Top-down parsing with backtrack

Roman Redziejowski

Concurrency, Specification and Programming
Rostock 2016

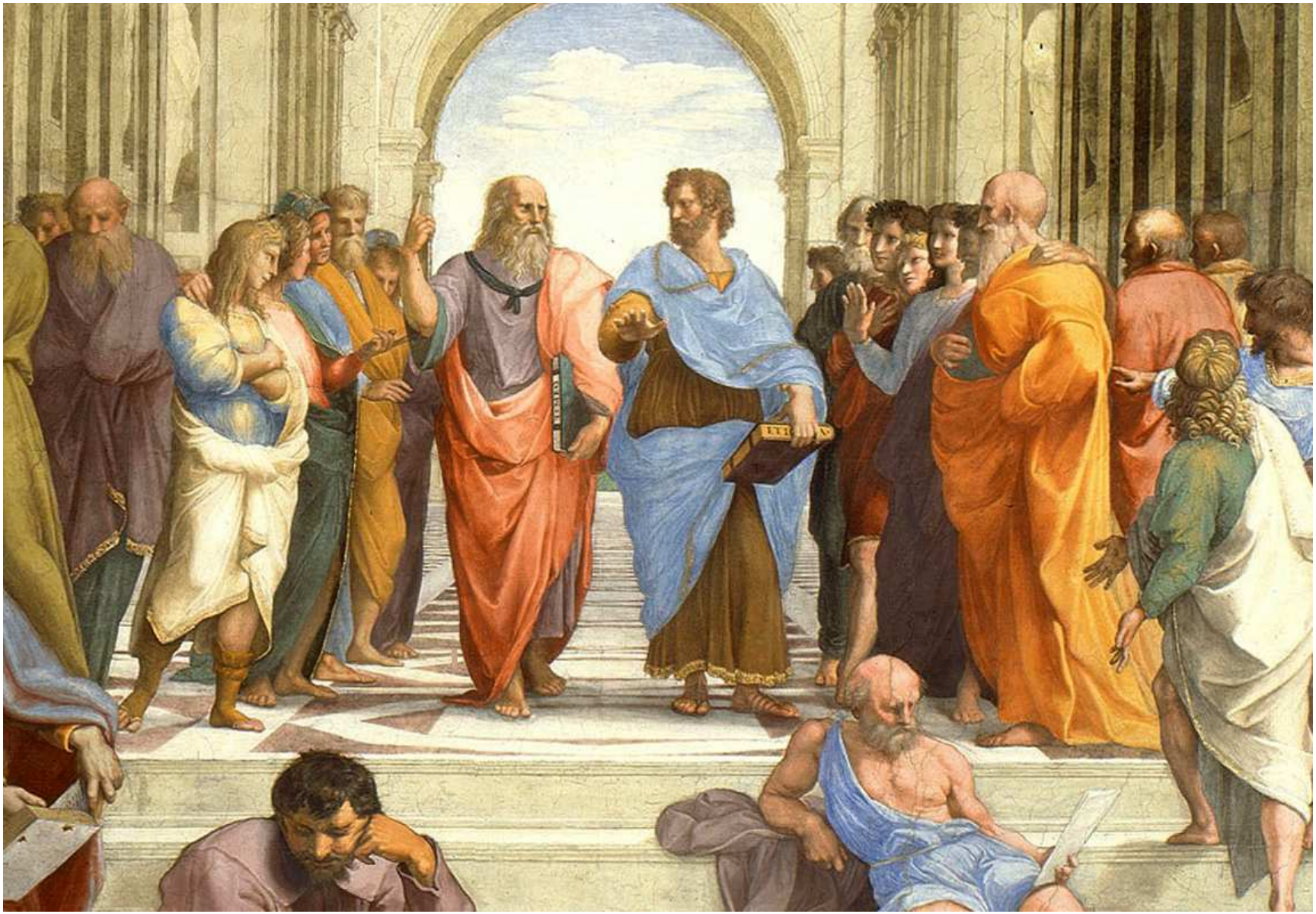
A long time ago

A long time ago ...

Perhaps in the middle of XX century ...

A group of ancient philosophers ...

A long time ago



A long time ago

... decided to create a Programming Language

to end all programming languages:

ALGOL!

A long time ago

Syntax definition of Algol 60

```
<Expr>     ::= <Term> | <Expr> <AddOp> <Term>
<Term>     ::= <Factor> | <Term> <MultOp> <Factor>
<Factor>   ::= <Float> | <Integer> | ( <Expr> )
<Integer>  ::= <Digits>
<Float>    ::= . <Digits> | <Digits> . <Digits>
<AddOp>    ::= + | -
<MultOp>   ::= * | /
<Digits>   ::= <Digit> | <Digits> <Digit>
<Digit>    ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

A long time ago

Syntax definition of Algol 60

```
<Expr>     ::= <Term> | <Expr> <AddOp> <Term>
<Term>     ::= <Factor> | <Term> <MultOp> <Factor>
<Factor>   ::= <Float> | <Integer> | ( <Expr> )
<Integer>  ::= <Digits>
<Float>    ::= . <Digits> | <Digits> . <Digits>
<AddOp>    ::= + | -
<MultOp>   ::= * | /
<Digits>   ::= <Digit> | <Digits> <Digit>
<Digit>    ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Designed by

John Backus
Peter Naur

and called Backus-Naur Form (BNF).

EBNF: Extended Backus-Naur Form

Algol is dead, but BNF is much alive and got a facelift

EBNF: Extended Backus-Naur Form

Algol is dead, but BNF is much alive and got a facelift

```
Expr    = Term (AddOp Term)*
Term    = Factor (MultOp Factor)*
Factor  = Float | Integer | "(" Expr ")"
Integer = Digits
Float   = Digits? "." Digits
AddOp   = [+-]
MultOp  = [*/]
Digits  = Digit+
Digit   = [0-9]
```

EBNF: Extended Backus-Naur Form

Algol is dead, but BNF is much alive and got a facelift

```
Expr    = Term (AddOp Term)*
Term    = Factor (MultOp Factor)*
Factor  = Float | Integer | "(" Expr ")"
Integer = Digits
Float   = Digits? "." Digits
AddOp   = [+-]
MultOp  = [*/]
Digits  = Digit+
Digit   = [0-9]
```

Abbreviated notation

EBNF: Extended Backus-Naur Form

Algol is dead, but BNF is much alive and got a facelift

```
Expr    = Term (AddOp Term)*
Term    = Factor (MultOp Factor)*
Factor  = Float | Integer | "(" Expr ")"
Integer = Digits
Float   = Digits? "." Digits
AddOp   = [+-]
MultOp  = [*/]
Digits  = Digit+
Digit   = [0-9]
```

Abbreviated notation
Option instead of alternative

EBNF: Extended Backus-Naur Form

Algol is dead, but BNF is much alive and got a facelift

```
Expr    = Term (AddOp Term)*
Term    = Factor (MultOp Factor)*
Factor  = Float | Integer | "(" Expr ")"
Integer = Digits
Float   = Digits? "." Digits
AddOp   = [+-]
MultOp  = [*/]
Digits  = Digit+
Digit   = [0-9]
```

Abbreviated notation
Option instead of alternative
Repetition instead of recursion

EBNF: Extended Backus-Naur Form

You may have also seen this:

```
Expr    = Term {AddOp Term}
Term    = Factor {MultOp Factor}
Factor  = Float | Integer | "(" Expr ")"
Integer = Digits
Float   = [Digits] "." Digits
AddOp   = "+" | "-"
MultOp  = "*" | "/"
Digits  = Digit{Digit}
Digit   = one of 0 1 2 3 4 5 6 7 8 9
```

Parsing

```
Expr    = Term (AddOp Term)*
Term    = Factor (MultOp Factor)*
Factor  = Float | Integer | "(" Expr ")"
Integer = Digits
Float   = Digits? "." Digits
AddOp   = [+-]
MultOp  = [*/]
Digits  = Digit+
Digit   = [0-9]
```

```
5*(1.73/3+.2)
```

Parsing

```
Expr    = Term (AddOp Term)*
Term    = Factor (MultOp Factor)*
Factor  = Float | Integer | "(" Expr ")"
Integer = Digits
Float   = Digits? "." Digits
AddOp   = [+-]
MultOp  = [*/]
Digits  = Digit+
Digit   = [0-9]
```

`5*(1.73/3+.2)` How to parse this?

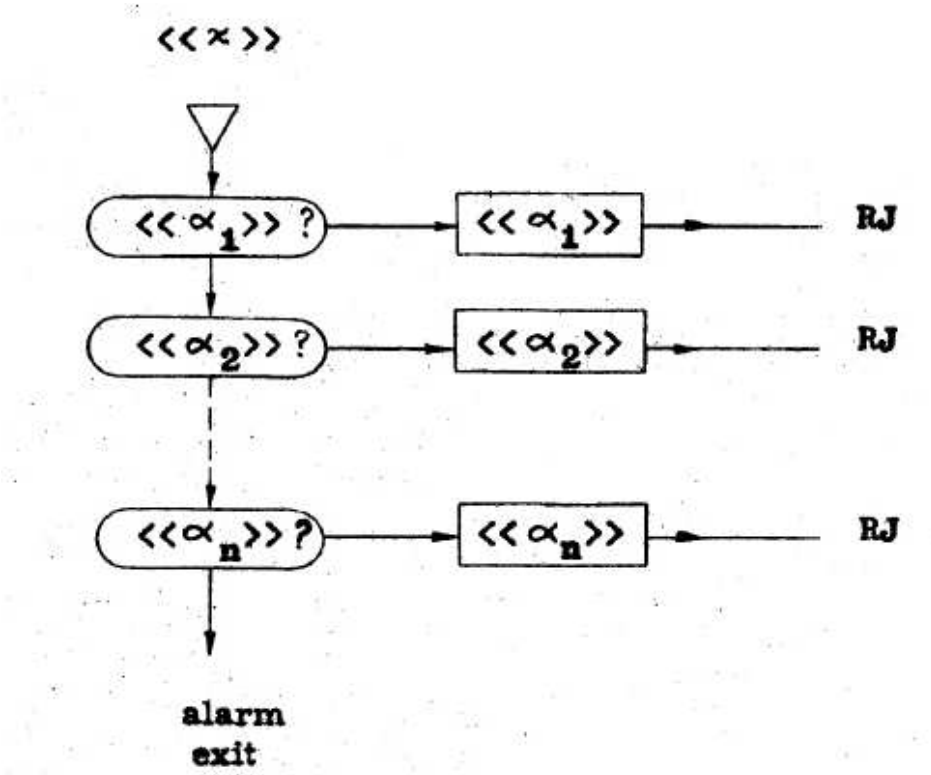
Parsing

```
Expr    = Term (AddOp Term)*
Term    = Factor (MultOp Factor)*
Factor  = Float | Integer | "(" Expr ")"
Integer = Digits
Float   = Digits? "." Digits
AddOp   = [+-]
MultOp  = [*/]
Digits  = Digit+
Digit   = [0-9]
```

`5*(1.73/3+.2)` How to parse this?

Peter Lucas:
For each syntactic unit write a procedure that "consumes" the unit.

Procedure for $\chi = \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$



(From ancient papyrus attributed to P. Lucas)

Recursive descent

```
Expr    = Term (AddOp Term)*
Term    = Factor (MultOp Factor)*
Factor  = Float | Integer | "(" Expr ")"
Integer = Digits
Float   = Digits? "." Digits
AddOp   = [+-]
MultOp  = [*/]
Digits  = Digit+
Digit   = [0-9]
```

```
5*(1.73/3+.2)
```

Recursive descent

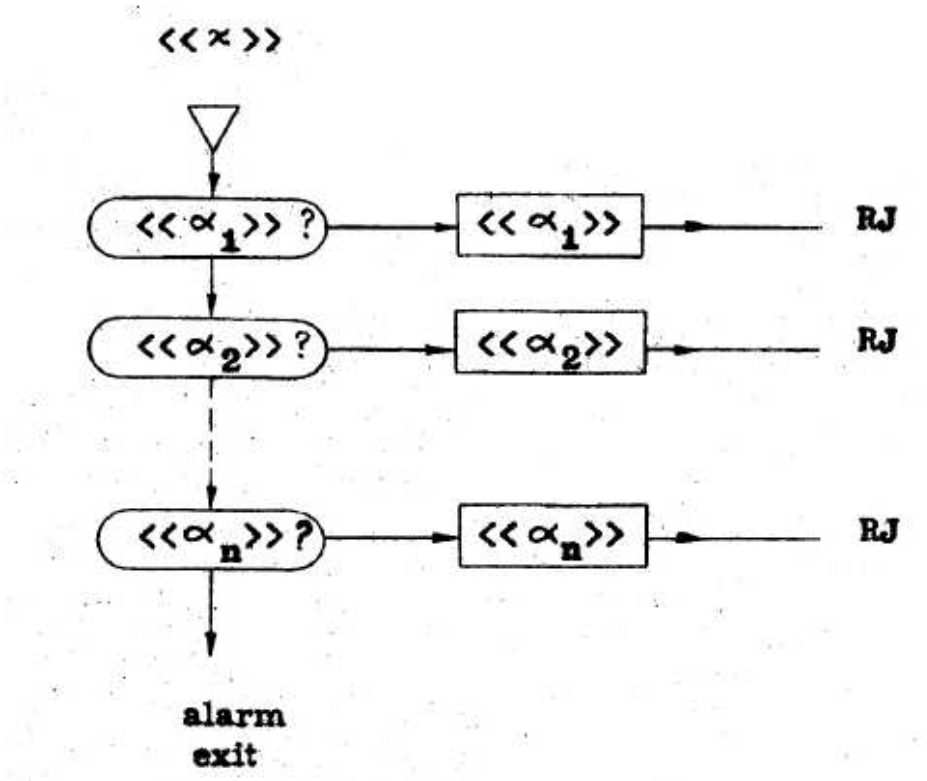
```
Expr    = Term (AddOp Term)*
Term    = Factor (MultOp Factor)*
Factor  = Float | Integer | "(" Expr ")"
Integer = Digits
Float   = Digits? "." Digits
AddOp   = [+-]
MultOp  = [*/]
Digits  = Digit+
Digit   = [0-9]
```

```
5*(1.73/3+.2)
```

Top-down parsing by recursive descent.

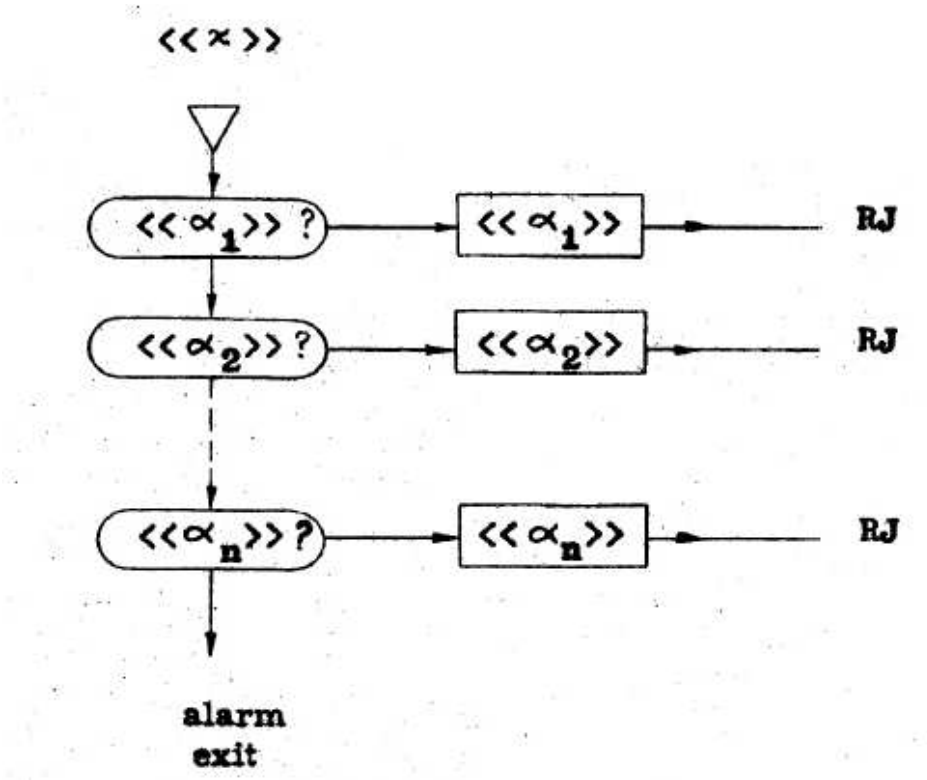
Recursive descent

Little problem



"Each metalinguistic variable must be distinguishable ...
... by the very first symbol of the string represented by the variable."

Little problem



"Each metalinguistic variable must be distinguishable ...
... by the very first symbol of the string represented by the variable."

The LL(1) property.

Another little problem

Left recursion:

A = A B | C

Infinite descent

Avoiding problems

Not LL(1): A = aB | aC

Avoiding problems

Not LL(1): A = aB | aC

Transform to A = a (B | C)

Avoiding problems

Not LL(1): A = aB | aC

Transform to A = a (B | C)

Left recursion: A = AB | C

Avoiding problems

Not LL(1): A = aB | aC

Transform to A = a (B | C)

Left recursion: A = AB | C

Transform to A = CB*

Manchester 1962

A monster: Manchester University Atlas
1 000 000 operations per second
1 000 000 words of addressable storage.

Manchester 1962

Brooker and Morris: Compiler Compiler

Manchester 1962

Brooker and Morris: Compiler Compiler

Took "phrase structure grammar" - very like EBNF.

Manchester 1962

Brooker and Morris: Compiler Compiler

Took "phrase structure grammar" - very like EBNF.

Constructed recursive-descent parser.

Manchester 1962

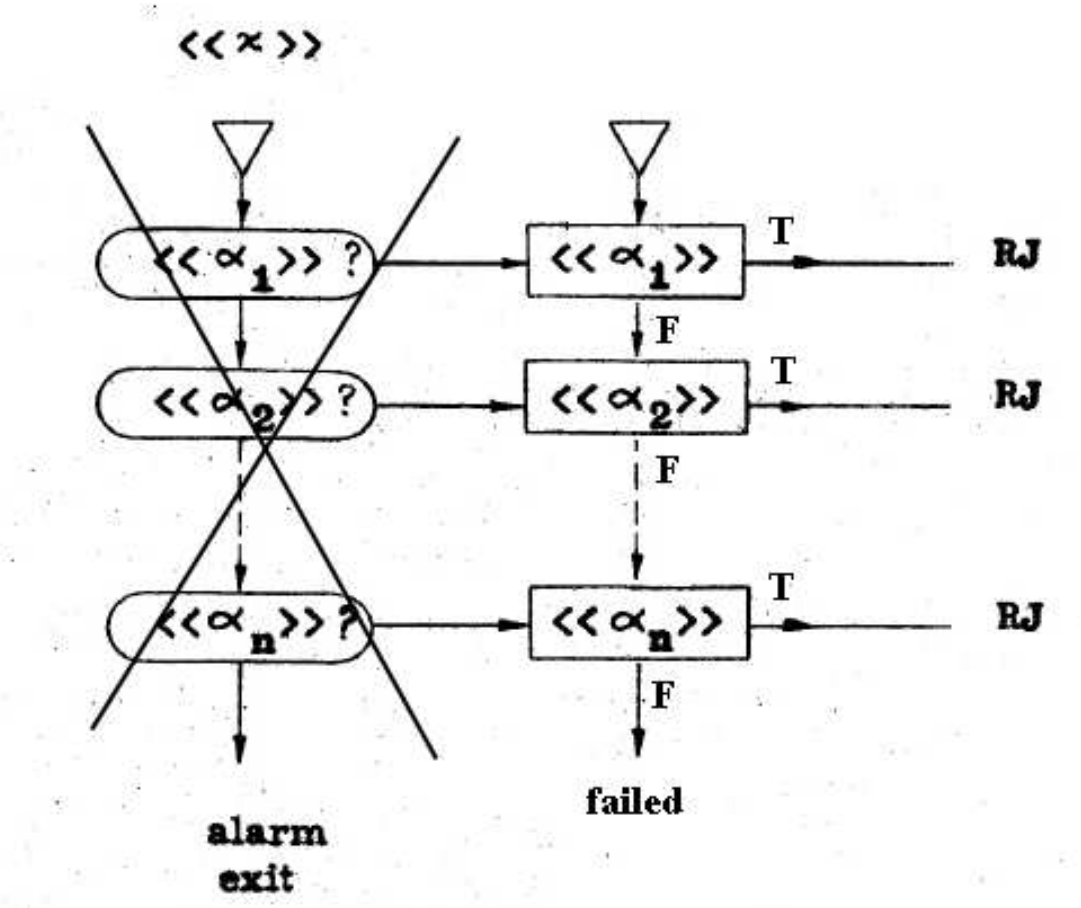
Brooker and Morris: Compiler Compiler

Took "phrase structure grammar" - very like EBNF.

Constructed recursive-descent parser.

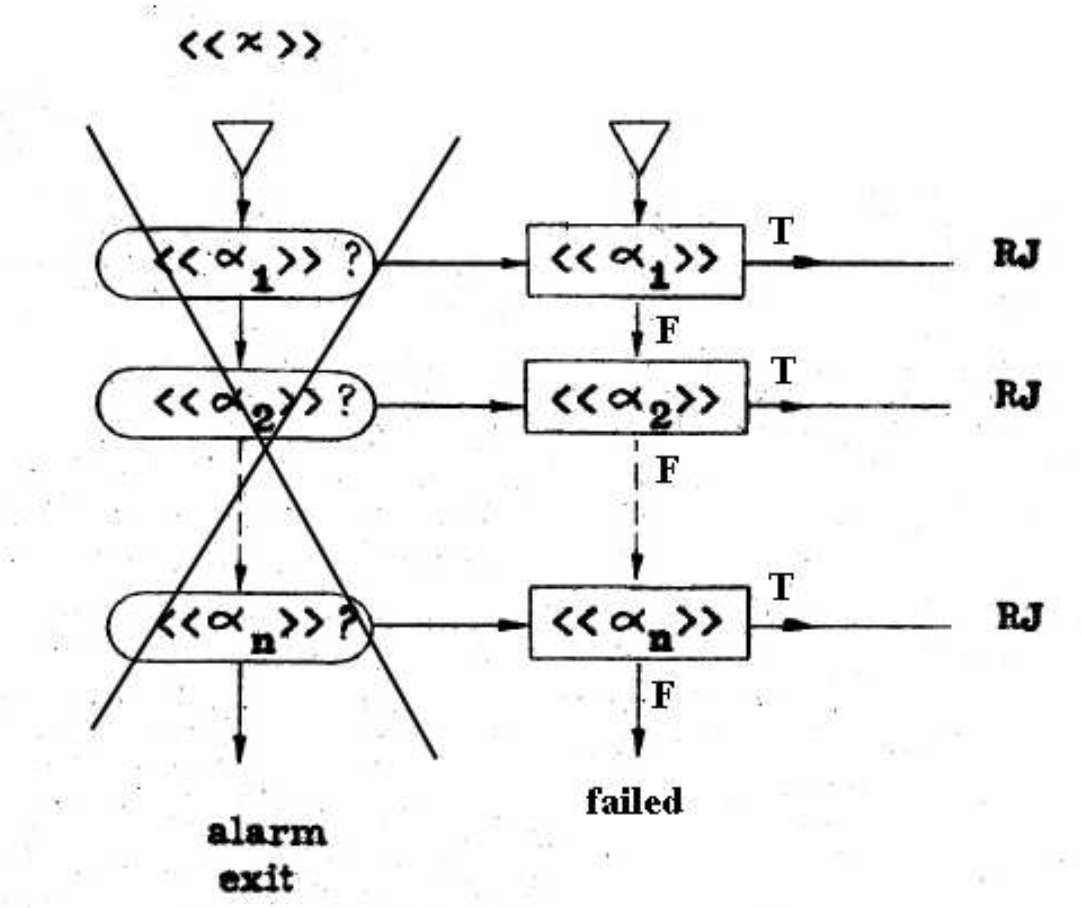
Solution to LL(1) problem:
Just try everything. Atlas can do it!

Backtracking



Parsing procedure tries to "consume" its syntactic unit.
On failure: rewinds the input - **backtracks**.

Limited backtracking



Once α_i succeeded, never try any of $\alpha_{i+1} \ldots \alpha_n$

Left-recursion?

Unfortunately, no medicine for left-recursion.

Atlas was only one in the crowd ...

Hopgood (1969) names four projects that use limited backtracking.

Atlas was only one in the crowd ...

Hopgood (1969) names four projects that use limited backtracking.

Another approach: LL(*). Uses a lookahead automaton (ANTLR).

Atlas was only one in the crowd ...

Hopgood (1969) names four projects that use limited backtracking.

Another approach: LL(*). Uses a lookahead automaton (ANTLR).

Some attempts to marry left-recursion and top-down.

Atlas was only one in the crowd ...

Hopgood (1969) names four projects that use limited backtracking.

Another approach: LL(*). Uses a lookahead automaton (ANTLR).

Some attempts to marry left-recursion and top-down.

Bottom-up parsing: LR, LALR (YACC).

Atlas was only one in the crowd ...

Hopgood (1969) names four projects that use limited backtracking.

Another approach: LL(*). Uses a lookahead automaton (ANTLR).

Some attempts to marry left-recursion and top-down.

Bottom-up parsing: LR, LALR (YACC).

Earley parsing.

Meanwhile in the US

1965: McClure from Texas Instruments.
TransMoGrifier (TMG). Rewrites syntax definition into a parser.

Meanwhile in the US

1965: McClure from Texas Instruments.
TransMoGrifier (TMG). Rewrites syntax definition into a parser.

1970: Ph.D. thesis of Alexander Birman.
Analyzes the grammar used by TMG.

Meanwhile in the US

1965: McClure from Texas Instruments.
TransMoGrifier (TMG). Rewrites syntax definition into a parser.

1970: Ph.D. thesis of Alexander Birman.
Analyzes the grammar used by TMG.

1972: Book by Alfred V. Aho and Jeffrey D. Ullman
(precursor of "Dragon Book").
A comprehensible account of Birman's findings.

TDPL

TDPL (Top-Down Parsing Language) - a language to write top-down parsers with limited backtracking.
Much like EBNF in the spirit of Lucas.

TDPL

TDPL (Top-Down Parsing Language) - a language to write top-down parsers with limited backtracking.
Much like EBNF in the spirit of Lucas.

New construction: A[B,C].
Call A. If successful, continue to call B.
Otherwise backtrack and call C.
Here A performs lookahead - unknown in EBNF.

TDPL

TDPL (Top-Down Parsing Language) - a language to write top-down parsers with limited backtracking.
Much like EBNF in the spirit of Lucas.

New construction: A[B,C].
Call A. If successful, continue to call B.
Otherwise backtrack and call C.
Here A performs lookahead - unknown in EBNF.

Can work in linear time.
But needs a lot of memory.
Too much for existing machines.

Fast forward 30 years

Packrat parsing

MIT 2002: M.Sc. thesis of Brian Ford uses the fact that TDPL can be processed in linear time.

Packrat parsing

MIT 2002: M.Sc. thesis of Brian Ford uses the fact
that TDPL can be processed in linear time.

The trick is to save all results so they can be reused after backtrack.
The new fantastic computers of the day can do it.

Packrat parsing

MIT 2002: M.Sc. thesis of Brian Ford uses the fact
that TDPL can be processed in linear time.

The trick is to save all results so they can be reused after backtrack.
The new fantastic computers of the day can do it.

Process is called "packrat parsing".
Pack rat (*neotoma cinerea*) is a rodent that collects lots of unnecessary things in its nest.
Also a person with such a compulsory behavior.

Packrat parsing

MIT 2002: M.Sc. thesis of Brian Ford uses the fact
that TDPL can be processed in linear time.

The trick is to save all results so they can be reused after backtrack.
The new fantastic computers of the day can do it.

Process is called "packrat parsing".
Pack rat (*neotoma cinerea*) is a rodent that collects lots of unnecessary things in its nest.
Also a person with such a compulsory behavior.

"Memoization" of parsing procedures.

Packrat parsing

MIT 2002: M.Sc. thesis of Brian Ford uses the fact
that TDPL can be processed in linear time.

The trick is to save all results so they can be reused after backtrack.
The new fantastic computers of the day can do it.

Process is called "packrat parsing".
Pack rat (*neotoma cinerea*) is a rodent that collects lots of unnecessary things in its nest.
Also a person with such a compulsory behavior.

"Memoization" of parsing procedures.

Pappy: written in Haskell
Generated packrat parsers written in Haskell.

PEG

2004: Ford announces a re-make of TDPL.
Calls it "Parsing Expression Grammar" (PEG).

Looks like EBNF:

```
Expr    <- Term (AddOp Term)*
Term    <- Factor (MultOp Factor)*
Factor  <- Float / Integer / "(" Expr ")"
Integer <- Digits
Float   <- Digits? "." Digits
AddOp   <- [+-]
MultOp  <- [*/]
Digits  <- Digit+
Digit   <- [0-9]
```

PEG

2004: Ford announces a re-make of TDPL.
Calls it "Parsing Expression Grammar" (PEG).

Looks like EBNF:

```
Expr    <- Term (AddOp Term)*
Term    <- Factor (MultOp Factor)*
Factor  <- Float / Integer / "(" Expr ")"
Integer <- Digits
Float   <- Digits? "." Digits
AddOp   <- [+-]
MultOp  <- [*/]
Digits  <- Digit+
Digit   <- [0-9]
```

- in reality a recursive-descent parser with limited backtracking.

All syntactic elements are Parsing Expressions:
recursive parsing procedures.

PEG

$E_1 / \ldots / E_n$	Ordered choice: Apply expressions $E_1, \ldots, E_n$, in this order, to the text ahead, until one of them succeeds and possibly consumes some text. Indicate success if one of expressions succeeded. Otherwise do not consume any text and indicate failure.
$E_1 \ldots E_n$	Sequence: Apply expressions $E_1, \ldots, E_n$, in this order, to consume consecutive portions of the text ahead, as long as they succeed. Indicate success if all succeeded. Otherwise do not consume any text and indicate failure.
$\&E$	And predicate: Indicate success if expression E matches the text ahead; otherwise indicate failure. Do not consume any text.
$!E$	Not predicate: Indicate failure if expression E matches the text ahead; otherwise indicate success. Do not consume any text.
E^+	One or more: Apply expression E repeatedly to match the text ahead, as long as it succeeds. Consume the matched text (if any) and indicate success if there was at least one match. Otherwise indicate failure.
E^*	Zero or more: Apply expression E repeatedly to match the text ahead, as long as it succeeds. Consume the matched text (if any). Always indicate success.
$E?$	Zero or one: If expression E matches the text ahead, consume it. Always indicate success.
$[s]$	Character class: If the character ahead appears in the string s, consume it and indicate success. Otherwise indicate failure.
$[c_1\text{-}c_2]$	Character range: If the character ahead is one from the range c_1 through c_2, consume it and indicate success. Otherwise indicate failure.
"s"	String: If the text ahead is the string s, consume it and indicate success. Otherwise indicate failure.
_	Any character: If there is a character ahead, consume it and indicate success. Otherwise (that is, at the end of input) indicate failure.

Syntactic predicates - lookahead

&E: Succeed if text ahead conforms to E.

!E: Succeed if text ahead does not conform to E.

In each case do not consume any input.

Syntactic predicates - lookahead

&E: Succeed if text ahead conforms to E.

!E: Succeed if text ahead does not conform to E.

In each case do not consume any input.

Can define:

!_ End of input

<!= "<" not followed by "="

(!a_)*a String of anything up to "a"

(a&a)* Consumes string of a's except the last one

Syntactic predicates - lookahead

&E: Succeed if text ahead conforms to E.

!E: Succeed if text ahead does not conform to E.

In each case do not consume any input.

Can define:

!_ End of input

<!= "<" not followed by "="

(!a_)*a String of anything up to "a"

(a&a)* Consumes string of a's except the last one

Can define $a^n b^n c^n$.

Syntactic predicates - lookahead

&E: Succeed if text ahead conforms to E.

!E: Succeed if text ahead does not conform to E.

In each case do not consume any input.

Can define:

!_ End of input

<!= "<" not followed by "="

(!a_)*a String of anything up to "a"

(a&a)* Consumes string of a's except the last one

Can define $a^n b^n c^n$.

Can be replaced by backtracking.

Integrated lexing

```
Expr    = Term (AddOp Term)*
Term    = Factor (MultOp Factor)*
Factor  = Float | Integer | Lparen Expr Rparen
Integer = Digits
Float   = Digits? "." Digits
AddOp   = [+-]
MultOp  = [*/]
Lparen  = "("
Rparen  = ")"
Digits  = Digit+
Digit   = [0-9]
```

Integrated lexing

```
Expr    = Term (AddOp Term)*
Term    = Factor (MultOp Factor)*
Factor  = Float | Integer | Lparen Expr Rparen
Integer = Digits Space
Float   = Digits? "." Digits Space
AddOp   = [+-] Space
MultOp  = [*/] Space
Lparen  = "(" Space
Rparen  = ")" Space
Digits  = Digit+
Digit   = [0-9]
Space   = (" " | "{"(!"}"_)*"}")*
```

Integrated lexing

```
Expr    = Term (AddOp Term)*
Term    = Factor (MultOp Factor)*
Factor  = Float | Integer | "(" Expr ")"
Integer = Digits Space
Float   = Digits? "." Digits Space
AddOp   = [+-] Space
MultOp  = [*/] Space
Lparen  = "(" Space
Rparen  = ")" Space
Digits  = Digit+
Digit   = [0-9]
Space   = (" " | "{"(!"}"_)*"}")*
```

```
Keyword = ("one"|"two") !Letter
Identifier = !Keyword Letter+
Letter = [a-z]
```

PEG fans

Ultimate beauty: syntax and complete parser in one!

PEG fans

Ultimate beauty: syntax and complete parser in one!

PEG fan club:

Discussion forum

Publication list.

Rats! and Mouse

2004: Rats! by Robert Grimm.
Parser generator.
Generates packrat parsers from PEG.
Both generator and parser in Java.

Rats! and Mouse

2004: Rats! by Robert Grimm.
Parser generator.
Generates packrat parsers from PEG.
Both generator and parser in Java.

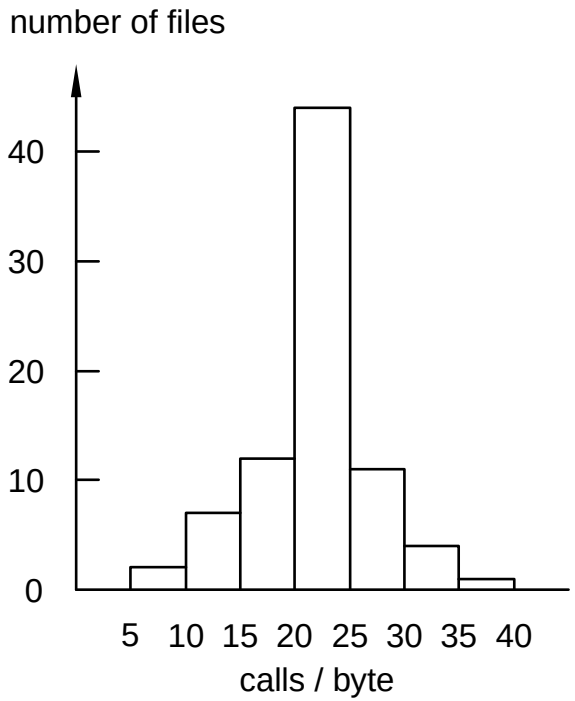
2006: Mouse.
My program in Java to transcribe Parsing Expressions into parsing procedures in Java. Not a pack rat.

Measuring performance

Wrote PEG for Java 1.6 and generated parser for it.
Applied the parser to 80 Java files of sizes from 0.1 to 476 kBytes.
Counted calls to parsing procedures.

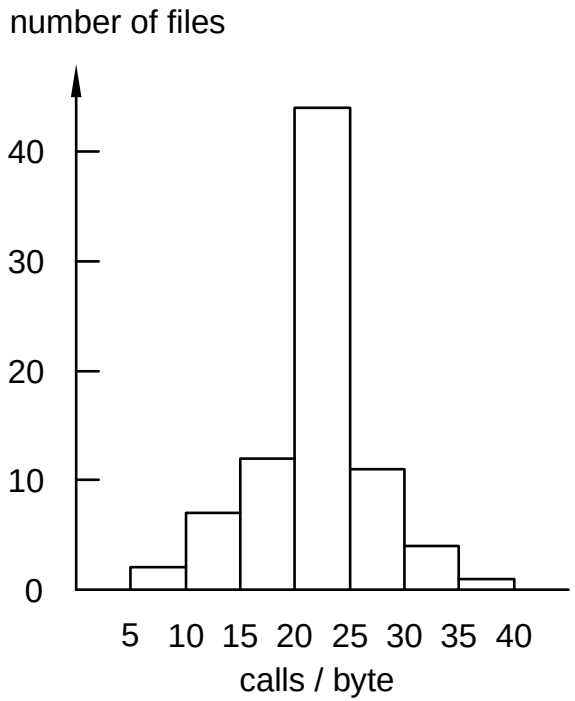
Measuring performance

Wrote PEG for Java 1.6 and generated parser for it.
Applied the parser to 80 Java files of sizes from 0.1 to 476 kBytes.
Counted calls to parsing procedures.



Measuring performance

Wrote PEG for Java 1.6 and generated parser for it.
Applied the parser to 80 Java files of sizes from 0.1 to 476 kBytes.
Counted calls to parsing procedures.



Time proportional to file size!

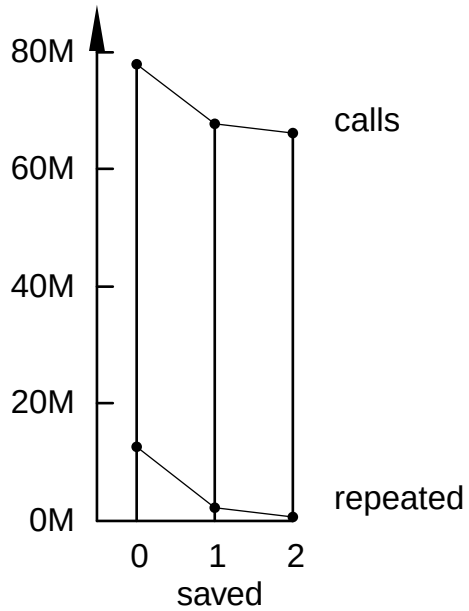
Measuring performance

Counted calls repeated due to backtracking.
Found them to be 16% of the total.

Measuring performance

Counted calls repeated due to backtracking.
Found them to be 16% of the total.

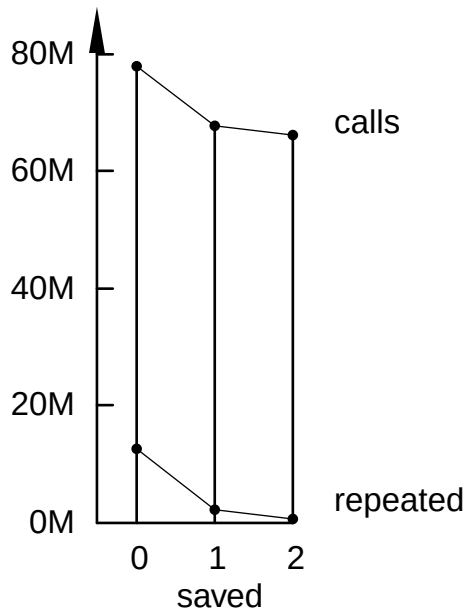
Tried minimal memoization.



Measuring performance

Counted calls repeated due to backtracking.
Found them to be 16% of the total.

Tried minimal memoization.



Not much use of saving more than 1 result.
Down with pack rats!

Mouse is not dead

Mouse is, by the way, alive and thrives in its hole on SourceForge.

You can feed it cheese ... well ... grammar & semantic procedures to create a compiler.

Welcome!

Is my PEG correct?

I wrote my PEG for Java as if it was EBNF.
It accepts 10 000 source files.

Is my PEG correct?

I wrote my PEG for Java as if it was EBNF.
It accepts 10 000 source files.

So what?
How do I know it is correct?

PEG is not EBNF

$\mathcal{L}^{EBNF}(a|aa) = \{a, aa\}$ $\mathcal{L}^{PEG}(a|aa) = \{a\}$

$\mathcal{L}^{EBNF}(aa|a)a = \{aaa, aa\}$ $\mathcal{L}^{PEG}(aa|a)a = \{aaa\}$

$\mathcal{L}^{EBNF}(aa|a)b = \{aab, ab\}$ $\mathcal{L}^{PEG}(aa|a)b = \{aab, ab\}$

$A = aAa|aa$ $\mathcal{L}^{EBNF}(A) =$ even number of a's $\mathcal{L}^{PEG}(A) = 2^n$ a's

Trying to define the language

$$\mathcal{S}(e_1)\mathcal{S}(e_3) \cup (\mathcal{S}(e_2)\mathcal{S}(e_3) - \mathcal{L}(e_1)\Sigma^* - \mathsf{Pref}(\mathcal{L}(e_1))) \subseteq \mathcal{S}((e_1/e_2)\,e_3)$$
$$\subseteq \mathcal{L}((e_1/e_2)\,e_3) \subseteq \mathcal{L}(e_1)\mathcal{L}(e_3) \cup (\mathcal{L}(e_2)\mathcal{L}(e_3) - \mathcal{S}(e_1)\Sigma^*)$$

Rio de Janeiro 2010

Rio de Janeiro 2010

Ph.D. thesis of Sérgio Medeiros at Pontifícia Universidade Católica.

Ph.D. thesis of Sérgio Medeiros at Pontifícia Universidade Católica.

Use relation $\overset{\text{PEG}}{\rightsquigarrow}$ to define the meaning of PEG:

- $[e]\ xy \overset{\text{PEG}}{\rightsquigarrow} y$ Expression e consumes prefix x of string xy.
- $[e]\ xy \overset{\text{PEG}}{\rightsquigarrow} \text{fail}$ Expression e fails on string xy.

Relation holds if it can be proved using given set of inference rules.

PEG semantics

$$\frac{[e(A)]\ xy \overset{\text{PEG}}{\rightsquigarrow} Y}{[A]\ xy \overset{\text{PEG}}{\rightsquigarrow} Y} \qquad \frac{}{[\varepsilon]\ x \overset{\text{PEG}}{\rightsquigarrow} x}$$

$$\frac{}{[a]\ ax \overset{\text{PEG}}{\rightsquigarrow} x} \qquad \frac{w \neq ax}{[a]\ w \overset{\text{PEG}}{\rightsquigarrow} \text{fail}}$$

$$\frac{[e]\ xy \overset{\text{PEG}}{\rightsquigarrow} y}{[!\,e]\ xy \overset{\text{PEG}}{\rightsquigarrow} \text{fail}} \qquad \frac{[e]\ x \overset{\text{PEG}}{\rightsquigarrow} \text{fail}}{[!\,e]\ x \overset{\text{PEG}}{\rightsquigarrow} x}$$

$$\frac{[e_1]\ xyz \overset{\text{PEG}}{\rightsquigarrow} yz \quad [e_2]\ yz \overset{\text{PEG}}{\rightsquigarrow} Z}{[e_1 e_2]\ xyz \overset{\text{PEG}}{\rightsquigarrow} Z} \qquad \frac{[e_1]\ x \overset{\text{PEG}}{\rightsquigarrow} \text{fail}}{[e_1 e_2]\ x \overset{\text{PEG}}{\rightsquigarrow} \text{fail}}$$

$$\frac{[e_1]\ xy \overset{\text{PEG}}{\rightsquigarrow} y}{[e_1 \mid e_2]\ xy \overset{\text{PEG}}{\rightsquigarrow} y} \qquad \frac{[e_1]\ xy \overset{\text{PEG}}{\rightsquigarrow} \text{fail} \quad [e_2]\ xy \overset{\text{PEG}}{\rightsquigarrow} Y}{[e_1 \mid e_2]\ xy \overset{\text{PEG}}{\rightsquigarrow} Y}$$

Example of proof

$X = aY$
$Y = b \mid X$

Proof of $[X]\ ab \overset{\text{PEG}}{\rightsquigarrow} \varepsilon$

$$\dfrac{\dfrac{}{[a]\ ab \overset{\text{PEG}}{\rightsquigarrow} b} \quad \dfrac{\dfrac{\dfrac{}{[b]\ b \overset{\text{PEG}}{\rightsquigarrow} \varepsilon}}{[b|X]\ b \overset{\text{PEG}}{\rightsquigarrow} \varepsilon}}{[Y]\ b \overset{\text{PEG}}{\rightsquigarrow} \varepsilon}}{\dfrac{[aY]\ ab \overset{\text{PEG}}{\rightsquigarrow} \varepsilon}{[X]\ ab \overset{\text{PEG}}{\rightsquigarrow} \varepsilon}}$$

Natural semantics

- Simpler than Ford’s
- Proofs by induction on height of derivation tree

EBNF semantics

The meaning of EBNF can be similarly defined by relation $\overset{\text{BNF}}{\rightsquigarrow}$ where $[e]\ xy \overset{\text{BNF}}{\rightsquigarrow} y$ means x conforms to syntax defined by e.

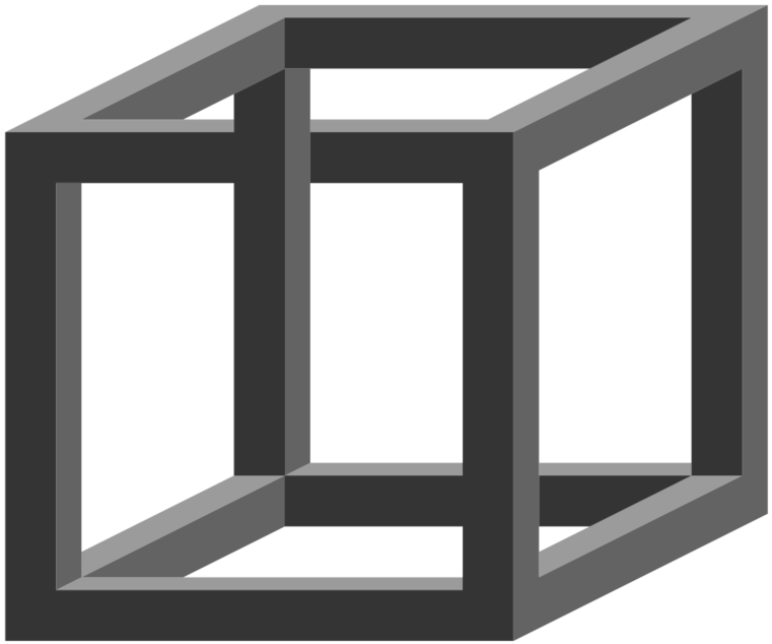
The inference rules are:

$$\frac{[e(A)]\ xy \overset{\text{BNF}}{\rightsquigarrow} Y}{[A]\ xy \overset{\text{BNF}}{\rightsquigarrow} Y} \qquad \frac{}{[\varepsilon]\ x \overset{\text{BNF}}{\rightsquigarrow} x} \qquad \frac{}{[a]\ ax \overset{\text{BNF}}{\rightsquigarrow} x}$$

$$\frac{}{[!\,e]\ x \overset{\text{BNF}}{\rightsquigarrow} x} \qquad \frac{[e_1]\ xyz \overset{\text{BNF}}{\rightsquigarrow} yz \quad [e_2]\ yz \overset{\text{BNF}}{\rightsquigarrow} z}{[e_1 e_2]\ xyz \overset{\text{BNF}}{\rightsquigarrow} z}$$

$$\frac{[e_1]\ xy \overset{\text{BNF}}{\rightsquigarrow} y}{[e_1 | e_2]\ xy \overset{\text{BNF}}{\rightsquigarrow} y} \qquad \frac{[e_2]\ xy \overset{\text{BNF}}{\rightsquigarrow} y}{[e_1 | e_2]\ xy \overset{\text{BNF}}{\rightsquigarrow} y}$$

Two interpretations



A grammar that can be viewed

- either as PEG, using $\overset{\text{PEG}}{\rightsquigarrow}$ and defining $\mathcal{L}^{PEG}(e)$

- or as EBNF, using $\overset{\text{BNF}}{\rightsquigarrow}$ and defining $\mathcal{L}^{EBNF}(e)$

Basic results

$\mathcal{L}^{PEG}(e) \subseteq \mathcal{L}^{EBNF}(e)$ for all e.

Basic results

$\mathcal{L}^{PEG}(e) \subseteq \mathcal{L}^{EBNF}(e)$ for all e.

if the grammar is not left-recursive
and for all $A = e_1|e_2$ holds

$$\boxed{\mathcal{L}^{EBNF}(e_1)\Sigma^* \cap \mathcal{L}^{EBNF}(e_2)\,\mathrm{Tail}(A) = \varnothing}$$

then $\mathcal{L}^{PEG}(e) = \mathcal{L}^{EBNF}(e)$ for all e.

Basic results

$\mathcal{L}^{PEG}(e) \subseteq \mathcal{L}^{EBNF}(e)$ for all e.

if the grammar is not left-recursive
and for all $A = e_1 | e_2$ holds

$$\boxed{\mathcal{L}^{EBNF}(e_1)\Sigma^* \cap \mathcal{L}^{EBNF}(e_2)\,\mathrm{Tail}(A) = \varnothing}$$

then $\mathcal{L}^{PEG}(e) = \mathcal{L}^{EBNF}(e)$ for all e.

Tail(A): set of strings that can follow $\mathcal{L}^{EBNF}(e_2)$ in corect input.
Can be formally defined using natural semantics.

Checking the condition

$A = e_1 | e_2$

$$\boxed{\mathcal{L}^{EBNF}(e_1)\Sigma^* \cap \mathcal{L}^{EBNF}(e_2)\,\mathrm{Tail}(A) = \varnothing}$$

Checking the condition

$A = e_1 | e_2$

$$\boxed{\mathcal{L}^{EBNF}(e_1)\Sigma^* \cap \mathcal{L}^{EBNF}(e_2)\,\mathrm{Tail}(A) = \varnothing}$$

Special case: LL(1) - can be checked by classical methods.

Checking the condition

$A = e_1 | e_2$

$$\boxed{\mathcal{L}^{EBNF}(e_1)\Sigma^* \cap \mathcal{L}^{EBNF}(e_2)\,\mathrm{Tail}(A) = \varnothing}$$

Special case: LL(1) - can be checked by classical methods.

No algorithm for general case.

Checking the condition

$A = e_1 | e_2$

$$\boxed{\mathcal{L}^{EBNF}(e_1)\Sigma^* \cap \mathcal{L}^{EBNF}(e_2)\,\mathrm{Tail}(A) = \varnothing}$$

Special case: LL(1) - can be checked by classical methods.

No algorithm for general case.

But the condition can often be checked by inspection.

Checking the condition

$A = e_1 | e_2$

$$\mathcal{L}^{EBNF}(e_1)\Sigma^* \cap \mathcal{L}^{EBNF}(e_2)\,\mathrm{Tail}(A) = \varnothing$$

Special case: LL(1) - can be checked by classical methods.

No algorithm for general case.

But the condition can often be checked by inspection.

Developed experimental tool, the *PEG Analyzer*, to assist it.
Presents for inspection the cases failing LL(1) test.

Analyzer example

```
Expr    = Term (AddOp Term)*
Term    = Factor (MultOp Factor)*
Factor  = Float | Integer | "(" Expr ")"
Integer = Digits
Float   = Digits? "." Digits
AddOp   = [+-]
MultOp  = [*/]
Digits  = Digit+
Digit   = [0-9]
```

Analyzer example

```
Expr    = Term (AddOp Term)*
Term    = Factor (MultOp Factor)*
Factor  = Float | Integer | "(" Expr ")"
Integer = Digits
Float   = Digits? "." Digits
AddOp   = [+-]
MultOp  = [*/]
Digits  = Digit+
Digit   = [0-9]
```

```
Factor:  Float <==> Integer Tail(Factor)
```

Inspection

Factor = Float | Integer | "(" Expr ")"

```
Float  <==>  Integer Tail(Factor)
[0-9]  <==>  [0-9]


Float
Digits "." Digits

==============================

Integer Tail(Factor)
Integer (AddOp Term | MultOp Factor | ")") ...
Digits ([+-] Term | [*/] Factor | ")") ...
```

A modification

Suppose I want to write 2(3+5) instead of 2*(3+5)

```
Expr    = Term (AddOp Term)*
Term    = Factor (MultOp Factor)*
Factor  = Float | Integer | "(" Expr ")"
Integer = Digits
Float   = Digits? "." Digits
AddOp   = [+-]
MultOp  = [*]? [/]
Digits  = Digit+
Digit   = [0-9]
```

A modification

Suppose I want to write 2(3+5) instead of 2*(3+5)

```
Expr    = Term (AddOp Term)*
Term    = Factor (MultOp Factor)*
Factor  = Float | Integer | "(" Expr ")"
Integer = Digits
Float   = Digits? "." Digits
AddOp   = [+-]
MultOp  = [*]? [/]
Digits  = Digit+
Digit   = [0-9]
```

```
Factor:  Float  <==>  Integer Tail(Factor)
Digits:  Digit  <==>  Tail(Digits)
```

Inspection

Factor = Float | Integer | "(" Expr ")"

```
Float  <==>  Integer Tail(Factor)
[0-9]  <==>  [0-9]


Float
Digits? "." Digits
Digits "." Digits

==============================

Integer Tail(Factor)
Integer (AddOp Term | MultOp Factor | ")") ...
Digits ([+-] Term | [*] Factor | Factor | [/] Factor | ")") ...
Digits ([+-] Term | [*] Factor | "." Digits | [/] Factor | ")") ...
```

Inspection

Factor = Float | Integer | "(" Expr ")"

```
Float  <==>  Integer Tail(Factor)
[0-9]  <==>  [0-9]


Float
Digits? "." Digits
Digits "." Digits

==============================

Integer Tail(Factor)
Integer (AddOp Term | MultOp Factor | ")") ...
Digits ([+-] Term | [*] Factor | Factor | [/] Factor | ")") ...
Digits ([+-] Term | [*] Factor | "." Digits | [/] Factor | ")") ...
```

It means 3.5 is not recognized as 3*.5

Inspection

Factor = Float | Integer | "(" Expr ")"

```
Float  <==>  Integer Tail(Factor)
[0-9]  <==>  [0-9]


Float
Digits? "." Digits
Digits "." Digits

==============================

Integer Tail(Factor)
Integer (AddOp Term | MultOp Factor | ")") ...
Digits ([+-] Term | [*] Factor | Factor | [/] Factor | ")") ...
Digits ([+-] Term | [*] Factor | "." Digits | [/] Factor | ")") ...
```

It means 3.5 is not recognized as 3*.5

The other indication means 35 is not recognized as 3*5

What about predicates?

If the grammar uses predicates, it is no longer EBNF.

What about predicates?

If the grammar uses predicates, it is no longer EBNF.

No $\mathcal{L}^{EBNF}(e)$.

Can not talk about equivalence.

What about predicates?

If the grammar uses predicates, it is no longer EBNF.

No $\mathcal{L}^{EBNF}(e)$.

Can not talk about equivalence.

Can only make a loose argument that limited backtracking does not affect the result if for all $A = e_1|e_2$ holds

$$\boxed{\mathcal{L}^{PEG}(e_1)\Sigma^* \cap \mathcal{L}^{PEG}(e_2)\,\mathrm{Tail}(A) = \varnothing}$$

A sort of solution

Define $\mathcal{L}^{EBNF}(\&e) = \mathcal{L}^{EBNF}(!e) = \{\varepsilon\}$.

A sort of solution

Define $\mathcal{L}^{EBNF}(\&e) = \mathcal{L}^{EBNF}(!e) = \{\varepsilon\}$.

This preserves $\mathcal{L}^{PEG}(e) \subseteq \mathcal{L}^{EBNF}(e)$.

And LL(1) makes sense.

A sort of solution

Define $\mathcal{L}^{EBNF}(\&e) = \mathcal{L}^{EBNF}(!e) = \{\varepsilon\}$.

This preserves $\mathcal{L}^{PEG}(e) \subseteq \mathcal{L}^{EBNF}(e)$.

And LL(1) makes sense.

And LL(1) implies $\mathcal{L}^{EBNF}(e_1)\Sigma^* \cap \mathcal{L}^{EBNF}(e_2)\,\mathrm{Tail}(A) = \varnothing$

and this implies $\mathcal{L}^{PEG}(e_1)\Sigma^* \cap \mathcal{L}^{PEG}(e_2)\,\mathrm{Tail}(A) = \varnothing$

and I can use LL(1) to select cases for inspection.

Analyzer example

```
Statement  = Keyword This | Identifier That
Keyword    = "print" !Letter
Identifier = !Keyword Letter+
Letter = [a-z]
This = ...
That = ...
```

Analyzer example

```
Statement  = Keyword This | Identifier That
Keyword    = "print" !Letter
Identifier = !Keyword Letter+
Letter = [a-z]
This = ...
That = ...
```

```
Statement:  Keyword This <==> Identifier That Tail(Statement)
```

Inspection

Statement = Keyword This | Identifier That

```
 Keyword This <==> Identifier That Tail(Statement)
  "print"  <==>  [a-z]


Keyword This
"print" !Letter This

==============================

Identifier That Tail(Statement)
Identifier That ...
!Keyword Letter+ That ...
```

Research problem

Replace predicates by something equally useful
but compatible with EBNF,
to enable formal proof of $\mathcal{L}^{PEG}(e) = \mathcal{L}^{EBNF}(e)$.

Research problem

Replace predicates by something equally useful
but compatible with EBNF,
to enable formal proof of $\mathcal{L}^{PEG}(e) = \mathcal{L}^{EBNF}(e)$.

EBNF defines language by construction.
PEG predicates belong to recognition.

Thanks

THANK YOU FOR YOUR PATIENCE!