Relations FIRST and FOLLOW for Parsing Expression Grammar

Roman R. Redziejowski

CS&P 2010

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)

Parsing Expression Grammar (PEG)

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

Parsing Expression Grammar (PEG)

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

Not a grammar: a recursive-descent parser.

Parsing Expression Grammar (PEG)

```
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").

Parsing Expression Grammar (PEG)

```
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".

Parsing Expression Grammar (PEG)

```
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).

PEG in action

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

PEG in action

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

          29.165
          ^
```

PEG in action

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

```
        29.165
        ^
```

```
number
```

PEG in action

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

          29.165
          ^

number->real
```

PEG in action

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

          29.165
          ^

number->real->digits
```

PEG in action

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

          29.165
          ^

number->real->digits->[0-9][0-9]*
```

PEG in action

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

          29.165
            ^

number->real->digits->[0-9][0-9]*: consume "29"
```

PEG in action

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

             29.165
               ^

number->real->digits
```

PEG in action

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

           29.165
              ^

number->real
```

PEG in action

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

          29.165
            ^

number->real->"."
```

PEG in action

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

            29.165
               ^

number->real->".": consumes "."
```

PEG in action

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

```
          29.165
             ^
```

```
number->real
```

PEG in action

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

           29.165
              ^

number->real->digits
```

PEG in action

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

         29.165
              ^

number->real->digits: consume "165"
```

PEG in action

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

```
          29.165
               ^
```

```
number->real
```

PEG in action

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

           29.165
                 ^

number
```

PEG in action

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

```
          29.165
                ^
```

PEG in action: backtracking

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

PEG in action: backtracking

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

          4711
          ^
```

PEG in action: backtracking

```
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
          ^

number->real
```

PEG in action: backtracking

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

          4711
          ^

number->real->digits
```

PEG in action: backtracking

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

         4711
             ^

number->real->digits: consume "4711"
```

PEG in action: backtracking

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

         4711
             ^

number->real
```

PEG in action: backtracking

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

          4711
              ^

number->real->"."
```

PEG in action: backtracking

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

          4711
              ^

number->real->".":  returns failure
```

PEG in action: backtracking

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

          4711
          ^

number->real: backtracks
```

PEG in action: backtracking

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

          4711
          ^

number->real: returns failure
```

PEG in action: backtracking

```
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
          ^

number->integer
```

PEG in action: backtracking

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

          4711
          ^

number->integer->digits
```

PEG in action: backtracking

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

          4711
              ^

number->integer->digits: consume "4711"
```

PEG in action: backtracking

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

           4711
               ^

number->integer
```

PEG in action: backtracking

```
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]*
```

Backtracking is limited

```
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
             ^

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
```

Backtracking is limited

```
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

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

```
          29.165
            ^
```

```
number
```

Backtracking is limited

```
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

```
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`.

`integer` hides part of the language of `real`.

PEG is not EBNF

All of these fail on input `aab`:

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

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

("a"/"c"?)"aab"
```

PEG is not EBNF

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.

Some fun

Guess what this is doing:

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

Some fun

Guess what this is doing:

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

```
aaaa       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
```

Some fun

Guess what this is doing:

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

```
aaaa         consumes 4 of 4
aaaaa                 2 of 5
aaaaaa                4 of 6
```

Some fun

Guess what this is doing:

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

```
aaaa         consumes 4 of 4
aaaaa                 2 of 5
aaaaaa                4 of 6
aaaaaaa               6 of 7
```

Some fun

Guess what this is doing:

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

```
aaaa        consumes 4 of 4
aaaaa                2 of 5
aaaaaa               4 of 6
aaaaaaa              6 of 7
aaaaaaaa             8 of 8
```

Some fun

Guess what this is doing:

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

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

Some fun

Guess what this is doing:

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

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

Result depends on input far ahead.

Some fun

Guess what this is doing:

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

aaaa        consumes 4 of 4
aaaaa                2 of 5
aaaaaa               4 of 6
aaaaaaa              6 of 7
aaaaaaaa             8 of 8
aaaaaaaaa            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.

Problem

- 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.

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).

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).
- How about just some hints where to look?

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).
- How about just some hints where to look?
- Observation: problems are associated
  with LL(1) violations.

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).
- How about just some hints where to look?
- Observation: problems are associated
  with LL(1) violations.
- Suggestion: detect LL(1) violations.

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).
- 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.

Classical FIRST and FOLLOW

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.

FIRST and FOLLOW for PEG

Adapted to PEG:

- 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

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 FIRST(e_1) and FIRST(e_2)
cannot succeed on the same input.

Example:

$e_1 = \texttt{"abc"[a-z]}^*$ FIRST(e_1) $= \{\texttt{"abc"}\}$,
$e_2 = \texttt{"abd"[a-z]}^*$ FIRST(e_2) $= \{\texttt{"abd"}\}$,
$e_3 = \texttt{[a-z][a-z]}^*$ FIRST(e_3) $= \{\texttt{[a-z]}\}$

e_1 and e_2 are disjoint.
e_2 and e_3 are not.

Main result

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.

1. 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 $e_1 / \dots / e_n$: all $e_1, \dots, e_n$ 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 $e_1, \dots, e_n$ 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)

There is more to it...

To handle special cases (e.g. expressions consuming empty string), we need to involve FOLLOW_s and FOLLOW_f.

There is more to it...

To handle special cases (e.g. expressions consuming empty string), we need to involve FOLLOW_s and FOLLOW_f.

There is a similar theory for star expressions that uses FOLLOW_s.

There is more to it...

To handle special cases (e.g. expressions consuming empty string), we need to involve FOLLOW_s and FOLLOW_f.

There is a similar theory for star expressions that uses FOLLOW_s.

But this is a long story...
See CS&P 2008, Fundamenta Inf. 93.

Everything fine?

⊕ Relations FIRST, FOLLOW_s, FOLLOW_f, and disjointness are easy to compute using bit vectors and bit matrices.

Everything fine?

⊕ 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.

Everything fine?

⊕ 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.

Everything fine?

⊕ 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.

Let us see why.

Trouble with lookahead

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.

Trouble with lookahead

Consider

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

Trouble with lookahead

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.

Trouble with lookahead

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"`}.

Trouble with lookahead

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.

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?

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.

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"}.

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:
$\text{FIRST}(e_1) = \{$`[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.

Aggresive expressions

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*.)

Aggresive expressions

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]`* bites any string in `[a-z]`Σ^*.

Aggresive expressions

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]*` bites any string in `[a-z]`Σ^*.

`"abc"[a-z]*` bites any string in `"abc"`Σ^*.

Aggresive expressions

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]`* bites any string in `[a-z]`Σ^*.

`"abc"[a-z]`* bites any string in `"abc"`Σ^*.

`(!"abc")[a-z]`* bites any string in $\overline{\texttt{"abc"}\Sigma^*} \cap$ `[a-z]`Σ^*.

BITES instead of FIRST

Define BITES(e) as a set of strings that e may bite:

e bites $s \;\Rightarrow\; s \in \text{BITES}(e)$.

BITES instead of FIRST

Define BITES(e) as a set of strings that e may bite:

e bites $s \Rightarrow s \in \text{BITES}(e)$.

Examples:

$\text{BITES}(\texttt{[a-z]}^*) = \texttt{[a-z]}\Sigma^*$.

BITES instead of FIRST

Define BITES(e) as a set of strings that e may bite:

e bites $s \Rightarrow s \in$ BITES(e).

Examples:

BITES($\texttt{[a-z]}^*$) $= \texttt{[a-z]}\Sigma^*$.

BITES($\texttt{"abc"[a-z]}^*$) $= \texttt{"abc"}\Sigma^*$.

BITES instead of FIRST

Define BITES(e) as a set of strings that e may bite:

e bites $s \Rightarrow s \in \text{BITES}(e)$.

Examples:

$\text{BITES}(\texttt{[a-z]*}) = \texttt{[a-z]}\Sigma^*$.

$\text{BITES}(\texttt{"abc"[a-z]*}) = \texttt{"abc"}\Sigma^*$.

$\text{BITES}(\texttt{(!"abc")[a-z]*}) = \overline{\texttt{"abc"}\Sigma^*} \cap \texttt{[a-z]}\Sigma^*$.

New disjointness

$\text{BITES}(e_1) \cap \text{BITES}(e_2) = \varnothing$ means:

e_1 and e_2 cannot both bite the same string.

New disjointness

$\text{BITES}(e_1) \cap \text{BITES}(e_2) = \varnothing$ means:

e_1 and e_2 cannot both bite the same string.

Redefine "e_1 and e_2 disjoint" to mean

$\text{BITES}(e_1) \cap \text{BITES}(e_2) = \varnothing$.

New disjointness

$\text{BITES}(e_1) \cap \text{BITES}(e_2) = \varnothing$ means:

e_1 and e_2 cannot both bite the same string.

Redefine "e_1 and e_2 disjoint" to mean

$\text{BITES}(e_1) \cap \text{BITES}(e_2) = \varnothing$.

`"abc"[a-z]*` and `(!"abc")[a-z]*` are now disjoint!

Updated main results

Redefine "$e_1 / \ldots / e_n$ disjoint" to mean

"$e_1, \ldots, e_n$ are pairwise disjoint in the new sense."

Updated main results

Redefine "$e_1 / \dots / e_n$ disjoint" to mean

"$e_1, \dots, e_n$ are pairwise disjoint in the new sense."

1. Language hiding does not occur in a disjoint choice.
2. If any of $e_1, \dots, e_n$ in a disjoint choice fails after biting the input, nothing will bite that input. (Until the parser backtracks and takes another try.)

Everything fine? Not really...

The lookahead is still a problem.

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

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

Everything fine? Not really...

The lookahead is still a problem.

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

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.

SUCC sucks

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 $s \Rightarrow s \in \text{BITES}(s)$".)

SUCC sucks

It is possible to find $\text{SUCC}(e)$ 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 $s \Rightarrow s \in \text{BITES}(s)$".)

One such approximation is $\widetilde{\text{SUCC}}(e) = \varnothing$

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.

Implementation sucks

Implementation is more complicated than with FIRST.

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

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.

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...)

That's all folks ...

Thanks for your attention!