A Partial Conversion of Parsing Expression Grammars to Deterministic Finite Automata

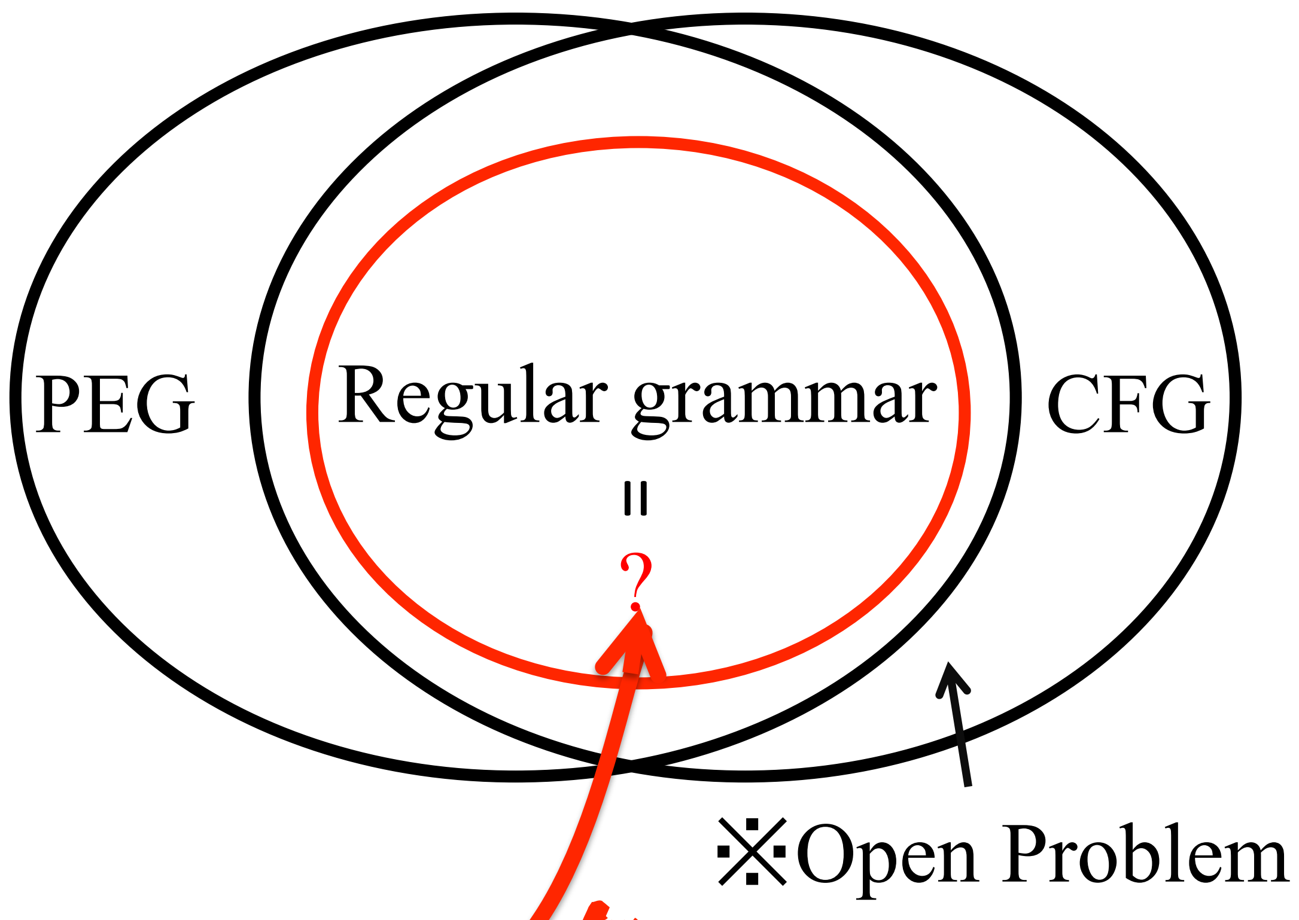
Nariyoshi Chida and Kimio Kuramitsu
Yokohama National University, Japan

Visit the website! → http://regex-and-pe-to-dfa.com

Background : Parsing Expression Grammars (PEGs)

PEGs are formal grammars introduced by Ford in 2004.

- Features of PEGs
  - Parsed in linear time using a memorizing parser called *packrat parser*
  - Easily implemented by recursive descent parsers
    - There is no need to implement a lexer (i.e. PEGs are scanner-less)
  - Deterministic
    - Behavior of each operators are deterministic (e.g. repetition operators are greedy)
  - Recognize languages that is not context-free such as $\{a^n b^n c^n | n \geq 0\}$



※Open Problem

proposal

What is a subclass of PEGs that is equivalent to DFAs?

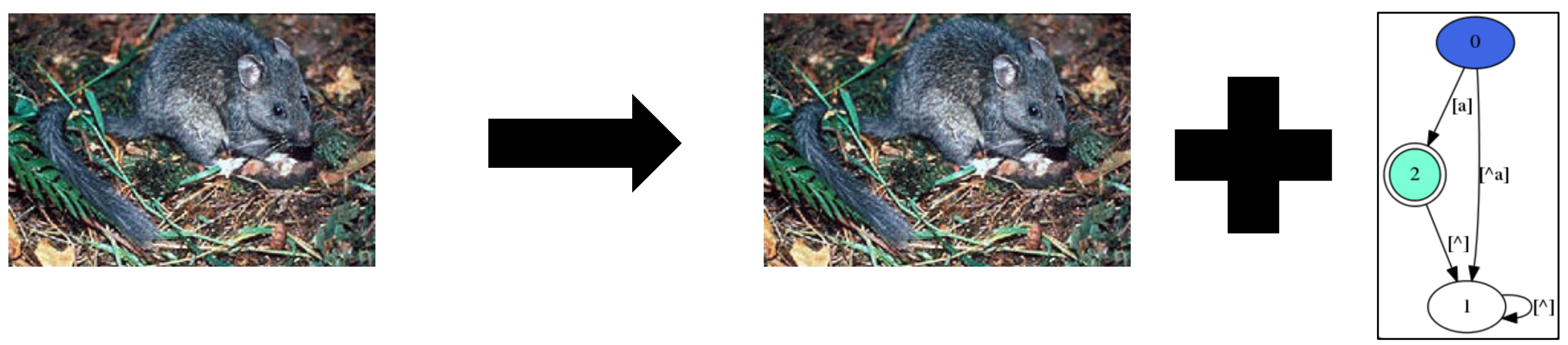
Since PEGs are relatively new, there are several unsolved problems.
One of the problems is that the revealing a subclass of PEGs that is equivalent to DFAs.

Motivation : Eliminating backtrackings

Backtracking arises when packrat parsers fail to match.

- Regular Grammar in PEGs
  - This allows to apply some techniques from the theory of RG to PEGs.
  - For example, the techniques of the theory of RG are used for faster parsing
    - A DFA optimization for PEG-based parsers

Improved!



We formalize the subclass as linear PEG (LPEG).

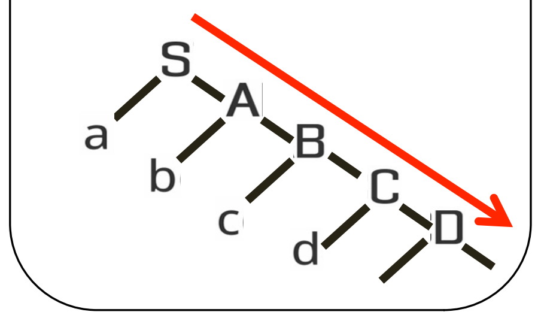
Idea : Linear PEGs

A LPEG is a set of named *linear parsing expressions.* They are specified by rules of the form $A \leftarrow e$ where e is a linear parsing expression and A is the name. The syntax of e is shown in Figure 1.

- Example 1 (LPEG)

A ← a A / b B / c
B ← a B / b A / c

The syntax is limited to right-linear



- Example 2 (not LPEG)

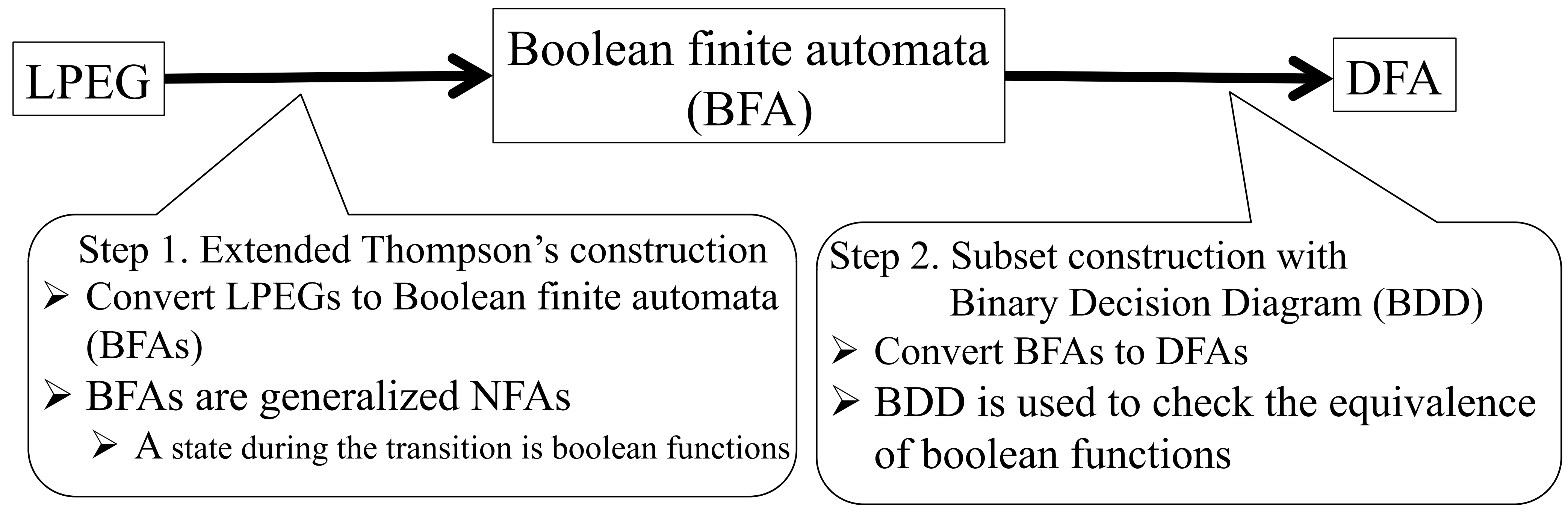
A ← a A b / c A* / d

bad

$$
\begin{array}{llll}
e & ::= & p & \\
 & | & p\,A & \\
 & | & p\,e & \\
 & | & e/e & \\
 & | & \&e\,e & \\
 & | & !e\,e & \\
p & ::= & \epsilon & \text{empty} \\
 & | & a & \text{character} \\
 & | & . & \text{any character} \\
 & | & p\,p & \text{sequence} \\
 & | & p/p & \text{prioritized choice} \\
 & | & p^* & \text{zero or more repetition} \\
 & | & \&p & \text{and-predicate} \\
 & | & !p & \text{not-predicate}
\end{array}
$$

Figure 1. Syntax of a linear parsing expression

Conversion from LPEGs to DFAs :

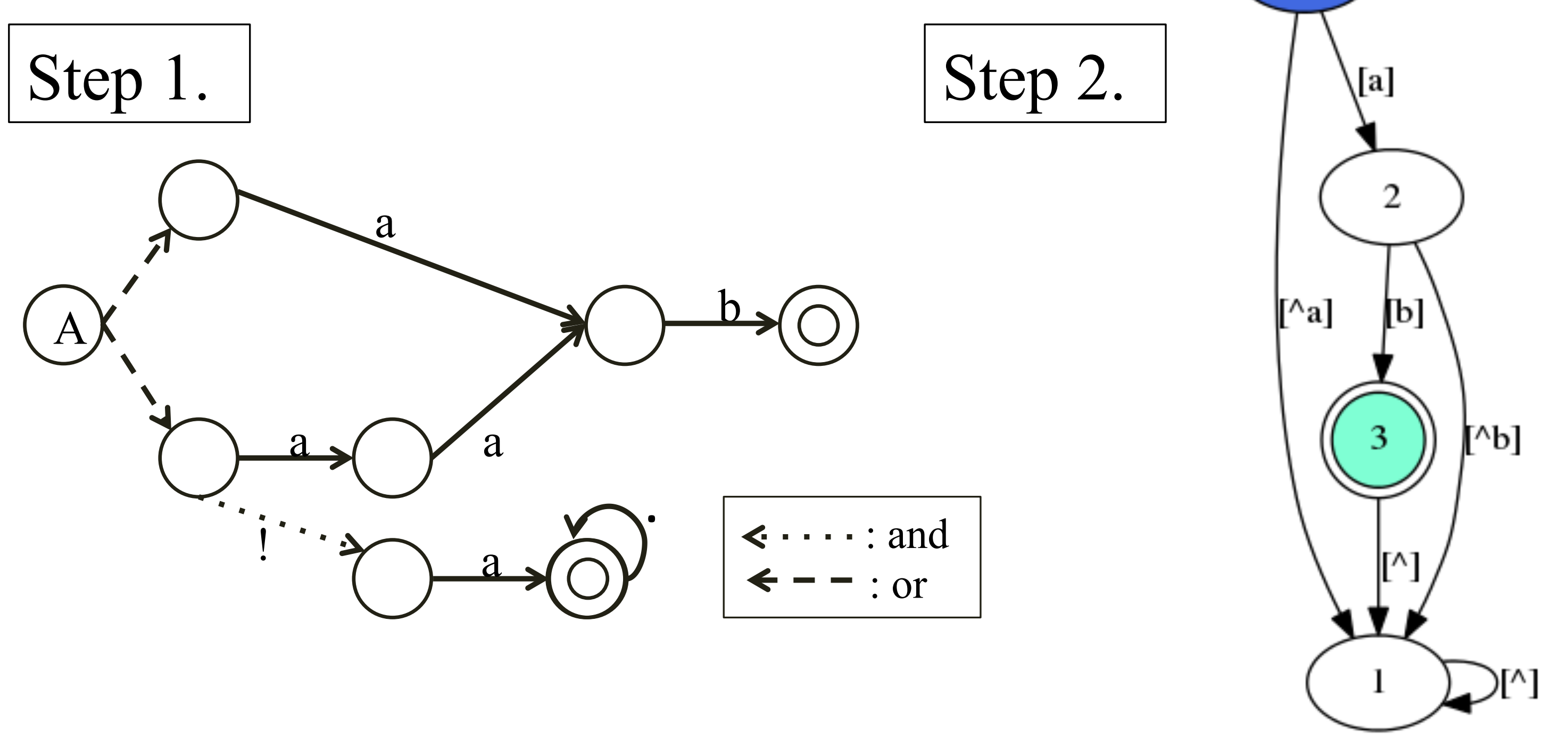


LPEG → Boolean finite automata (BFA) → DFA

Step 1. Extended Thompson's construction
- Convert LPEGs to Boolean finite automata (BFAs)
- BFAs are generalized NFAs
  - A state during the transition is boolean functions

Step 2. Subset construction with Binary Decision Diagram (BDD)
- Convert BFAs to DFAs
- BDD is used to check the equivalence of boolean functions

Example. A ← (a / aa) b



Performance : Applying the conversion to the parsers

We performed experiments to confirm the speed-up of PEG-based parsers by the partial conversion. The conversion is applied for each nonterminal iff the nonterminal is aLPEG. Figure 1 shows XML grammar in PEG. Rules written in red letters are applied to the partial conversion.

- Experiment Method

We measured runtimes ten times in a row and calculated the averages of the runtimes other than the maximum runtime and the minimum runtime.

Table 2 and Table 3 show the result of the runtimes and the number of backtrackings, respectively.

Table 2. Averages of the runtimes

Grammar	Input lines	Normal	DFA
XML	184,966	403ms	109ms
JSON	8,518	336ms	130ms

Table 3. The number of backtracking

Grammar	Normal	DFA
XML	16,589,635	1,054,513
JSON	5,360,918	3,066,286

DFAnized

```
File    <- PROLOG? DTD? Xml
Chunk   <- Xml
Expr    <- Xml
PROLOG  <- '<?xml' (!'?>' .)* '?>' S*
DTD     <- '<!' (!'>' .)* '>' S*
Xml     <- '<' Name S* Attribute* ('/>' / '>
    ' S*
           (Content / COMMENT)* '</' NAME '>
              ') S*
Name    <- NAME
NAME    <- [A-Z_a-z:] ('-' / [.0-9:A-Z_a-z])
    *
Attribute <- Name S* '=' S* String S*
String    <- '"' (!'"' .)* '"'
Content   <- Xml / CDataSec / Text
CDataSec  <- '<![CDATA[' CDATA ']]>' S*
CDATA     <- (!']]>' !'<![CDATA[' .)*
             ('<![CDATA[' CDATA ']]>' CDATA
                )?
COMMENT   <- '<!--' (!'-->' .)* '-->' S*
Text      <- (!'<' .)+
S         <- [ \t\r\n]
```

Figure 1. XML grammar in PEG