From EBNF to PEG

Roman R. Redziejowski

Concurrency, Specification and Programming
Berlin 2012
A way to define grammar.
A way to define grammar.

\[
\text{Literal} \quad = \quad \text{Decimal} \quad | \quad \text{Binary} \\
\text{Decimal} \quad = \quad [0-9]^+ \quad \text{"."} \quad [0-9]^* \\
\text{Binary} \quad = \quad [01]^+ \quad \text{"B"}
\]
Recursive-descent parsing

Parsing procedure for each equation and each terminal.

\[
\begin{align*}
\text{Literal} & = \text{Decimal} \mid \text{Binary} \\
\text{Decimal} & = [0-9]^+ \cdot [0-9]^* \\
\text{Binary} & = [01]^+ \cdot \text{B}
\end{align*}
\]

*Literal* calls *Decimal* or *Binary*.

*Decimal* calls repeatedly [0-9], then ".", then repeatedly [0-9].

*Binary* calls repeatedly [01], then "B".
Recursive-descent parsing

Parsing procedure for each equation and each terminal.

\[
\begin{align*}
\text{Literal} & \quad = \quad \text{Decimal} \quad | \quad \text{Binary} \\
\text{Decimal} & \quad = \quad [0-9]^+ \quad '.' \quad [0-9]^* \\
\text{Binary} & \quad = \quad [01]^+ \quad 'B'
\end{align*}
\]

Literal calls Decimal or Binary.
Decimal calls repeatedly [0-9], then ".", then repeatedly [0-9].
Binary calls repeatedly [01], then "B".

Problem: Decimal and Binary may start with any number of 0’s and 1’s.
Literal cannot choose which procedure to call by looking at any fixed distance ahead.
**Solution: Backtracking**

\[
\text{Literal} = \text{Decimal} \mid \text{Binary}
\]

\[
\text{Decimal} = [0-9]^+ \text{"."} [0-9]^*
\]

\[
\text{Binary} = [01]^+ \text{"B"}
\]

\[
101B
\]

\^
Solution: Backtracking

\[
\begin{align*}
\text{Literal} & \quad = \quad \text{Decimal} \; | \; \text{Binary} \\
\text{Decimal} & \quad = \quad [0-9]^+ \; . \; [0-9]^* \\
\text{Binary} & \quad = \quad [01]^+ \; \text{"B"} \\
\end{align*}
\]

\[
101\text{B}^\wedge
\]

\[
\text{Literal}
\]
Solution: Backtracking

\[
\text{Literal} = \text{Decimal} \mid \text{Binary}
\]

\[
\text{Decimal} = [0-9]^+ \ . \ [0-9]^*
\]

\[
\text{Binary} = [01]^+ \ "B"
\]

101B
^

\text{Literal} \rightarrow \text{Decimal}
Solution: Backtracking

\[
\begin{align*}
\text{Literal} & \quad = \quad \text{Decimal} \quad | \quad \text{Binary} \\
\text{Decimal} & \quad = \quad [0-9]^+ \quad "." \quad [0-9]^* \\
\text{Binary} & \quad = \quad [01]^+ \quad \text{"B"} \\
\end{align*}
\]

101B
\^ \\

\[
\text{Literal} \rightarrow \text{Decimal} \rightarrow [0-9]
\]
Solution: Backtracking

\[
\begin{align*}
\text{Literal} & = \text{Decimal} \mid \text{Binary} \\
\text{Decimal} & = [0-9]^+ \, .\, [0-9]^* \\
\text{Binary} & = [01]^+ \, \text{"B"} \\
\end{align*}
\]

\[101\text{B}\]

\[
\text{Literal} \rightarrow \text{Decimal} \rightarrow [0-9] : \text{advance 3 times}
\]
Solution: Backtracking

\[
\begin{align*}
\text{Literal} & \ = \ \text{Decimal} \mid \text{Binary} \\
\text{Decimal} & \ = \ [0-9]^+ \ . \ [0-9]^* \\
\text{Binary} & \ = \ [01]^+ \ \text{"B"}
\end{align*}
\]

\[
101\text{B}
\]

\[
\uparrow
\]

\[
\text{Literal} \rightarrow \text{Decimal} \rightarrow \text{"."}
\]
Solution: Backtracking

\[
\begin{align*}
\text{Literal} & = \text{Decimal} \mid \text{Binary} \\
\text{Decimal} & = [0-9]^+ \cdot [0-9]^* \\
\text{Binary} & = [01]^+ \cdot \text{"B"}
\end{align*}
\]

\[101\text{\texttt{B}}\]

\[
\text{Literal} \rightarrow \text{Decimal} \rightarrow \cdot \text{: fail, backtrack}
\]
Solution: Backtracking

Literal = Decimal | Binary

Decimal = [0-9]+ "." [0-9]*

Binary = [01]+ "B"

101B

^
Solution: Backtracking

Literal = Decimal | Binary

Decimal = [0-9]+ "." [0-9]*

Binary = [01]+ "B"

101B

^
Solution: Backtracking

\[
\text{Literal} = \text{Decimal} \mid \text{Binary}
\]

\[
\text{Decimal} = [0-9]^+ \cdot [0-9]^*
\]

\[
\text{Binary} = [01]^+ \cdot \text{"B"}
\]

\[
101\text{B}
\]

\[
\wedge
\]

\[
\text{Literal} \rightarrow \text{Binary} \rightarrow [01]
\]
Solution: Backtracking

\[
\begin{align*}
  \text{Literal} &\quad = \quad \text{Decimal} \mid \text{Binary} \\
  \text{Decimal} &\quad = \quad [0-9]^+ \".\" [0-9]^\ast \\
  \text{Binary} &\quad = \quad [01]^+ \"B\"
\end{align*}
\]

\[
101B
\]

\[
\wedge
\]

Literal → Binary → [01] : advance 3 times
Solution: Backtracking

Literal = Decimal | Binary

Decimal = [0-9]+ "." [0-9]*

Binary = [01]+ "B"

101B

Literal → Binary → "B"
Solution: Backtracking

\[
\begin{align*}
\text{Literal} & = \text{Decimal} \mid \text{Binary} \\
\text{Decimal} & = [0-9]^+ \cdot [0-9]^* \\
\text{Binary} & = [01]^+ \ "B"
\end{align*}
\]

\[101B\]

\[\wedge\]

\text{Literal} \rightarrow \text{Binary} \rightarrow "B" : advance, return
Literal = Decimal | Binary

Decimal = [0-9]⁺ "." [0-9]*

Binary = [01]⁺ "B"
Limited backtracking

Backtracking solves the problem, but may take exponential time.
Backtracking solves the problem, but may take exponential time.

Solution: limited backtracking. Never go back after one alternative succeeded.
Limited backtracking

Backtracking solves the problem, but may take exponential time.

Solution: limited backtracking. Never go back after one alternative succeeded.

- 1961 Brooker & Morris - Altas Compiler Compiler
- 1965 McClure - TransMoGrifier (TMG)
- 1972 Aho & Ullman - Top-Down Parsing Language (TDPL)
- ...
- 2004 Ford - Parsing Expression Grammar (PEG)
Limited backtracking

Backtracking solves the problem, but may take exponential time.

Solution: limited backtracking. Never go back after one alternative succeeded.

- 1961 Brooker & Morris - Altas Compiler Compiler
- 1965 McClure - TransMoGrifier (TMG)
- 1972 Aho & Ullman - Top-Down Parsing Language (TDPL)
- ...
- 2004 Ford - Parsing Expression Grammar (PEG)

It can work in linear time.
Looks exactly like EBNF:

\[
\begin{align*}
\text{Literal} &= \text{Decimal} / \text{Binary} \\
\text{Decimal} &= [0-9]^+ \cdot [0-9]^* \\
\text{Binary} &= [01]^+ \text{"B"}
\end{align*}
\]
Looks exactly like EBNF:

\[
\text{Literal} \quad = \quad \text{Decimal} \quad / \quad \text{Binary} \\
\text{Decimal} \quad = \quad [0-9]^+ \quad \text{"."} \quad [0-9]^* \\
\text{Binary} \quad = \quad [01]^+ \quad \text{"B"}
\]

Specification of a recursive-descent parser with limited backtracking, where "/" means an ordered no-return choice.
PEG is not EBNF

<table>
<thead>
<tr>
<th>EBNF:</th>
<th>PEG:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A = (&quot;a&quot; / &quot;aa&quot;) &quot;b&quot; ) {ab, aab}</td>
<td>{ab}</td>
</tr>
<tr>
<td>( A = (&quot;aa&quot; / &quot;a&quot;) &quot;ab&quot; ) {aaab, aab}</td>
<td>{aaab}</td>
</tr>
<tr>
<td>( A = (&quot;a&quot; / &quot;b&quot;?) &quot;a&quot; ) {aa, ba, a}</td>
<td>{aa, ba}</td>
</tr>
</tbody>
</table>
Backtracking may examine input far ahead so result may depend on context in front.
PEG is not EBNF

EBNF:  

A = ("a" / "aa") "b"  \{ab, aab\}  
A = ("aa" / "a") "ab"  \{aaab, aab\}  
A = ("a" / "b"?) "a"  \{aa, ba, a\}  

PEG:  

\{ab\}  
\{aaab\}  
\{aa, ba\}  

Backtracking may examine input far ahead so result may depend on context in front.

A = "a" A "a" / "aa"  
EBNF:  \(a^{2n}\)  
PEG:  \(a^{2n}\)
In this case PEG = EBNF:

\[
\begin{align*}
\text{Literal} & = \text{Decimal} / \text{Binary} \\
\text{Decimal} & = [0-9]^+ \ "." \ [0-9]^* \\
\text{Binary} & = [01]^+ "B"
\end{align*}
\]
In this case PEG = EBNF:

\[
\begin{align*}
\text{Literal} & = \text{Decimal} / \text{Binary} \\
\text{Decimal} & = [0-9]^+ \cdot [0-9]^* \\
\text{Binary} & = [01]^+ \text{"B"}
\end{align*}
\]

When does it happen?
When PEG = EBNF?

Sérgio Queiroz de Medeiros

Correspondência entre PEGs e Classes de Gramáticas Livres de Contexto.

Ph.D. Thesis

When PEG = EBNF?

Sérgio Queiroz de Medeiros
Correspondência entre PEGs e Classes de Gramáticas Livres de Contexto.
Ph.D. Thesis

If EBNF has LL(1) property then PEG = EBNF
But this is not LL(1):

\[
\begin{align*}
\text{Literal} & = \text{Decimal} \ / \ \text{Binary} \\
\text{Decimal} & = [0-9]^+ \ "." \ [0-9]^* \\
\text{Binary} & = [01]^+ \ "B" 
\end{align*}
\]
But this is not LL(1):

- **Literal** = Decimal / Binary
- **Decimal** = [0-9]+ "." [0-9]*
- **Binary** = [01]+ "B"

Which means PEG = EBNF for a wider class.
But this is not LL(1):

\[
\text{Literal} = \text{Decimal} / \text{Binary} \\
\text{Decimal} = [0-9]^+ "." [0-9]^* \\
\text{Binary} = [01]^+ "B"
\]

Which means PEG = EBNF for a wider class. Let us find more about it.
Simple grammar

Alphabet $\Sigma$ (the "terminals").

Set $N$ of names (the "nonterminals").

For each $A \in N$ one rule of the form:
- $A = e_1 \ e_2$ (Sequence) or
- $A = e_1 \mid e_2$ (Choice)

where $e_1, e_2 \in N \cup \Sigma \cup \{ \varepsilon \}$.

Start symbol $S \in A$.

"Syntax expressions": $\mathcal{E} = N \cup \Sigma \cup \{ \varepsilon \}$.
Simple grammar

Alphabet $\Sigma$ (the "terminals").

Set $N$ of names (the "nonterminals").

For each $A \in N$ one rule of the form:

- $A = e_1 \ e_2$ (Sequence) or
- $A = e_1 \ | \ e_2$ (Choice)

where $e_1, e_2 \in N \cup \Sigma \cup \{\varepsilon\}$.

Start symbol $S \in A$.

"Syntax expressions": $E = N \cup \Sigma \cup \{\varepsilon\}$.

Will consider two interpretations: EBNF and PEG.
\( \mathcal{L}(e) \) – language of expression \( e \in \mathbb{E} \).

- \( \mathcal{L}(\varepsilon) = \{\varepsilon\} \)
- \( \mathcal{L}(a) = \{a\} \) for \( a \in \Sigma \)
- \( \mathcal{L}(A) = \mathcal{L}(e_1)\mathcal{L}(e_2) \) for \( A = e_1 e_2 \)
- \( \mathcal{L}(A) = \mathcal{L}(e_1) \cup \mathcal{L}(e_2) \) for \( A = e_1 \mid e_2 \)

Language defined by the grammar: \( \mathcal{L}(S) \).
Relation $\leadsto \subseteq \mathbb{E} \times \Sigma^* \times \Sigma^*$, written $[e] \ x \leadsto y$.

$[e] \ xy \leadsto y$ means "$xy$ has prefix $x \in \mathcal{L}(e)$".

Or: parsing procedure for $e$, applied to $xy$ consumes $x$.

$w \in \mathcal{L}(S) \iff [S] \ w$ $\leadsto$ $\$$

where $\$$ is "end of text" marker.
*[e] x ↑^{BNF} y holds if and only if it can be proved using these inference rules:

\[
\begin{align*}
\varepsilon x & \xrightarrow{BNF} x \quad \alpha x & \xrightarrow{BNF} x \\
A = e_1 e_2 & \quad [e_1] xyz & \xrightarrow{BNF} yz \quad [e_2] yz & \xrightarrow{BNF} z \\
& \quad [A] xyz & \xrightarrow{BNF} z \\
A = e_1 | e_2 & \quad [e_1] xy & \xrightarrow{BNF} y \\
& \quad [A] xy & \xrightarrow{BNF} y \\
A = e_1 | e_2 & \quad [e_2] xy & \xrightarrow{BNF} y \\
& \quad [A] xy & \xrightarrow{BNF} y
\end{align*}
\]
Example of proof

Grammar: $S = aX$, $X = S|b$  

Proof of $aab \in \mathcal{L}(S)$

$$
\begin{array}{c}
\hline
[b] \ b$ \ BNF \ \rightsquigarrow \ \$

\hline
\end{array}

\begin{array}{c}
\hline
X = S|b \quad \begin{array}{c}
\hline
[b] \ b$ \ BNF \ \rightsquigarrow \ \$

\hline
\end{array}

\begin{array}{c}
\hline
[a] \ ab$ \ BNF \ \rightsquigarrow \ b$

\hline
\end{array}

\begin{array}{c}
\hline
[X] \ b$ \ BNF \ \rightsquigarrow \ $

\hline
\end{array}

\begin{array}{c}
\hline
S = aX \quad [a] \ ab$ \ BNF \ \rightsquigarrow \ b$ \quad [X] \ b$ \ BNF \ \rightsquigarrow \ $

\hline
\end{array}

\begin{array}{c}
\hline
[S] \ ab$ \ BNF \ \rightsquigarrow \ $

\hline
\end{array}

\begin{array}{c}
\hline
X = S|b \quad [S] \ ab$ \ BNF \ \rightsquigarrow \ $

\hline
\end{array}

\begin{array}{c}
\hline
[a] \ aab$ \ BNF \ \rightsquigarrow \ ab$

\hline
\end{array}

\begin{array}{c}
\hline
[X] \ ab$ \ BNF \ \rightsquigarrow \ $

\hline
\end{array}

\begin{array}{c}
\hline
S = aX \quad [a] \ aab$ \ BNF \ \rightsquigarrow \ ab$ \quad [X] \ ab$ \ BNF \ \rightsquigarrow \ $

\hline
\end{array}

\begin{array}{c}
\hline
[S] \ aab$ \ BNF \ \rightsquigarrow \ $

\hline
\end{array}

Roman R. Redziejowski  
From EBNF to PEG
Elements of $\mathcal{E}$ are parsing procedures that consume input or return "failure".

- $\varepsilon$ returns success without consuming input.
- $a$ consumes $a$ if input starts with $a$. Otherwise returns failure.
- $A = e_1 e_2$ calls $e_1$ then $e_2$. If any of them failed, backtracks and returns failure.
- $A = e_1 | e_2$ calls $e_1$. If $e_1$ succeeded, returns success. If $e_1$ failed, calls $e_2$ and returns its result.
Relation $\overset{\text{PEG}}{\rightsquigarrow} \subseteq \mathbb{E} \times \Sigma^* \times (\Sigma^* \cup \text{fail})$, written $[e] \overset{\text{PEG}}{\rightsquigarrow} y$.

- $[e] \overset{\text{PEG}}{\rightsquigarrow} xy$ means "$e$ consumes prefix $x$ of $xy$".
- $[e] \overset{\text{PEG}}{\rightsquigarrow} x \text{fail}$ means "$e$ applied to $x$ returns failure".

$w$ accepted by the grammar iff $[S] \overset{\text{PEG}}{\rightsquigarrow} w$. 

Roman R. Redziejowski  From EBNF to PEG
Natural semantics" (after Medeiros)

\[ e \] \( \xrightarrow{\text{PEG}} \) \( Y \) holds if and only if
it can be proved using these inference rules:

- \( b \neq a \)
- \( [\varepsilon] \) \( \xrightarrow{\text{PEG}} \) \( x \)
- \( [a] \) \( ax \xrightarrow{\text{PEG}} \) \( x \)
- \( [b] \) \( ax \xrightarrow{\text{PEG}} \) \( \text{fail} \)
- \( [a] \) \( \varepsilon \xrightarrow{\text{PEG}} \) \( \text{fail} \)

\( A = e_1 e_2 \)
\( [e_1] \) \( xyz \xrightarrow{\text{PEG}} \) \( yz \)
\( [e_2] \) \( yz \xrightarrow{\text{PEG}} \) \( Z \)
\( [A] \) \( xyz \xrightarrow{\text{PEG}} \) \( Z \)

\( A = e_1 e_2 \)
\( [e_1] \) \( \xrightarrow{\text{PEG}} \) \( \text{fail} \)
\( [A] \) \( \xrightarrow{\text{PEG}} \) \( \text{fail} \)

\( A = e_1 | e_2 \)
\( [e_1] \) \( \xrightarrow{\text{PEG}} \) \( \text{fail} \)
\( [e_2] \) \( xy \xrightarrow{\text{PEG}} \) \( y \)
\( [A] \) \( xy \xrightarrow{\text{PEG}} \) \( y \)

\( A = e_1 | e_2 \)
\( [e_1] \) \( \xrightarrow{\text{PEG}} \) \( \text{fail} \)
\( [e_2] \) \( xy \xrightarrow{\text{PEG}} \) \( Y \)
\( [A] \) \( xy \xrightarrow{\text{PEG}} \) \( Y \)

where \( Y \) is \( y \) or \( \text{fail} \) and \( Z \) is \( z \) or \( \text{fail} \).
By induction on the height of proof trees for $[S] \, w$\, PEG $\Rightarrow$ and $[S] \, w$\, BNF $\Rightarrow$:

- $[S] \, w$\, PEG $\Rightarrow$ $\Rightarrow$ $[S] \, w$\, BNF $\Rightarrow$. (Medeiros)
By induction on the height of proof trees for 
\( [S] \ w^\text{PEG} \rightsquigarrow \)$ and \( [S] \ w^\text{BNF} \rightsquigarrow \)$:

- \( [S] \ w^\text{PEG} \rightsquigarrow \) ⇒ \( [S] \ w^\text{BNF} \rightsquigarrow \). (Medeiros)
- \( [S] \ w^\text{BNF} \rightsquigarrow \) ⇒ \( [S] \ w^\text{PEG} \rightsquigarrow \)
  if for every Choice \( A = e_1 | e_2 \) holds
  \( \mathcal{L}(e_1) \cap \text{Pref}(\mathcal{L}(e_2) \text{Tail}(A)) = \emptyset \).
When PEG = EBNF?

By induction on the height of proof trees for
\([S] \ x \ PEG \sim \ y \) and \([S] \ x \ BNF \sim \ y \):

- \([S] \ x \ PEG \sim \ y \) \implies \([S] \ x \ BNF \sim \ y \). (Medeiros)

- \([S] \ x \ BNF \sim \ y \) \implies \([S] \ x \ PEG \sim \ y \)

  if for every Choice \(A = e_1 | e_2\) holds

\[\mathcal{L}(e_1) \cap \text{Pref}(\mathcal{L}(e_2) \text{Tail}(A)) = \emptyset.\]

(Tail(A) is any possible continuation after A:
\(y \in \text{Tail}(A)\) iff proof tree of \([S] \ x \ BNF \sim \ y \) for some \(x\)
contains partial result \([A] \ xy \ BNF \sim \ y \).)
Let us say that Choice $A = e_1 | e_2$ is "safe" to mean 
$\mathcal{L}(e_1) \cap \text{Pref}(\mathcal{L}(e_2) \text{ Tail}(A)) = \emptyset$. 
Let us say that Choice $A = e_1 | e_2$ is "safe" to mean $\mathcal{L}(e_1) \cap \text{Pref} (\mathcal{L}(e_2) \text{ Tail}(A)) = \emptyset$.

The two interpretations are equivalent if every Choice in the grammar is safe.
$\mathcal{L}(e_1) \cap \text{Pref}(\mathcal{L}(e_2) \text{Tail}(A)) = \emptyset$
\[ \mathcal{L}(e_1) \cap \text{Pref}(\mathcal{L}(e_2) \text{Tail}(A)) = \emptyset \]

Requires \( \varepsilon \notin \mathcal{L}(e_1) \).
\[ \mathcal{L}(e_1) \cap \text{Pref}(\mathcal{L}(e_2) \text{Tail}(A)) = \emptyset \]

Requires \( \varepsilon \notin \mathcal{L}(e_1) \).

Depends on context.
\[ \mathcal{L}(e_1) \cap \text{Pref}(\mathcal{L}(e_2) \text{Tail}(A)) = \emptyset \]

Requires \( \varepsilon \notin \mathcal{L}(e_1) \).

Depends on context.

Difficult to check: \( \mathcal{L}(e_1) \), \( \mathcal{L}(e_2) \), and \( \text{Tail}(A) \) can be any context-free languages.

Intersection of context-free languages is in general undecidable.
\[ \mathcal{L}(e_1) \cap \text{Pref}(\mathcal{L}(e_2) \text{Tail}(A)) = \emptyset \]

Requires \( \varepsilon \not\in \mathcal{L}(e_1) \).

Depends on context.

Difficult to check: \( \mathcal{L}(e_1) \), \( \mathcal{L}(e_2) \), and \( \text{Tail}(A) \) can be any context-free languages.

Intersection of context-free languages is in general undecidable.

Can be "approximated" by stronger conditions.
Consider $A = e_1|e_2$.

$\text{FIRST}(e_1), \text{FIRST}(e_2)$: sets of possible first letters of words in $\mathcal{L}(e_1)$ respectively $\mathcal{L}(e_2)$. 
Consider $A = e_1|e_2$.

$\textsc{First}(e_1), \textsc{First}(e_2)$:
sets of possible first letters
of words in $\mathcal{L}(e_1)$ respectively $\mathcal{L}(e_2)$.

If $\mathcal{L}(e_1), \mathcal{L}(e_2)$, do not contain $\varepsilon$,
$\textsc{First}(e_1) \cap \textsc{First}(e_2) = \emptyset$
implies $\mathcal{L}(e_1) \cap \text{Pref}(\mathcal{L}(e_2) \text{Tail}(A)) = \emptyset$. 
Consider $A = e_1 | e_2$.

$\text{FIRST}(e_1), \text{FIRST}(e_2)$:  
sets of possible first letters 
of words in $\mathcal{L}(e_1)$ respectively $\mathcal{L}(e_2)$.

If $\mathcal{L}(e_1), \mathcal{L}(e_2)$, do not contain $\varepsilon$,  
$\text{FIRST}(e_1) \cap \text{FIRST}(e_2) = \emptyset$ 
implies $\mathcal{L}(e_1) \cap \text{Pref}(\mathcal{L}(e_2) \text{Tail}(A)) = \emptyset$.

This is LL(1) for grammar without $\varepsilon$.  
Each choice in such grammar is safe.  
The two interpretations are equivalent.
To go beyond LL(1), we shall look at first *expressions* rather than first *letters*.
\[
S = X \mid Y \\
X = Z \mid V \\
Y = WX \\
Z = a \mid b \\
V = b \mid T \\
W = d \mid U \\
T = cV \\
U = cW
\]
Computing $\text{FIRST}$

$S = X \mid Y$
$X = Z \mid V$
$Y = WX$
$Z = a \mid b$
$V = b \mid T$
$W = d \mid U$
$T = cV$
$U = cW$

$\text{FIRST}(X) = \{a, b, c\}$
Computing \textsc{First}

\begin{align*}
S &= X \mid Y \\
X &= Z \mid V \\
Y &= WX \\
Z &= a \mid b \\
V &= b \mid T \\
W &= d \mid U \\
T &= cV \\
U &= cW
\end{align*}

\textsc{First}(X) = \{a, b, c\}
\textsc{First}(Y) = \{c, d\}
Computing FIRST

\[ S = X \mid Y \]
\[ X = Z \mid V \]
\[ Y = W X \]
\[ Z = a \mid b \]
\[ V = b \mid T \]
\[ W = d \mid U \]
\[ T = c V \]
\[ U = c W \]

\( \text{FIRST}(X) = \{a, b, c\} \)
\( \text{FIRST}(Y) = \{c, d\} \)
\( \{a, b, c\} \cap \{c, d\} \neq \emptyset : S = X \mid Y \) is not LL(1).
Truncated computation of FIRST

\[ S = X \mid Y \]
\[ X = Z \mid V \]
\[ Y = WX \]
\[ Z = a \mid b \]
\[ V = b \mid T \]
\[ W = d \mid U \]
\[ T = cV \]
\[ U = cW \]
Truncated computation of FIRST

\[
S = X | Y \\
X = Z | V \\
Y = WX \\
Z = a | b \\
V = b | T \\
W = d | U \\
T = cV \\
U = cW
\]

Each word in \( \mathcal{L}(X) \) has a prefix in \( \{a, b\} \cup \mathcal{L}(T) = a \cup c^*b \).
Truncated computation of \textsc{First}

\[
\begin{align*}
S &= X \mid Y \\
X &= Z \mid V \\
Y &= WX \\
Z &= a \mid b \\
V &= b \mid T \\
W &= d \mid U \\
T &= cV \\
U &= cW
\end{align*}
\]

Each word in \( \mathcal{L}(X) \) has a prefix in \( \{a, b\} \cup \mathcal{L}(T) = a \cup c^*b \).
Each word in \( \mathcal{L}(Y) \) has a prefix in \( \{d\} \cup \mathcal{L}(U) = d^*b \).
Approximation by first expressions

Each word in \( \mathcal{L}(X) \) has a prefix in \( a \cup c^*b \).
Each word in \( \mathcal{L}(Y) \) has a prefix in \( d^*b \).

\[
\begin{align*}
\mathcal{L}(X) &= (a \cup c^*b)(\ldots) \\
\mathcal{L}(Y) &= (c^*d)(\ldots)
\end{align*}
\]
Each word in $\mathcal{L}(X)$ has a prefix in $a \cup c^*b$.
Each word in $\mathcal{L}(Y)$ has a prefix in $d^*b$.

$\mathcal{L}(X) = (a \cup c^*b)(\ldots)$

$\mathcal{L}(Y) = (c^*d)(\ldots)$

$\mathcal{L}(X) \cap \text{Pref}(\mathcal{L}(Y) \text{Tail}(S))$

$= (a \cup c^*b)(\ldots) \cap \text{Pref}((c^*d)(\ldots)(\ldots))$
Each word in $\mathcal{L}(X)$ has a prefix in $a \cup c^* b$.
Each word in $\mathcal{L}(Y)$ has a prefix in $d^* b$.

$\mathcal{L}(X) = (a \cup c^* b)(\ldots)$
$\mathcal{L}(Y) = (c^* d)(\ldots)$

$\mathcal{L}(X) \cap \text{Pref}(\mathcal{L}(Y) \text{Tail}(S))$
\hspace{1cm} = (a \cup c^* b)(\ldots) \cap \text{Pref}((c^* d)(\ldots)(\ldots))$

No word in $a \cup c^* b$ is a prefix of word in $c^* d$ and vice-versa.
Each word in $\mathcal{L}(X)$ has a prefix in $a \cup c^* b$.
Each word in $\mathcal{L}(Y)$ has a prefix in $d^* b$.

$L(X) = (a \cup c^* b)(\ldots)$
$L(Y) = (c^* d)(\ldots)$

$L(X) \cap \text{Pref}(L(Y) \text{ Tail}(S))$
\hspace{1cm} = (a \cup c^* b)(\ldots) \cap \text{Pref}((c^* d)(\ldots)(\ldots))$

No word in $a \cup c^* b$ is a prefix of word in $c^* d$ and vice-versa.
The intersection is empty: $S = X|Y$ is safe.
Some terminology

$X$ starts with $a$, $b$, or $T$:
"$X$ has $a$, $b$, and $T$ as possible first expressions".

$\{a, b, T\} \sqsubseteq X$
Some terminology

$X$ starts with $a$, $b$, or $T$:
"$X$ has $a$, $b$, and $T$ as possible first expressions".

$\{a, b, T\} \subseteq X$

No word in $a$, $b$, or $T$ is a prefix of a word in $d$ or $U$ and vice-versa:
"$\{a, b, T\}$ and $\{d, U\}$ are exclusive".

$\{a, b, T\} \preceq \{d, U\}$
If $\varepsilon \notin e_1$ and $\varepsilon \notin e_2$
and there exist $\text{FIRST}_1 \sqsubseteq e_1$, $\text{FIRST}_2 \sqsubseteq e_2$
such that $\text{FIRST}_1 \asymp \text{FIRST}_2$
then $A = e_1 | e_2$ is safe.
The two interpretations of an $\varepsilon$-free grammar are equivalent if for every Choice $A = e_1 | e_2$, $e_1$ and $e_2$ have exclusive sets of first expressions.
(Good news) Grammar with $\varepsilon$ is easy to handle. This involves first expressions of Tail($A$), that are obtained using the classical computation of FOLLOW.
(Good news) Grammar with $\varepsilon$ is easy to handle. This involves first expressions of $\text{Tail}(A)$, that are obtained using the classical computation of $\text{FOLLOW}$.

(Good news) The results for simple grammar are easily extended to full EBNF / PEG.
Final remarks

- (Good news) Grammar with $\varepsilon$ is easy to handle. This involves first expressions of $\text{Tail}(A)$, that are obtained using the classical computation of $\text{FOLLOW}$.
- (Good news) The results for simple grammar are easily extended to full EBNF / PEG.
- (Good news) The possible sets of first expressions are easily obtained in a mechanical way.
Final remarks

- (Good news) Grammar with $\varepsilon$ is easy to handle. This involves first expressions of Tail($A$), that are obtained using the classical computation of FOLLOW.

- (Good news) The results for simple grammar are easily extended to full EBNF / PEG.

- (Good news) The possible sets of first expressions are easily obtained in a mechanical way.

- (Bad news) Checking that they are exclusive is not easy: it is undecidable in general case (but we may hope first expressions are simple enough to be decidable.)
S = (aa|a)b  (that is: S = Xb, X = aa|a.)
S = (aa|a)b  (that is: S = Xb, X = aa|a.)

\( \mathcal{L}(e_1) \cap \text{Pref}(\mathcal{L}(e_2) \text{Tail}(X)) = aa \cap \text{Pref}(ab) = \emptyset. \)
S = (aa|a)b  (that is: S = Xb, X = aa|a.)

\[ \mathcal{L}(e_1) \cap \text{Pref}(\mathcal{L}(e_2) \text{Tail}(X)) = aa \cap \text{Pref}(ab) = \emptyset. \]

X is safe. Both interpretations accept \{aab,ab\}. 
S = (aa|a)b  (that is: S = Xb, X = aa|a.)

\[ \mathcal{L}(e_1) \cap \text{Pref}(\mathcal{L}(e_2) \text{Tail}(X)) = aa \cap \text{Pref}(ab) = \emptyset. \]

X is safe. Both interpretations accept \{aab,ab\}.

Sets of first expressions in X: \{aa\} and \{a\}. Not exclusive!
Final final remark

\[ S = (aa|a)b \quad \text{(that is: } S = Xb, \ X = aa|a.) \]

\[ \mathcal{L}(e_1) \cap \text{Pref}(\mathcal{L}(e_2) \text{Tail}(X)) = aa \cap \text{Pref}(ab) = \emptyset. \]

\( X \) is safe. Both interpretations accept \{aab,ab\}.

Sets of first expressions in \( X \): \{aa\} and \{a\}. Not exclusive!

There is more to squeeze out of \( \mathcal{L}(e_1) \cap \text{Pref}(\mathcal{L}(e_2) \text{Tail}(A)). \)
That’s all

Thanks for your attention!