What is Parsing Expression Grammar?

The last fad in top-down parsing with limited backtracking.

- 1961 Brooker & Morris - Altas Compiler Compiler
- 1965 McClure - TransMoGrifier (TMG)
- 1972 Aho & Ullman - Top-Down Parsing Language (TDPL)
- ...  
- 2004 Ford - Parsing Expression Grammar (PEG)
number = real / integer
real = digits? "." digits
integer = digits
digits = [0-9][0-9]∗
number = real / integer
real = digits? "." digits
integer = digits
digits = [0-9][0-9]*

Not a grammar: a recursive-descent parser.
number = real / integer
real = digits? "." digits
integer = digits
digits = [0-9][0-9] *

Not a grammar: a recursive-descent parser.
Named parsing procedures ("parsing expressions").
number  = real / integer
real    = digits? "." digits
integer = digits
digits  = [0-9][0-9]∗

Not a grammar: a recursive-descent parser.
Named parsing procedures ("parsing expressions").
Call other procedures and "terminals".
number = real / integer
real    = digits? "." digits
integer = digits
digits  = [0-9][0-9]*

Not a grammar: a recursive-descent parser.
Named parsing procedures ("parsing expressions").
Call other procedures and "terminals".
Note: not LL(1).
number = real / integer
real = digits? "." digits
integer = digits
digits = [0-9][0-9]*
number = real / integer
real = digits? "." digits
integer = digits
digits = [0-9][0-9] *

29.165
^
number = real / integer
real = digits? "." digits
integer = digits
digits = [0-9][0-9]*

29.165

number
number = real / integer
real = digits? "." digits
integer = digits
digits = [0-9][0-9]*

29.165
^

number->real
number = real / integer
real    = digits? "." digits
integer = digits
digits  = [0-9][0-9]*

29.165
^

number->real->digits
number = real / integer
real = digits? "." digits
integer = digits
digits = [0-9][0-9]*k

29.165
^

number->real->digits->[0-9][0-9]*k
number = real / integer
real = digits? "." digits
integer = digits
digits = [0-9][0-9]*

29.165
^

number->real->digits->[0-9][0-9]*: consume "29"
number  = real / integer
real    = digits? "." digits
integer = digits
digits  = [0-9][0-9]*

29.165
^  

number->real->digits
number  = real / integer
real    = digits? "." digits
integer = digits
digits  = [0-9][0-9]∗

```
29.165
^
```

number->real
number = real / integer
real = digits? "." digits
integer = digits
digits = [0-9][0-9] *

29.165
^

number->real->"."
number = real / integer
real = digits? "." digits
integer = digits
digits = [0-9][0-9]*

29.165
^

number->real->".": consumes "."
number = real / integer
real = digits? "." digits
integer = digits
digits = [0-9][0-9]*

29.165
^

number->real
number  = real / integer  
real     = digits? "." digits
integer = digits
digits  = [0-9][0-9]*

29.165
 ^

number->real->digits
number  = real / integer
real     = digits? "." digits
integer  = digits
digits   = [0-9][0-9]*

29.165
^

number->real->digits: consume "165"
number  = real / integer
real     = digits? "." digits
integer = digits
digits  = [0-9][0-9]*

29.165
    ^

number->real
number  = real / integer
real     = digits? "." digits
integer  = digits
digits   = [0-9][0-9]*

29.165
   ^

number
number = real / integer
real = digits? "." digits
integer = digits
digits = [0-9][0-9]*

29.165
^
number = real / integer
real = digits? "." digits
integer = digits
digits = [0-9][0-9]∗
number = real / integer
real = digits? "." digits
integer = digits
digits = [0-9][0-9]*

4711
^
number = real / integer
real = digits? "." digits
integer = digits
digits = [0-9][0-9]*

4711
^
number   = real / integer
real     = digits? "." digits
integer  = digits
digits   = [0-9][0-9]*

4711
^
number  = real / integer
real    = digits? "." digits
integer = digits
digits  = [0-9][0-9]*
number  = real / integer
real     = digits? "." digits
integer  = digits
digits   = [0-9][0-9] *

4711
     ^

number->real->digits: consume "4711"
number = real / integer
real = digits? "." digits
integer = digits
digits = [0-9][0-9]*

4711
^

number->real
number = real / integer
real = digits? "." digits
integer = digits
digits = [0-9][0-9]*

4711
    ^

number->real->"."
number  = real / integer  
real     = digits? "." digits  
icentenger  = digits  
digits   = [0-9][0-9] *  

4711  

number->real->".": returns failure
number = real / integer
real = digits? "." digits
integer = digits
digits = [0-9][0-9]*

4711
^

number->real: backtracks
PEG in action: backtracking

\[
\begin{align*}
\text{number} & = \text{real} / \text{integer} \\
\text{real} & = \text{digits? } "." \text{ digits} \\
\text{integer} & = \text{digits} \\
\text{digits} & = [0-9][0-9]^* \\
\end{align*}
\]

4711
^

\text{number->real: returns failure}
number = real / integer
digits = [0-9]*
integer = digits
real = digits? "." digits
decimal = 4711
4711
^
number = real / integer
real = digits? "." digits
integer = digits
digits = [0-9][0-9]*

number -> integer

4711
^
number = real / integer
real    = digits? "." digits
integer = digits
digits  = [0-9][0-9]*

4711
^

number->integer->digits
number  = real / integer  
real     = digits? "." digits  
integer  = digits  
digits   = [0-9][0-9]*  

4711

number->integer->digits: consume "4711"
number = real / integer
real = digits? "." digits
integer = digits
digits = [0-9][0-9]*

4711
^

number->integer
number = real / integer
real = digits? "." digits
integer = digits
digits = [0–9][0–9]∗

4711
^

number
PEG in action: backtracking

number = real / integer
real = digits? "." digits
integer = digits
digits = [0-9][0-9]*

4711
^
Backtracking is limited

```
number   = integer / real
real     = digits? "." digits
integer  = digits
digits   = [0-9][0-9]*
```

Once a number succeeds, nothing can force it to try a real. The integer hides part of the language of real.
Backtracking is limited

```
number = integer / real
real = digits? "." digits
integer = digits
digits = [0-9][0-9] *

29.165
^
Backtracking is limited

\[
\begin{align*}
\text{number} & = \text{integer} / \text{real} \\
\text{real} & = \text{digits}? \text{"."} \text{ digits} \\
\text{integer} & = \text{digits} \\
\text{digits} & = [0-9][0-9]^* \\
\end{align*}
\]

29.165
^

\text{number}
Backtracking is limited

```
number = integer / real
real   = digits? "." digits
integer = digits
digits  = [0-9][0-9]*

29.165
^  
number->integer
```
Backtracking is limited

number = integer / real
real = digits? "." digits
integer = digits
digits = [0-9][0-9]*

29.165
^

number->integer->digits
number = integer / real
real = digits? "." digits
integer = digits
digits = [0-9][0-9] *

29.165

^

number->integer->digits: consume "29"
Backtracking is limited

number = integer / real
real = digits? "." digits
integer = digits
digits = [0-9][0-9] *

29.165
^

number->integer
Backtracking is limited

```plaintext
number  = integer / real
real    = digits? "." digits
integer = digits
digits  = [0-9][0-9]*

29.165
  ^

number
```
number = integer / real
real = digits? "." digits
integer = digits
digits = [0-9][0-9]*

29.165
   ^
Backtracking is limited

number = integer / real
real = digits? "." digits
integer = digits
digits = [0-9][0-9]*

\[29.165\]

Once `number` succeeded, nothing can force it to try `real`. 
Backtracking is limited

\[
\begin{align*}
\text{number} & = \text{integer} / \text{real} \\
\text{real} & = \text{digits}? \cdot \text{digits} \\
\text{integer} & = \text{digits} \\
\text{digits} & = [0-9][0-9]^* \\
\end{align*}
\]

29.165

Once \text{number} succeeded, nothing can force it to try \text{real}.
\text{integer} hides part of the language of \text{real}. 
All of these fail on input aab:

("a"/"aa")"b" - "a" consumes a, "b" fails on ab

("aa"/"a")"ab"

("a"/"c"?)"aab"
All of these fail on input `aab`:

- `("a"/"aa") "b"` - "a" consumes `a`, "b" fails on `ab`
- `("aa"/"a") "ab"`
- `("a"/"c"?) "aab"`

Not easy to see what happens in a complex grammar.
Guess what this is doing:

\[ A = \text{"a"}A\text{"a"} \ /	ext{"aa"} \]
Guess what this is doing:

\[ A = "a"A"a" / "aa" \]

aaaaa consumes 4 of 4
Some fun

Guess what this is doing:

\[ A = \"a\"A\"a\" / \"aa\" \]

aaaa consumes 4 of 4
aaaaa 2 of 5

Result depends on input far ahead.

Programmer's paradise: write, try, debug, show your skill.

Roman R. Redziejowski

FIRST and FOLLOW for PEG
Guess what this is doing:

\[ A = "a"A"a" / "aa" \]

aaaa consumes 4 of 4
aaaaa 2 of 5
aaaaaaa 4 of 6

Result depends on input far ahead.
Some fun

Guess what this is doing:

\[ A = "a"A"a" / "aa" \]

aaaa consumes 4 of 4
aaaaaa 2 of 5
aaaaaaa 4 of 6
aaaaaaaa 6 of 7

Result depends on input far ahead.

Programmer's paradise: write, try, debug, show your skill.

Roman R. Redziejowski
FIRST and FOLLOW for PEG
Some fun

Guess what this is doing:

A = "a""A""a" / "aa"

aaaaa consumes 4 of 4
aaaaaa 2 of 5
aaaaaaa 4 of 6
aaaaaaaaa 6 of 7
aaaaaaaaaa 8 of 8

Result depends on input far ahead.

Programmer's paradise: write, try, debug, show your skill.

Roman R. Redziejowski

FIRST and FOLLOW for PEG
Guess what this is doing:

\[ A = "a"A"a" / "aa" \]

- aaaaa consumes 4 of 4
- aaaaa 2 of 5
- aaaaaaa 4 of 6
- aaaaaaaaa 6 of 7
- aaaaaaaaaa 8 of 8
- aaaaaaaaaaa 2 of 9
Guess what this is doing:

\[ A = "a"A"a" / "aa" \]

aaaa consumes 4 of 4
aaaaa 2 of 5
aaaaaaa 4 of 6
aaaaaaaa 6 of 7
aaaaaaaaa 8 of 8
aaaaaaaaaa 2 of 9

Result depends on input far ahead.
Guess what this is doing:

\[ A = "a"A"a" / "aa" \]

- `aaaa` consumes 4 of 4
- `aaaaaa` 2 of 5
- `aaaaaaaa` 4 of 6
- `aaaaaaaaa` 6 of 7
- `aaaaaaaaaa` 8 of 8
- `aaaaaaaaaaa` 2 of 9

Result depends on input far ahead.

Programmer’s paradise: write, try, debug, show your skill.
Problem

- General problem:
  understand what this damned thing is doing.

Observation: problems are associated with LL(1) violations.

Suggestion: detect LL(1) violations.

How: adapt known techniques to PEG.
General problem: 
understand what this damned thing is doing.
Very difficult. (CS&P 2007, Fundamenta Inf. 85).
Problem

- General problem: understand what this damned thing is doing.
- Very difficult. (CS&P 2007, Fundamenta Inf. 85).
- Partial problem: detect language hiding in a complex grammar.

Observation: problems are associated with LL(1) violations.
Suggestion: detect LL(1) violations. How: adapt known techniques to PEG.
Problem

- General problem:
  understand what this damned thing is doing.
- Very difficult. (CS&P 2007, Fundamenta Inf. 85).
- Partial problem:
  detect language hiding in a complex grammar.
- Very difficult (Schmitz).

Observation: problems are associated with LL(1) violations.
Suggestion: detect LL(1) violations.
How: adapt known techniques to PEG.
General problem:
understand what this damned thing is doing.
Very difficult. (CS&P 2007, Fundamenta Inf. 85).
Partial problem:
detect language hiding in a complex grammar.
Very difficult (Schmitz).
How about just some hints where to look?

Observation: problems are associated with LL(1) violations.
Suggestion: detect LL(1) violations.
How: adapt known techniques to PEG.
General problem: understand what this damned thing is doing.
Very difficult. (CS&P 2007, Fundamenta Inf. 85).
Very difficult (Schmitz).
How about just some hints where to look?
Observation: problems are associated with LL(1) violations.
General problem:
understand what this damned thing is doing.
Very difficult. (CS&P 2007, Fundamenta Inf. 85).
Partial problem:
detect language hiding in a complex grammar.
Very difficult (Schmitz).
How about just some hints where to look?
Observation: problems are associated
with LL(1) violations.
Suggestion: detect LL(1) violations.
General problem:
understand what this damned thing is doing.

Very difficult. (CS&P 2007, Fundamenta Inf. 85).

Partial problem:
detect language hiding in a complex grammar.

Very difficult (Schmitz).

How about just some hints where to look?

Observation: problems are associated with LL(1) violations.

Suggestion: detect LL(1) violations.

How: adapt known techniques to PEG.
A known technique to check for LL(1) uses these relations:

- **FIRST**\((s)\) - set of possible first letters in a string derived from grammar symbol \(s\).
- **FOLLOW**\((s)\) - set of possible letters that can follow a string derived from grammar symbol \(s\).
Adapted to PEG:

- $\text{FIRST}(e)$ - set of terminals that may be invoked by expression $e$ on the start of input.
- $\text{FOLLOW}_s(e)$ - set of expressions that may be invoked after success of $e$.
- $\text{FOLLOW}_f(e)$ - set of expressions that may be invoked after failure of $e$. 
Disjoint expressions $e_1$ and $e_2$: terminals from FIRST($e_1$) and FIRST($e_2$) cannot succeed on the same input.
Disjoint expressions $e_1$ and $e_2$: terminals from $\text{FIRST}(e_1)$ and $\text{FIRST}(e_2)$ cannot succeed on the same input.

Example:

$$
\begin{align*}
  e_1 &= "abc" [a-z]^* & \text{FIRST}(e_1) &= \{"abc"\}, \\
  e_2 &= "abd" [a-z]^* & \text{FIRST}(e_2) &= \{"abd"\}, \\
  e_3 &= [a-z][a-z]^* & \text{FIRST}(e_3) &= \{[a-z]\}
\end{align*}
$$

$e_1$ and $e_2$ are disjoint. $e_2$ and $e_3$ are not.
Disjoint choice $e_1 / \ldots / e_n$: all $e_1, \ldots, e_n$ are pairwise disjoint.
Main result

Disjoint choice $e_1/\ldots/e_n$: all $e_1,\ldots,e_n$ are pairwise disjoint.

- Language hiding does not occur in a disjoint choice.
  - We can flag non-disjoint choices for examination.
Main result

Disjoint choice $e_1 / \ldots / e_n$: all $e_1, \ldots, e_n$ are pairwise disjoint.

1. Language hiding does not occur in a disjoint choice.
   - We can flag non-disjoint choices for examination.

2. If any of $e_1, \ldots, e_n$ in a disjoint choice fails after succeeding with at least one terminal, no terminal will succeed on that input. (Until the parser backtracks and takes another try.)
Main result

Disjoint choice $e_1 / \ldots / e_n$: all $e_1, \ldots, e_n$ are pairwise disjoint.

1. Language hiding does not occur in a disjoint choice.
   - We can flag non-disjoint choices for examination.

2. If any of $e_1, \ldots, e_n$ in a disjoint choice fails after succeeding with at least one terminal, no terminal will succeed on that input. (Until the parser backtracks and takes another try.)
   - We can stop trying other alternatives.
   This a PEG version of predictive parsing.
   (Mizushima, Meada & Yamaguchi)
To handle special cases (e.g. expressions consuming empty string), we need to involve FOLLOW\textsubscript{s} and FOLLOW\textsubscript{f}. There is a similar theory for star expressions that uses FOLLOW\textsubscript{s}.
To handle special cases (e.g. expressions consuming empty string), we need to involve $\text{FOLLOW}_s$ and $\text{FOLLOW}_f$.

There is a similar theory for star expressions that uses $\text{FOLLOW}_s$. 

There is more to it...
To handle special cases (e.g. expressions consuming empty string), we need to involve $\text{FOLLOW}_s$ and $\text{FOLLOW}_f$.

There is a similar theory for star expressions that uses $\text{FOLLOW}_s$.

But this is a long story...
See CS&P 2008, Fundamenta Inf. 93.
⊕ Relations $\text{FIRST}$, $\text{FOLLOW}_s$, $\text{FOLLOW}_t$, and disjointness are easy to compute using bit vectors and bit matrices.
Everything fine?

⊕ Relations FIRST, FOLLOW$_s$, FOLLOW$_t$, and disjointness are easy to compute using bit vectors and bit matrices.

⊕ Good news: experiment with a large grammar (Java 1.6) found 264 of 329 choice and star expressions to be disjoint.

Bad news: most of the remaining 65 are false alarms. Let us see why.
Relations FIRST, FOLLOW$_s$, FOLLOW$_f$, and disjointness are easy to compute using bit vectors and bit matrices.

Good news: experiment with a large grammar (Java 1.6) found 264 of 329 choice and star expressions to be disjoint.

Bad news: most of the remaining 65 are false alarms.
Relations $\text{FIRST}_s$, $\text{FOLLOW}_s$, $\text{FOLLOW}_f$, and disjointness are easy to compute using bit vectors and bit matrices.

Good news: experiment with a large grammar (Java 1.6) found 264 of 329 choice and star expressions to be disjoint.

Bad news: most of the remaining 65 are false alarms.

Let us see why.
Lookahead expression: \( !e \) where \( e \) is any expression.
For example: \( !"abc" \).
Trouble with lookahead

Lookahead expression: \( !e \) where \( e \) is any expression.

For example: \( !"abc" \).

It means:
- **Call** "abc".
- If it succeeds, backtrack and report failure.
- Otherwise report success.
Trouble with lookahead

Lookahead expression: !e where e is any expression.

For example: ! "abc".

It means:
- Call "abc".
- If it succeeds, backtrack and report failure.
- Otherwise report success.

In other words:
- Make sure the input does not start with abc.
- But do not consume anything.
Trouble with lookahead

Lookahead expression: !\(e\) where \(e\) is any expression.
For example: ! "abc".

It means:
- Call "abc".
- If it succeeds, backtrack and report failure.
- Otherwise report success.

In other words:
- Make sure the input does not start with abc.
- But do not consume anything.
- "abc" is included in FIRST.
Consider

\[ e_1 = (!"abc") [a-z]^*, \]
\[ e_2 = "abc" [a-z]^*. \]
Consider

\[ e_1 = (!"abc") [a-z]^*, \]
\[ e_2 = "abc" [a-z]^*. \]

\(e_1\) consumes strings of letters that do not start with \(abc\).
\(e_2\) consumes strings of letters that do start with \(abc\).
They never succeed on the same input.
Consider

\[ e_1 = (!"abc") [a-z]^*, \]
\[ e_2 = "abc" [a-z]^*. \]

\( e_1 \) consumes strings of letters that do not start with \( \text{abc} \).
\( e_2 \) consumes strings of letters that do start with \( \text{abc} \).

They never succeed on the same input.

\[
\text{FIRST}(e_1) = \{ "\text{abc}" , [a-z] \} \text{ (yes, } e_1 \text{ tries both),}
\text{FIRST}(e_2) = \{ "\text{abc}" \}.
\]
Consider

e_1 = ( ! "abc" ) [a-z] *,
e_2 = "abc" [a-z] *.

_ e_1 consumes strings of letters that do not start with abc. 
_ e_2 consumes strings of letters that do start with abc. 

They never succeed on the same input.

FIRST(e_1) = \{ "abc", [a-z] \} (yes, e_1 tries both),
FIRST(e_2) = \{ "abc" \}.

They are flagged as non-disjoint.
What is wrong?

$$\text{FIRST}(e_1) = \{ "abc", [a-z]\}$$ is clearly too big.

Leaving "abc" out does not help:

$$\text{FIRST}(e_1) = \{ [a-z]\}$$ and $$\text{FIRST}(e_2) = \{ "abc"\}$$ are still not disjoint.

We need something like $$\text{FIRST}(e_1) = \{ [a-z]\}$$ but not "abc".

Unfortunately, this does not work in general. We need something new.
What is wrong?

FIRST($e_1$) = \{"abc", [a-z]\} is clearly too big.

Only [a-z] is called to really bite off a piece of input, while "abc" is trying to prevent this.
What is wrong?

\[ \text{FIRST}(e_1) = \{"abc", [a-z]\} \] is clearly too big.

Only \([a-z]\) is called to really bite off a piece of input, while \"abc\" is trying to prevent this.

Leaving \"abc\" out does not help: \n\[ \text{FIRST}(e_1) = \{[a-z]\} \] and \[ \text{FIRST}(e_2) = \{"abc\}\] are still not disjoint.
What is wrong?

FIRST($e_1$) = \{"abc", [a-z]\} is clearly too big.

Only [a-z] is called to really bite off a piece of input, while "abc" is trying to prevent this.

Leaving "abc" out does not help: FIRST($e_1$) = \{[a-z]\} and FIRST($e_2$) = \{"abc"\}

are still not disjoint.

We need something like FIRST($e_1$) = \{[a-z] but not "abc"\}. 

Roman R. Redziejowski
FIRST and FOLLOW for PEG
What is wrong?

FIRST($e_1$) = \{"abc", [a–z]\} is clearly too big.

Only [a–z] is called to really bite off a piece of input, while "abc" is trying to prevent this.

Leaving "abc" out does not help: FIRST($e_1$) = \{[a–z]\} and FIRST($e_2$) = \{"abc\}\} are still not disjoint.

We need something like FIRST($e_1$) = \{[a–z] but not "abc"\}. Unfortunately, this does not work in general. We need something new.
Define "e bites s" to mean "a terminal called by e, otherwise than via a lookahead, consumes a prefix of s". (In other words, e takes the first real step to consume s.)
Define "e bites s" to mean "a terminal called by e, otherwise than via a lookahead, consumes a prefix of s". (In other words, e takes the first real step to consume s.)

Example:

\[ [a-zA-Z]^* \text{ bites any string in } [a-zA-Z] \Sigma^*. \]
Define "e bites s" to mean "a terminal called by e, otherwise than via a lookahead, consumes a prefix of s". (In other words, e takes the first real step to consume s.)

Example:

\[
[a-z]^* \text{ bites any string in } [a-z] \Sigma^*.
\]

"abc" [a-z]^* bites any string in "abc" \Sigma^*.
Define "e bites s" to mean "a terminal called by e, otherwise than via a lookahead, consumes a prefix of s". (In other words, e takes the first real step to consume s.)

Example:

\[ [a-z]^* \text{ bites any string in } [a-z] \Sigma^*. \]

"abc" \[a-z]^* \text{ bites any string in } "abc" \Sigma^*. \]

(!"abc") \[a-z]^* \text{ bites any string in } "abc" \Sigma^* \cap [a-z] \Sigma^*. \]
Define \( \text{BITES}(e) \) as a set of strings that \( e \) may bite:

\[
e \text{ bites } s \Rightarrow s \in \text{BITES}(e).
\]
Define BITES(e) as a set of strings that e may bite:

\[ e \text{ bites } s \implies s \in \text{BITES}(e). \]

Examples:

\[
\text{BITES}([a-z]^*) = [a-z]\Sigma^*.
\]
Define BITES(e) as a set of strings that e may bite: 
\[ e \text{ bites } s \implies s \in \text{BITES}(e). \]

Examples:
\[ \text{BITES}([a-z]^*) = [a-z]\Sigma^*. \]
\[ \text{BITES}("abc" [a-z]^*) = "abc"\Sigma^*. \]
Define BITES(e) as a set of strings that e may bite:

\[ e \text{ bites } s \implies s \in \text{BITES}(e). \]

Examples:

\[
\text{BITES}([a-z]^*) = [a-z] \Sigma^*. \\
\text{BITES}("abc" [a-z]*) = "abc" \Sigma^*. \\
\text{BITES}( !"abc" ) [a-z]^*) = \underbrace{"abc" \Sigma^*} \cap [a-z] \Sigma^*. 
\]
BITES(e₁) ∩ BITES(e₂) = ∅ means:

e₁ and e₂ cannot both bite the same string.
BITES(e_1) \cap BITES(e_2) = \emptyset\ means:\

\(e_1\) and \(e_2\) cannot both bite the same string.

Redefine "\(e_1\) and \(e_2\) disjoint" to mean

BITES(e_1) \cap BITES(e_2) = \emptyset.
BITES\( (e_1) \cap BITES(e_2) = \emptyset \) means:
e_1 and e_2 cannot both bite the same string.

Redefine "e_1 and e_2 disjoint" to mean
BITES\( (e_1) \cap BITES(e_2) = \emptyset \).

"abc" [a-z]* and (!"abc") [a-z]* are now disjoint!
Redefine "$e_1 \ldots / e_n$ disjoint" to mean "$e_1, \ldots, e_n$ are pairwise disjoint in the new sense."
Redefine "$e_1/\ldots/e_n$ disjoint" to mean
"$e_1, \ldots, e_n$ are pairwise disjoint in the new sense."

1. Language hiding does not occur in a disjoint choice.

2. If any of $e_1, \ldots, e_n$ in a disjoint choice fails after biting the input, nothing will bite that input. (Until the parser backtracks and takes another try.)
The lookahead is still a problem.

\[
\text{BITES}((!e_1)e_2) = \overline{\text{SUCC}(e_1)} \cap \text{BITES}(e_2)
\]

where SUCC\((e_1)\) should be the set of strings on which \(e_1\) succeeds.
The lookahead is still a problem.

\[
\text{BITES}((\neg e_1)e_2) = \text{SUCC}(e_1) \cap \text{BITES}(e_2)
\]

where \(\text{SUCC}(e_1)\) should be the set of strings on which \(e_1\) succeeds.

Finding \(\text{SUCC}(e)\) for arbitrary \(e\) is difficult.

It is about \(e\) succeeding on \(s\), not just biting it. And remember, it may depend on input far ahead. (Back to square one?)
SUCC sucks

It is possible to find SUCC\((e)\) if \(e\) is an expression on terminals.
Which is useful in many cases.
It is possible to find SUCC(e) if e is an expression on terminals. Which is useful in many cases.

Otherwise we can approximate SUCC "from below", by $\overline{\text{SUCC}}(e) \subseteq \text{SUCC}(e)$.

(We have to preserve "e bites s $\Rightarrow$ s $\in$ BITES(s)".)
It is possible to find SUCC(e) if e is an expression on terminals. Which is useful in many cases.

Otherwise we can approximate SUCC "from below", by \( \widetilde{\text{SUCC}}(e) \subseteq \text{SUCC}(e) \).

(We have to preserve "e bites s \( \Rightarrow \) s \( \in \) BITES(s)".)

One such approximation is \( \widetilde{\text{SUCC}}(e) = \emptyset \) which gives \( \text{BITES}((!e_1)e_2) = \text{BITES}(e_2) \), loosing all info on \( e_1 \). Not good, but I do not see any better yet.
Implementation sucks

Implementation is more complicated than with FIRST.

Instead of sets, we have regular expressions with Boolean operations. Of course, the emptiness problem for such expressions is decidable, but standard procedures are cumbersome with a large alphabet.
Implementation sucks

Implementation is more complicated than with FIRST.

Instead of sets, we have regular expressions with Boolean operations.
Implementation is more complicated than with FIRST.

Instead of sets, we have regular expressions with Boolean operations.

Of course, the emptiness problem for such expressions is decidable, but standard procedures are cumbersome with a large alphabet.
Conclusions

1. BITES is better than FIRST, but still not perfect.

2. BITES is more difficult to implement, but this is one-off, not run-time, analysis.

3. There is still much left to be detected.
What next

1. Implement and see how it works?

2. Forget it?

3. More research? (Need something for CSP 2011... )
Thanks for your attention!