# PARSING WITH REGULAR EXPRESSIONS AND EXTENSIONS TO KLEENE ALGEBRA

Niels Bjørn Bugge Grathwohl
DIKU, November 4th 2015

PhD Thesis defense

# STRING REWRITING

```
1,John,john@gmail.com,male,123456,DK
2,Benny,benny@hotmail.com,male,98234,UK
```

$\rightarrow$

```
John 123456
Benny 98234
```

Want:

- *Streaming* – i.e., output while reading input.
- *Fast* – several Gbps throughput per core.
- Linear running time in the size of the input.

```
main := (row /\n/)*
col  := /[^,\n]*/
row  := ~(col /,/) col "\t" ~/,/ ~(col /,/)
        ~(col /,/) col ~/,/        ~col
```

Program is essentially a *regular expression* with outputs.

## Regular expression syntax

$$E ::= 0 \mid 1 \mid a \mid E_1 + E_2 \mid E_1 E_2 \mid E_1^\star$$

$(a \in \Sigma)$

## Examples

$(\Sigma = \{a, b\})$

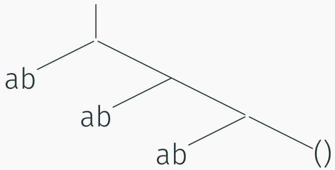
$$a$$
$$(ab)^\star + (a + b)^\star$$
$$(a + b)^\star$$

## WHAT IS REGULAR EXPRESSION “MATCHING”?

**Expression** $(ab)^\star + (a+b)^\star$

**Input** $s =$ **ababab**

- acceptance testing—is input string member of language?
  Answer: “Yes!”
- subgroup matching—substrings in input for subterms in expression.
  Answer: $[0,6],[4,2]$
- parsing—what is the parse tree of the input?



Input $s$ matches $E$ iff $s \in \mathcal{L}[\![E]\!]$.

## Language interpretation

$$
\begin{aligned}
\mathcal{L}[\![0]\!] &= \emptyset \\
\mathcal{L}[\![1]\!] &= \{\epsilon\} \\
\mathcal{L}[\![a]\!] &= \{a\} \\
\mathcal{L}[\![E + F]\!] &= \{s \mid s \in \mathcal{L}[\![E]\!]\} \\
&\cup \{t \mid t \in \mathcal{L}[\![F]\!]\} \\
\mathcal{L}[\![EF]\!] &= \{s \cdot t \mid s \in \mathcal{L}[\![E]\!], t \in \mathcal{L}[\![F]\!]\} \\
\mathcal{L}[\![E^\star]\!] &= \mathcal{L}[\![E]\!]^\star
\end{aligned}
$$

**Example**

$$
\begin{aligned}
& \mathcal{L}[\![(ab)^\star + (a+b)^\star]\!] \\
= \;& \mathcal{L}[\![(ab)^\star]\!] \cup \mathcal{L}[\![(a+b)^\star]\!] \\
= \;& \mathcal{L}[\![ab]\!]^\star \cup \mathcal{L}[\![a+b]\!]^\star \\
= \;& \{ab\}^\star \cup \{a, b\}^\star \\
= \;& \{\epsilon, ab, abab, \ldots\} \cup \{\epsilon, a, b, ab, ba, aba, \ldots\} \\
= \;& \{\epsilon, a, b, aa, ab, aaa, aab, \ldots\}
\end{aligned}
$$

Construct parse tree from input $s$ such that *flattening* of parse tree is $s$.

**Type interpretation [FC'04;HN'11]**

$$
\begin{array}{rcl}
\mathcal{T}[\![0]\!] & = & \emptyset \\
\mathcal{T}[\![1]\!] & = & \{()\} \\
\mathcal{T}[\![a]\!] & = & \{a\} \\
\mathcal{T}[\![E+F]\!] & = & \{\mathsf{inl}\ v \mid v \in \mathcal{T}[\![E]\!]\} \\
 & \cup & \{\mathsf{inr}\ w \mid w \in \mathcal{T}[\![F]\!]\} \\
\mathcal{T}[\![EF]\!] & = & \mathcal{T}[\![E]\!] \times \mathcal{T}[\![F]\!] \\
\mathcal{T}[\![E^\star]\!] & = & \{[v_1,\ldots,v_n] \mid n \geq 0, v_i \in \mathcal{T}[\![E]\!]\}
\end{array}
$$

Values in $\mathcal{T}[\![E]\!]$ are *parse trees*.

## Example

$\mathcal{T}[\![(ab)^\star + (a+b)^\star]\!]$ contains the parse trees:

- $\mathsf{inl}\ [(a,b),(a,b),(a,b)]$
- $\mathsf{inr}\ [\mathsf{inl}\ a, \mathsf{inr}\ b, \mathsf{inl}\ a, \mathsf{inr}\ b, \mathsf{inl}\ a, \mathsf{inr}\ b]$

which are *not* in $\mathcal{T}[\![(a+b)^\star]\!]$!

So

$$\mathcal{T}[\![(ab)^\star + (a+b)^\star]\!] \neq \mathcal{T}[\![(a+b)^\star]\!],$$

whereas

$$\mathcal{L}[\![(ab)^\star + (a+b)^\star]\!] = \mathcal{L}[\![(a+b)^\star]\!]$$

One input string can be parsed in multiple ways: `ababab` under $E = (ab)^\star + (a+b)^\star$ can be parsed *both* as

$$\text{inl } [(a, b), (a, b), (a, b)]$$
and
$$\text{inr } [\text{inl } a, \text{inr } b, \text{inl } a, \text{inr } b, \text{inl } a, \text{inr } b]$$

*Disambiguation policy*: the left-most option is always prioritized. “Greedy parsing.”

One input string can be parsed in multiple ways: ababab under $E = (ab)^\star + (a+b)^\star$ can be parsed *both* as

$$\text{inl}\ [(a,b),(a,b),(a,b)]$$
and
$$\text{inr}\ [\text{inl}\ a, \text{inr}\ b, \text{inl}\ a, \text{inr}\ b, \text{inl}\ a, \text{inr}\ b]$$

*Disambiguation policy*: the left-most option is always prioritized. "Greedy parsing."

Bit-coded parse trees: only store *choices*.

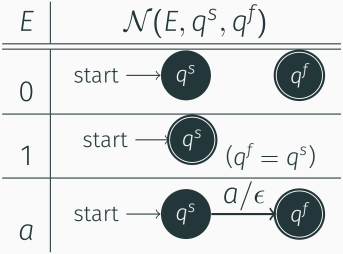
Parse tree as stream of bits; meaningless without expression!

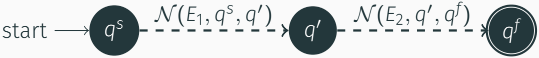
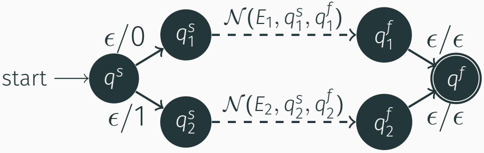
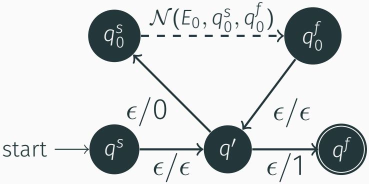
**Example**

$E = (ab)^\star + (a+b)^\star$, ababab:

| | |
|---|---|
| inl $[(a,b),(a,b),(a,b)]$ | 00001 |
| inr [inl $a$, inr $b$, inl $a$, inr $b$, inl $a$, inr $b$] | 10001000100011 |

- Thompsons FSTs with input alphabet $\Sigma$, output alphabet $\{0, 1\}$.
- Construction:



<table>
<tr><th>$E$</th><th>$\mathcal{N}(E, q^s, q^f)$</th></tr>
<tr><td>$E_1 E_2$</td><td>
</td></tr>
<tr><td>$E_1 + E_2$</td><td>
</td></tr>
<tr><td>$E_0^\star$</td><td>
</td></tr>
</table>

**Theorem (Brüggemann-Klein 1993, GHNR 2013)**

1-to-1 correspondence between

- parse trees for $E$,
- paths in Thompson FST for $E$,
- bit-coded parse trees.

Constructing the parse tree corresponds to finding a path through the FST.

## Optimally streaming parsing

Output the longest common prefix of possible parse trees after reading each input symbol.

## Example

$E = (aaa + aa)^\star$

Possible parse tree prefixes after **aaaa**:

$$\{01011, 000\ldots\}$$

Possible parse tree prefixes after **aaaaa**:

$$\{00011, 0000\ldots\}$$

| | Time | Space | Aux | Answer |
|---|---|---|---|---|
| Parse (3-p)[1] | $O(mn)$ | $O(m)$ | $O(n)$ | greedy parse |
| Parse (2-p)[2] | $O(mn)$ | $O(m)$ | $O(n)$ | greedy parse |
| Parse (str.)[3] | $O(mn + 2^{m \log m})$ | $O(m)$ | $O(n)$ | greedy parse |

($n$ size of input, $m$ size of expression)

[1] Frisch, Cardelli (2004)

[2] Grathwohl, Henglein, Nielsen, Rasmussen (2013)

[3] Grathwohl, Henglein, Rasmussen (2014)

| | Time | Space | Aux | Answer |
|---|---|---|---|---|
| Parse (3-p)[1] | $O(mn)$ | $O(m)$ | $O(n)$ | greedy parse |
| Parse (2-p)[2] | $O(mn)$ | $O(m)$ | $O(n)$ | greedy parse |
| Parse (str.)[3] | $O(mn + 2^{m \log m})$ | $O(m)$ | $O(n)$ | greedy parse |

($n$ size of input, $m$ size of expression)

[1] Frisch, Cardelli (2004)
[2] Grathwohl, Henglein, Nielsen, Rasmussen (2013)
[3] Grathwohl, Henglein, Rasmussen (2014)

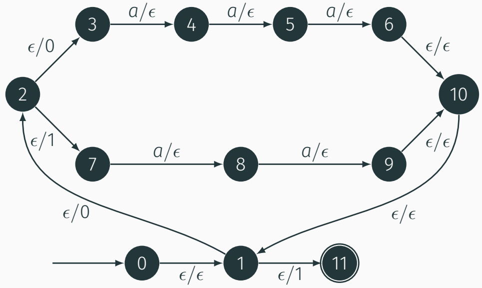
## Optimally streaming algorithm

- Preprocessing step of FST: compute *coverage* of state sets.
- Maintain a *path tree* during FST simulation, recording the path taken to each state in the FST.
    - Prune states that are covered by higher-prioritized states.
- Output on the stem of the path tree is longest common prefix of any succeeding parse.

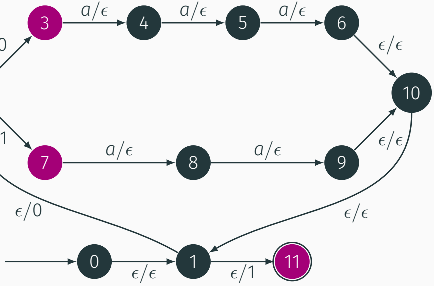
## Theorem (GHR'14)

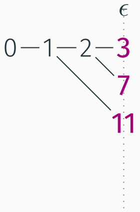
Optimally streaming algorithm computes the optimally streaming parsing function in time $O(mn + 2^{m \log m})$.

# PATH TREE EXAMPLE: $(aaa + aa)^\star$

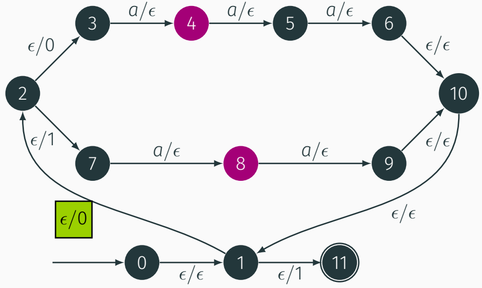


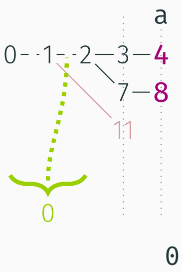
# PATH TREE EXAMPLE: $(aaa + aa)^\star$



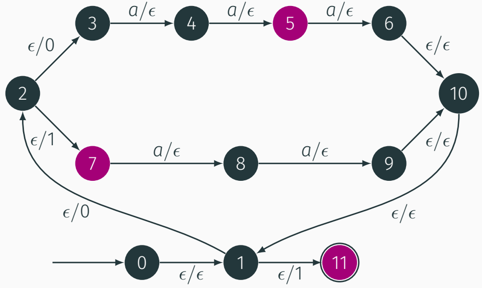



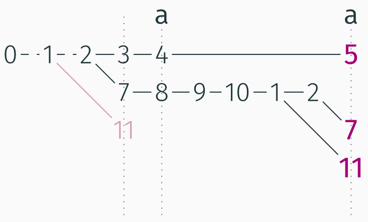
# PATH TREE EXAMPLE: $(aaa + aa)^{\star}$



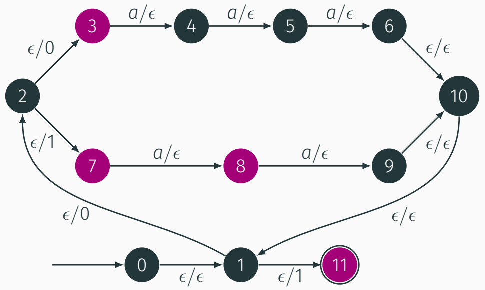
# PATH TREE EXAMPLE: $(aaa + aa)^{\star}$

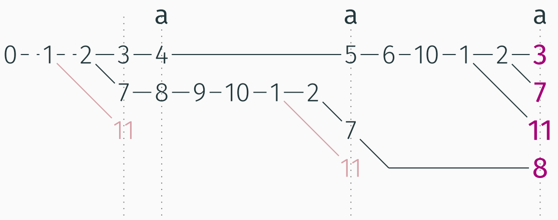



0

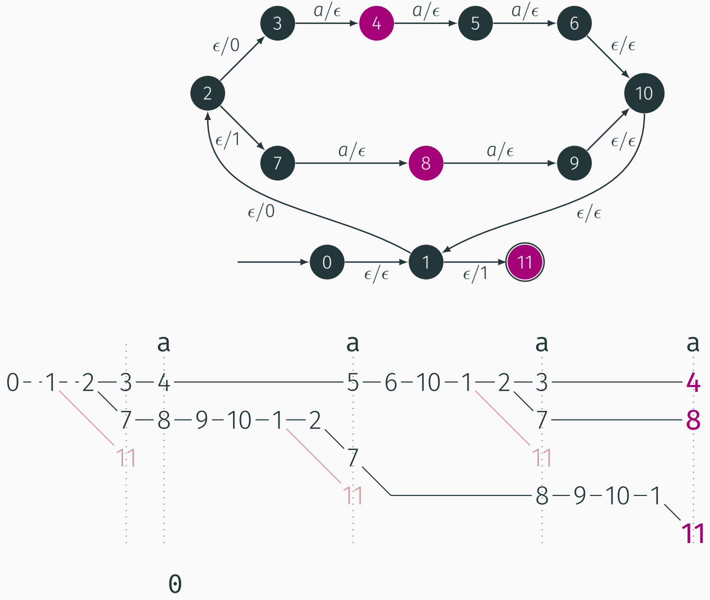
# PATH TREE EXAMPLE: $(aaa + aa)^\star$



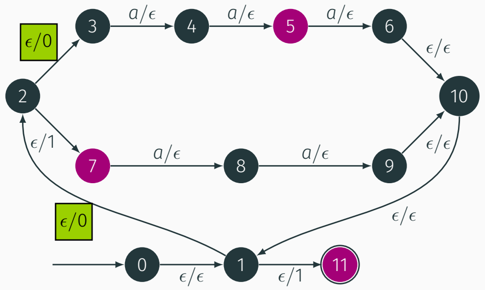



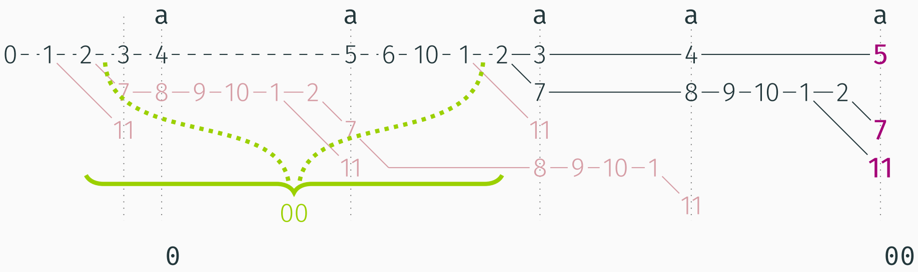

0

# PATH TREE EXAMPLE: $(aaa + aa)^{\star}$



# PATH TREE EXAMPLE: $(aaa + aa)^\star$



**Observation**

Approach is not limited to Thompson FSTs outputting bit-coded parse trees.

Kleenex is a surface language for specifying FSTs and their output:

- *grammar* with greedy disambiguation;
- embedded *output actions*.

- Essentially optimally streaming behaviour.
- Linear running time in size of input string.
- *Fast.* >1 Gbps common.

```
main  := (num /\n/)*
num   := digit{1,3} ("," digit{3})*
digit := /[0-9]/
```

”100000000000” → ”100,000,000,000”

- Problem: need to read entire number; no bounded lookahead!
- But: each newline ends a number, so output.
- Optimal streaming gives this for free!

Path tree algorithm is “NFA simulation with path trees as state sets.”

*Compilation* of FSTs? Analogy to NFA-DFA determinization with subset construction?

Path tree algorithm is “NFA simulation with path trees as state sets.”

*Compilation* of FSTs? Analogy to NFA-DFA determinization with subset construction?

Problem: Inifinite number of path trees!

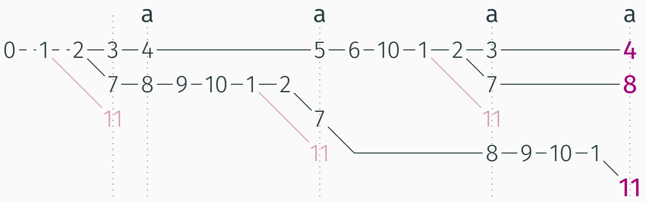
Path tree algorithm is “NFA simulation with path trees as state sets.”

*Compilation* of FSTs? Analogy to NFA-DFA determinization with subset construction?

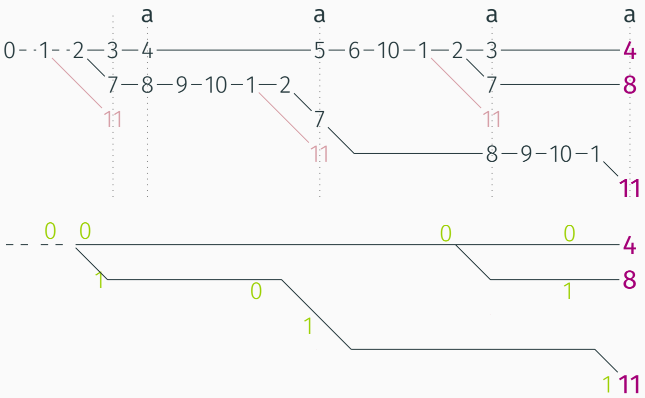
Problem: Inifinite number of path trees!

Solution: *contract* unary paths in path trees and store output in registers.

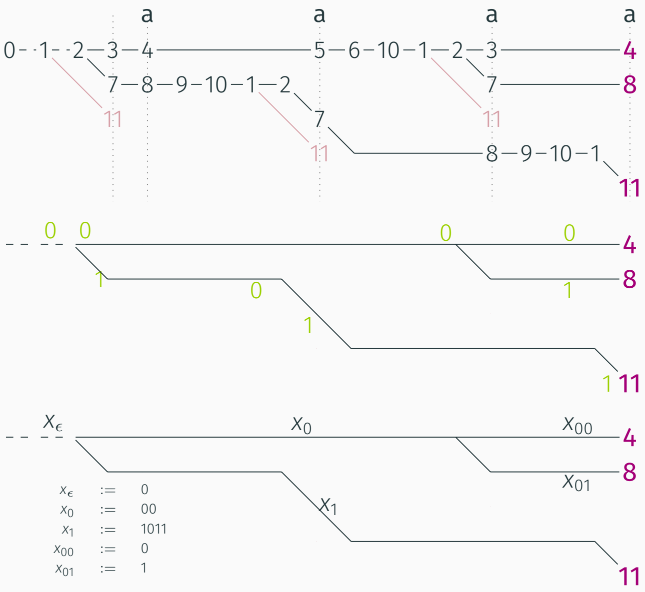
# DETERMINIZATION



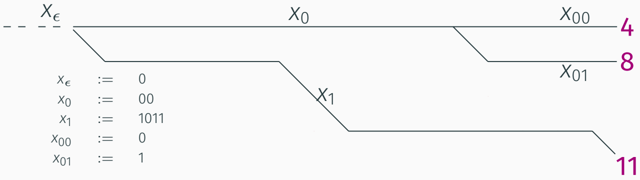
# DETERMINIZATION



# DETERMINIZATION

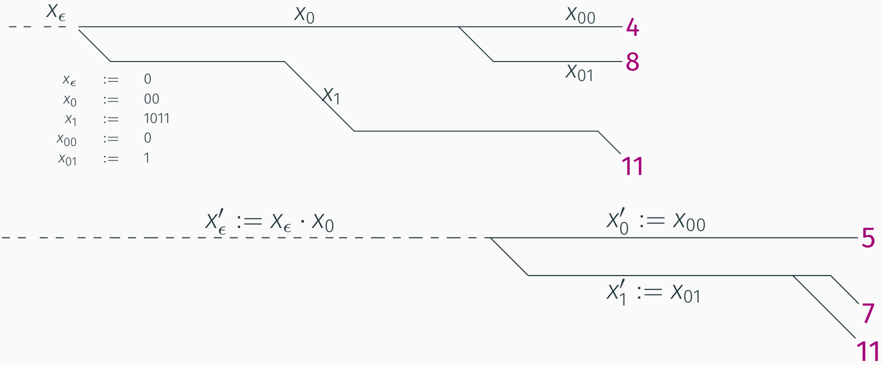


# DETERMINIZATION




$$
\begin{aligned}
x_\epsilon &:= 0\\
x_0 &:= 00\\
x_1 &:= 1011\\
x_{00} &:= 0\\
x_{01} &:= 1
\end{aligned}
$$

# DETERMINIZATION



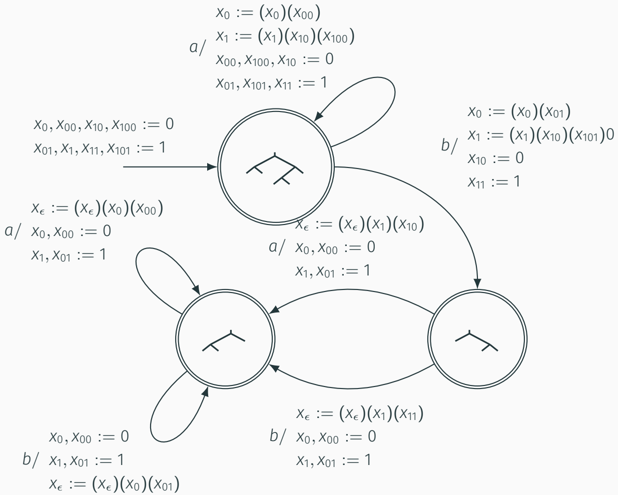
- Streaming string transducer:
  - deterministic finite automata,
  - each state equipped with fixed number of registers containing strings
  - registers updated on transititon by affine function;
  - Alur, D'Antoni, Raghothaman (2015).

- Streaming string transducer:
  - deterministic finite automata,
  - each state equipped with fixed number of registers containing strings
  - registers updated on transititon by affine function;
  - Alur, D'Antoni, Raghothaman (2015).

**Theorem**

FSTs with greedy order semantics correspond to SSTs.

- States are contracted path trees.
- Edges in contracted path trees $\cong$ registers in SST.


$a/$ $x_0 := (x_0)(x_{00})$
$x_1 := (x_1)(x_{10})(x_{100})$
$x_{00}, x_{100}, x_{10} := 0$
$x_{01}, x_{101}, x_{11} := 1$
$x_0, x_{00}, x_{10}, x_{100} := 0$
$x_{01}, x_1, x_{11}, x_{101} := 1$
$b/$ $x_0 := (x_0)(x_{01})$
$x_1 := (x_1)(x_{10})(x_{101})0$
$x_{10} := 0$
$x_{11} := 1$
$a/$ $x_\epsilon := (x_\epsilon)(x_0)(x_{00})$
$x_0, x_{00} := 0$
$x_1, x_{01} := 1$
$a/$ $x_\epsilon := (x_\epsilon)(x_1)(x_{10})$
$x_0, x_{00} := 0$
$x_1, x_{01} := 1$
$b/$ $x_0, x_{00} := 0$
$x_1, x_{01} := 1$
$x_\epsilon := (x_\epsilon)(x_0)(x_{01})$
$b/$ $x_\epsilon := (x_\epsilon)(x_1)(x_{11})$
$x_0, x_{00} := 0$
$x_1, x_{01} := 1$

**Haskell implementation**

Kleenex source $\rightarrow$ FST $\rightarrow$ SST $\rightarrow$ C

C code compiled with GCC/clang

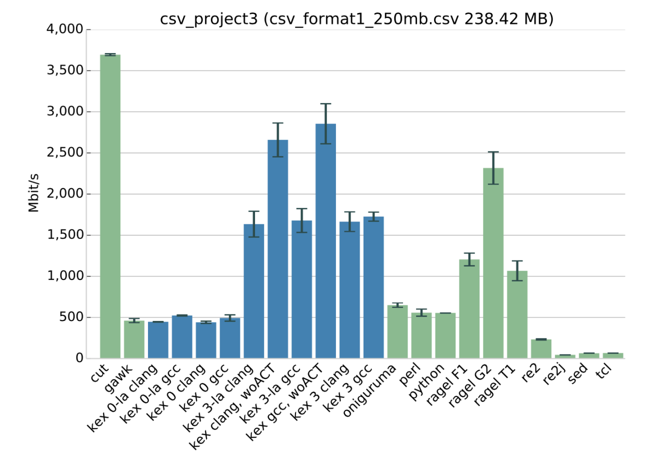
Performance comparison with regular expression libraries:

- AWK, Perl, Python, Sed, Tcl
- RE2/RE2j
- Ragel state machine compiler

`https://github.com/diku-kmc/kleenexlang`

# PERFORMANCE

```
main := (row /\n/)*
col  := /[^,\n]*/
row  := ~(col /,/) col "\t" ~/,/ ~(col /,/)
        ~(col /,/) col ~/,/       ~col
```



# FUTURE WORK

- Program fragments as output actions
- Memoization techniques à la NFA/DFA memoization in RE2.
- Applications – bioinformatics, finance, log digging, ....;
- Parallel processing: read >8 bits in parallel;
- Approximate matching — necessary in biological applications;
- Expressiveness, visibly pushdown automata;
- Automatically generate interfaces for various programming languages.

# Kleene algebra

**Kleene algebra**

A structure $(K, +, \cdot, ^\star, 0, 1)$:

- A set of elements $K$,
- binary operators $+$ and $\cdot$,
- unary operator $^\star$,
- special elements 0 and 1,

that satisfies the *Kleene algebra axioms*.

## Semiring

$$x \cdot (y \cdot z) = (x \cdot y) \cdot z \qquad x + (y + z) = (x + y) + z$$
$$1 \cdot x = x = x \cdot 1 \qquad 0 + x = x = x + 0$$
$$x + y = y + x$$

$$x \cdot (y + z) = x \cdot y + x \cdot z$$
$$(x + y) \cdot z = x \cdot z + y \cdot z$$
$$0 \cdot x = 0 = x \cdot 0$$

- idempotence: $x + x = x$
- partial order: $x \leq y \iff x + y = y$

## Kleene algebra

Idempotent semiring with $\star$ operator:

$$
\begin{array}{ll}
1 + x \cdot x^\star \leq x^\star & \quad b + a \cdot x \leq x \implies a^\star \cdot b \leq x \\
1 + x^\star \cdot x \leq x^\star & \quad b + x \cdot a \leq x \implies b \cdot a^\star \leq x
\end{array}
$$

Any structure with these operators that satisfies the axioms is a Kleene algebra.

- Languages: $(L, \cup, \cdot, ^\star, \emptyset, \{\epsilon\})$.
  - $L$ is set of strings over an alphabet,
  - $\cup$ is set union,
  - $\cdot$ is string concatenation,
  - $^\star$ is repetition of strings,
  - partial order $\leq$ is subset inclusion $\subseteq$.

  *Language interpretation* of regular expressions from before.
- Relation model, tropical semiring, ...

"Regular expressions:" syntax to describe elements in a Kleene algebra.

## Canonical interpretation

The *canonical interpretation* of a term $E$ is the regular language interpretation:

$$\begin{aligned} L_\Sigma(x) &= \{x\} & L_\Sigma(e_0 + e_1) &= L_\Sigma(e_0) \cup L_\Sigma(e_1) \\ L_\Sigma(0) &= \emptyset & L_\Sigma(e_0 e_1) &= \{vw \mid v \in L_\Sigma(e_0),\ w \in L_\Sigma(e_1)\} \\ L_\Sigma(1) &= \{\epsilon\} & L_\Sigma(e^\star) &= \bigcup_{n \geq 0} L_\Sigma(e^n). \end{aligned}$$

## Polynomials

Given idempotent semiring $C$ and a set of variables $X$, form *polynomials* over $C$ and $X$:

$$0 \qquad a \qquad ax^2 + bxy^3 + 1 \qquad 1 + a + ax + by$$

## System of polynomial inequalities

$$\begin{aligned} 1 + aB + bA &\leq S \\ A + aS + bAA &\leq A \\ bS + aBB &\leq B \end{aligned}$$

*Solution*: valuation of variables in $X$ such that the inequalities are satisfied.

A semiring $C$ is *algebraically closed* if all finite systems of polynomials have *least* solutions.

**Definition**

A *Chomsky algebra* is a an algebraically closed idempotent semiring.

*Context-free* languages over symbols from $X$ are not Kleene algebras, but they are Chomsky algebras.

Context-free grammar corresponds to system of polynomial inequalities:

$$\begin{aligned} S &\to \epsilon \mid aB \mid bA \\ A &\to aS \mid bAA \\ B &\to bS \mid aBB \end{aligned} \qquad \begin{aligned} 1 + aB + bA &\leq S \\ aS + bAA &\leq A \\ bS + aBB &\leq B \end{aligned}$$

# $\mu$-TERMS

- Regular expressions: denote elements in Kleene algebra.
- $\mu$-terms: denote elements in Chomsky algebra.

## $\mu$-terms

$\mathsf{T}X$ are $\mu$-terms over an alphabet $X$:

$$t ::= 0 \mid 1 \mid x \mid t + t \mid t \cdot t \mid \mu x.t \qquad x \in X$$

## $n$-fold composition

$$0x.t \equiv 0 \qquad (n+1)x.t \equiv t[x/nx.t]$$

## Examples

$$0x.axb+1 = 0$$
$$1x.axb+1 = a(0x.axb+1)b+1 = a0b+1 = 1$$
$$2x.axb+1 = a(1x.axb+1)b+1 = ab+1$$

Given interpretation of literals: $\sigma : X \to C$, *interpretation* of $\mu$-terms over Chomsky algebra $C$.

Function $\sigma : \mathsf{T}X \to C$ where:

$$\begin{array}{rclrcl} \sigma(0) & = & 0 & \sigma(a+b) & = & \sigma(a)+\sigma(b) \\ \sigma(1) & = & 1 & \sigma(a \cdot b) & = & \sigma(a)\cdot\sigma(b) \end{array}$$

$$\sigma(\mu x.t) = \text{least } a \in C \text{ such that } \sigma[x/a](t) \leq a$$

## Canonical interpretation

Canonical interpretation as *context-free languages*:

$$
\begin{aligned}
L_X(x) &= \{x\} & L_X(t_0 + t_1) &= L_X(t_0) \cup L_X(t_1) \\
L_X(0) &= \emptyset & L_X(t_0 \cdot t_1) &= \{vw \mid v \in L_X(t_0),\ w \in L_X(t_1)\} \\
L_X(1) &= \{\epsilon\} & L_X(\mu x.t) &= \bigcup_{n \geq 0} L_X(nx.t).
\end{aligned}
$$

# $\mu$-CONTINUITY

$\sum_{n\geq 0} t_n$ denotes *supremum* with respect to partial order $\leq$

## $\mu$-continuity

A Chomsky algebra $C$ is *$\mu$-continuous* if

$$\sigma\,(a(\mu x.t)b) = \sum_{n\geq 0} \sigma\,(a(nx.t)n)$$

for any interpretation $\sigma$ over $C$.

Canonical interpretation as context-free language is $\mu$-continuous:

$$L_X(\mu x.t) = \bigcup_{n\geq 0} L_X(nx.t).$$

**Theorem**

The following are equivalent:

(i) $s = t$ holds in all $\mu$-continous Chomsky algebras,

(ii) $L_X(s) = L_X(t)$ holds in the canonical interpretation as a context-free language over variables $X$.

Two context free languages $L_X(s)$ and $L_X(t)$ are equivalent if and only if $s = t$ is provable from the axioms of $\mu$-continuous Chomsky algebra:

**Axioms**

$$
\begin{aligned}
x \cdot (y \cdot z) &= (x \cdot y) \cdot z & x + (y + z) &= (x + y) + z \\
1 \cdot x = x &= x \cdot 1 & 0 + x = x &= x + 0 \\
x \cdot (y + z) &= x \cdot y + x \cdot z & x + y &= y + x \\
(x + y) \cdot z &= x \cdot z + y \cdot z & x + x &= x \\
0 \cdot x = 0 &= x \cdot 0 & &
\end{aligned}
$$

$$a(\mu x.t)b = \sum_{n \geq 0} a(nx.t)b$$

$\mu$-continuity axiom is infinitary:

$$a(nx.t)b \leq a(\mu x.t)b, \qquad n \geq 0$$

$$\left(\bigwedge_{n \geq 0} (a(nx.t)b \leq w)\right) \implies a(\mu x.t)b \leq w$$

- Equivalence of context-free languages is undecidable.
- To use inference, one must establish infinitely many premises.

# SUMMARY, FURTHER DIRECTIONS

- Extend Chomsky algebra with test symbols, analogously to Kleene algebra with tests.
- Coalgebraic treatment of Chomsky algebra?
- Applications to program verification, like Kleene algebra?
- “Visibly pushdown” Chomsky algebra?
- KAT+B! is an extension to Kleene algebra with tests adding mutable state:
    - elements correspond to square matrices with regular language entries.
    - extend Kleenex with mutable state?

Thank you