "_._") {
  answ = epsIn(chld(ast, 0));
  for (i = 1; i < chldNo(ast); ++i)
    asw = answ && epsIn(chld(ast, i));
  return answ;
}
```

- $\varepsilon?(e^*) = \mathit{true}$

```
if (root(ast) == "_*")
  return true;
return false;
```

Assembling

```
epsIn(ast) {
  if (ast == []) return false;
  if (root(ast) == "")  // eps
    return true;
  if (root(ast) == "_+_") {
    answ = epsIn(chld(ast, 0));
    for (i = 1; i < chldNo(ast); ++i)
      asw = answ || epsIn(chld(ast, i));
    return answ;
  }
  if (root(ast) == "_._") {
    answ = epsIn(chld(ast, 0));
    for (i = 1; i < chldNo(ast); ++i)
      asw = answ && epsIn(chld(ast, i));
    return answ;
  }
  if (ast[0] == "_*")
    return true;
  return false;
}
```

Computing Brzozowski derivatives 1/3

- $\delta_a(empty) = empty$

```
if (ast == []) astder = [];
```

Computing Brzozowski derivatives 1/3

- $\delta_a(empty) = empty$

```
if (ast == []) astder = [];
```

- $\delta_a(\varepsilon) = empty$, $\delta_a(b) = \begin{cases} \varepsilon & , b = a \\ empty & , b \neq a \end{cases}$

```
if (chldNo(ast) == 0) {
  if (root(ast) == a) astder = ["", <>];
  else astder = [];
}
```

Computing Brzozowski derivatives 2/3

- $\delta_a(e_1 e_2 e_3 \ldots) = \delta_a(e_1)e_2 e_3 \ldots + \varepsilon?(e_1)\delta_a(e_2)e_3 \ldots + (\varepsilon?(e_1)\varepsilon?(e_2))\delta_a(e_3) \ldots$

```
if (root(ast) == "_._") {
  epsInPref = true;
  chlds = < >;
  i = 0;
  while (epsInPref && i < chldNo(ast) - 1 ) {
     chldsi = < der(chld(ast,i), a) >;
      for (j = i + 1; j < chldNo(ast); ++j)
        chldsi.pushBack(chld(ast,j));
      chlds.pushBack(["_._", chldsi]);
      epsInPref = epsIn(chld(ast, i));
      i++;
  }
  if (chlds.size() == 1)
    astder = chlds.at(0);
  else
    astder = ["_+_", chlds];
}
```

Computing Brzozowski derivatives 3/3

- $\delta_a(e_1 + e_2 + \cdots) = \delta_a(e_1) + \delta_a(e_2) + \cdots$

```
if (root(ast) == "_+_") {
  chlds = <>;
  for (i = 0; i < chldNo(ast); ++i)
    chlds.pushBack(der(chld(ast,i), a));
  astder = ["_+_", chlds];
}
```

Computing Brzozowski derivatives 3/3

- $\delta_a(e_1 + e_2 + \cdots) = \delta_a(e_1) + \delta_a(e_2) + \cdots$

```
if (root(ast) == "_+_") {
  chlds = <>;
  for (i = 0; i < chldNo(ast); ++i)
    chlds.pushBack(der(chld(ast,i), a));
  astder = ["_+_", chlds];
}
```

- $\delta_a(e^*) = \delta_a(e)e^*$

```
if (ast[0] == "_*")
    astder = ["_._", < der(chld(ast, 0), a), ast >];
```

Assembling

```
der(ast, a) {
  if (ast == []) astder = [];
  else if (chldNo(ast) == 0) {
    if (root(ast) == a) astder = ["", <>];
    else astder = [];
  }
  else if (root(ast) == "_+_") {
    chlds = <>;
    for (i = 0; i < chldNo(ast); ++i)
      chlds.pushBack(der(chld(ast,i), a));
    astder = ["_+_", chlds];
  }
  else if (root(ast) == "_._") {
    chlds = <der(chld(ast,0), a)>;
    for (i = 1; i < chldNo(ast); ++i)
      chlds.pushBack(chld(ast,i));
    chlds = < ["_._", chlds] >;
    for (i = 0; i < chldNo(ast); ++i) {
      if (epsIn(chld(ast, i)) && i < chldNo(ast) - 1) {
        chlds2 = <der(chld(ast,i + 1), a)>;
        for (j = i + 2; j < chldNo(ast); ++j)
          chlds2.pushBack(chld(ast,j));
        chlds.pushBack(["_._", chlds2]);
      }
    }
    if (chlds.size() == 1)
      astder = chlds.at(0);
    else
      astder = ["_+_", chlds];
  }
  else if (ast[0] == "_*")
    astder = ["_._", < der(chld(ast, 0), a), ast >];
```

Test

```
print(ast2string(["_*", <["_+_", <["a", <>], ["b", <>]>]>]));
print(der(["_*", <["_+_", <["a", <>], ["b", <>]>]>], "a"));
print(ast2string(der(["_*", <["_+_", <["a", <>], ["b", <>]>]>], "a"

$ alki.sh detaut.alk
(a + b)*
[_._, <[_+_, <[, <>], []>], [_*, <[_+_, <[a, <>], [b, <>]>]>]>]
( + )(a + b)*
```

Simplification

concatenation $+$ is associative, $+$ is also commutative

$e + e = e$

$e + empty = empty + e = e$

$e\ empty = empty\ e = empty$

$e\,\varepsilon = \varepsilon\, e = e$

(partial) implementation in `simplify(ast)` (the file `detaut.alk`).

Test

```
// der((a+b)*, a)
print(der(["_*", <["_+_", <["a", <>], ["b", <>]>]>], "a"));
print(ast2string(der(["_*", <["_+_", <["a", <>], ["b", <>]>]>], "a")));
print(simplify(der(["_*", <["_+_", <["a", <>], ["b", <>]>]>], "a")));
print(ast2string(simplify(der(["_*", <["_+_", <["a", <>], ["b", <>]>]>], "a
```

```
$ alki.sh -a detaut.alk
[_._, <[_+_, <[, <>], []>], [_*, <[_+_, <[a, <>], [b, <>]>]>]>]
( + )((a + b))*
[_*, <[_+_, <[a, <>], [b, <>]>]>]
(a + b)*
```

Plan

Building the Brzozowski automaton 1/4

- the set of states = the set of derivatives
- there is a transition a from q_1 to q_2 iff q_1 corersponds to a derivative $\delta_w(e)$ and q_2 corresponds to the derivative $\delta_{wa}(e)$ for a $w \in A^*$;
- the initial state is $e = \delta_\varepsilon(e)$
- a state q is accepting iff it corresponds to a derivative $\delta_w(e)$ and $\varepsilon?(\delta_w(e)) = \texttt{true}$.

Building the Brzozowski automaton 2/4

The derivatives and the states are stored into a `map` of tuples `[state, ast-derivative, type]`.

- Initial:

```
state = 0;
ast = simplify(ast);
if (epsIn(ast))
  map = < [state, ast, "acc"] >;
else
  map = < [state, ast, ""] >;
derSet = { ast };
```

Building the Brzozowski automaton 3/4

- Computing the derivatives associated to a state:

```
do {
  derSet1 = derSet;
  forall s in Sigma
    forall ast in derSet1 {
      ast1 = simplify(der(ast, s));
      if (!(ast1 in derSet) && ast1 != []) {
        derSet = derSet U { ast1 };
        state++;
        if (epsIn(ast1))
          map.pushBack([state, ast1,"acc"]);
        else
          map.pushBack([state, ast1, ""]);
      }
    }
} while (derSet != derSet1);
```

Building the Brzozowski automaton 4/4

- Building the automaton:

```
aut = {};
forall p in map
  forall q in map
    forall s in Sigma
      if (q[1] == simplify(der(p[1], s)))
        aut = aut U { < p[0], s, q[0] > };
forall p in map
  p[1] = ast2string(p[1]);
return [aut, map];
```

Test

Parsing the input:

```
input = "(a.b+b)*.(b.a)";
print(expression());

$ alki.sh -a parser.alk
[_._, <[_*, <[_+_, <[_._, <[a, <>], [b, <>]>], [b, <>]>]>], [_._, <[b, <>], [a, <>]
```

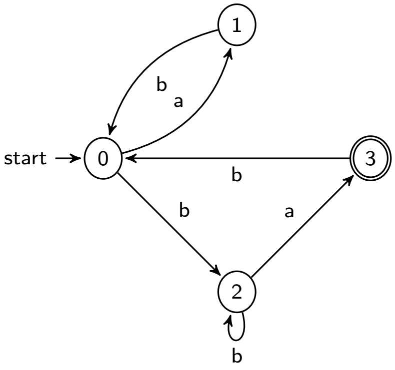
The automaton:

```
am = detAut(["_._", <["_*", <["_+_", <["_._", <["a", <>], ["b", <>]>], ["b", <>]>]>
          ["b", <>], ["a", <>]>], ["a", "b", "c"]);
print(am[0]);
forall p in am[1]
  print(p);

$ alki.sh -a detaut.alk
{<0, a, 1>, <0, b, 2>, <1, b, 0>, <2, b, 2>, <2, a, 3>, <3, b, 0>}
[0, (ab + b)*ba]
[1, b(ab + b)*ba]
[2, ((ab + b)*ba + a)]
[3, (b(ab + b)*ba + ), acc]
```

Example

$e = (\texttt{a b + b})^* \texttt{ b a}$



A bit of history

- the construction of the nondeterministic automaton given here is similar to that of Thompson în 1968
- the remove of internal states can given in quadratic time (Aho, Ullman, 1979)
- use of functions `first` and `follow` (Berry, Setti, 1986)
- parallelization (Myer, A Four Russians Algorithm for Regular Expression Pattern Matching, 1992)
- a good construction of the nondeterministic automaton is Glushkov-McNaughton-Yamada (1960-1961), which can be parallelised (Navarro & Raffinot, 2004)
- Berry and Setti found the natural relationship between Glushkov and the derivatives

More details at the course LFAC from the second year.

Plan

1. Regular Expressions
   - Recap
   - String Searching with Regular Expressions
   - Building the Abstract Syntax Tree, AST (parsing)
   - Language Defined by a Regular Expression

2. Automaton associated to a regular expression
   - Building the automaton
   - Searching Algorithm

Using the automaton in the searching process 1/4

- a very simple solution
- the goal is to show how the automaton is used
- the searching algorithm is obtained by a slight modification of the naive algorithm

Using the automaton in the searching process 2/4

- a helping function that tests if the symbol from the current position is expected by the automaton:

```
expected(am, state, a) {
  aut = am[0];
  map = am[1];
  forall trans in aut {
    if (trans.at(0) == state && trans.at(1) == a) { // expected
      newState = trans.at(2);
      return [newState, map.at(newState)[2]];
    }
  }
  return [-1];
}
```

Exercise

The above implementation is very inefficient. Find a suitable representation of the automaton such that the test is given in $O(1)$ time.

Using the automaton in the searching process 3/4

- modification of the function that test if a string defined by the pattern occurs at the position *i*:

```
occAtPos(s, am,  i) {
  n = s.size();
  aut = am[0];
  state = 0; // the current state is the initial state
  for (j = 0; true; ++j) {
    exp = expected (am, state, s[i+j]);
    if (i + j < n && exp[0] >= 0) { // expected
      state = exp[0];
      if (exp[1] == "acc") // accepted
        return true;
      // else: expected but not accepted
    }
    else
      return false;  // not expected
  }
}
```

Using the automaton in the searching process 4/4

- modification of the function that search for the first occurrence of a string defined by the pattern:

```
firstOcc(s, am)
{
  n = s.size();
  for (i = 0; i < n; ++i) {
    if (occAtPos(s, am, i)) {
      return i;
    }
  }
  return -1;
}
```

Test

Assume that am is the automaton from the previous example (forl "(a.b+b)*.(b.a)").

```
subj = "cbbaaabbacc";
s = subj.split();
print(firstOcc(s, am));
```

```
$ alki.sh -a detaut.alk
1
```

Complexity of string searching with regular expressions

Assume that the length of the expression is m (the number of characters) and $m_\Sigma = |\Sigma \cup \{\cdot, +, *\}|$.

Theorem (Kleene, 1956)

Searching with regular expressions can be solved in $O(n + 2^{m_\Sigma})$ time with deterministic automata and space $O(2^{m_\Sigma})$.

Theorem (Thomson, 1968)

Searching with regular expressions can be solved in $O(mn)$ time with nondeterministic automata and space $O(m)$.

Theorem (Myers, 1992)

Searching with regular expressions can be solved in $O(\frac{mn}{\log n} + (n+m)\log n)$ time deterministic automata and space $O(mn/\log n)$.

Ph. Bille and M. Thorup (2009) decrease the limit to $O(\frac{mn}{(\log n)^{3/2}} + n + m)$.