Dependently Typed Grammars

MPC 2010

Kasper Brink, Stefan Holdermans, Andres Löh

June 22, 2010

Parser Combinators

Expression Grammar

$$
\begin{aligned}
E &\rightarrow E\ B\ N \mid N \\
B &\rightarrow + \mid - \\
N &\rightarrow 0 \mid 1
\end{aligned}
$$

```
pExpr, pNum :: Parser Int
pBin        :: Parser (Int → Int → Int)

pExpr  =    (λ e b n → b e n) <$> pExpr <*> pBin <*> pNum
       <|>  pNum

pBin   =    (+) <$ pSymbol '+'
       <|>  (−) <$ pSymbol '-'

pNum   =    0 <$ pSymbol '0'
       <|>  1 <$ pSymbol '1'
```

Parser Combinators

Expression Grammar

$$
\begin{aligned}
E &\rightarrow E\; B\; N \;\mid\; N \\
B &\rightarrow + \;\mid\; - \\
N &\rightarrow 0 \;\mid\; 1
\end{aligned}
$$

```
pExpr, pNum :: Parser Int
pBin        :: Parser (Int → Int → Int)

pExpr  =    (λ e b n → b e n) <$> pExpr <*> pBin <*> pNum
       <|>  pNum

pBin   =    (+) <$ pSymbol '+'
       <|>  (−) <$ pSymbol '-'

pNum   =    0 <$ pSymbol '0'
       <|>  1 <$ pSymbol '1'
```

Left Recursion ⟶ Non-termination!

Representing grammars instead of parsers

- Represent a grammar as a *data value*
- Analyze and transform
- Generate a parser

Representing grammars instead of parsers

- Represent a grammar as a *data value*
- Analyze and transform
- Generate a parser

This talk

- Representation in Agda
- Transform grammar to remove left recursion

Outline

- Grammar Representation
- Left-Corner Transform
- (Part of) Correctness Proof
- Conclusion

Grammar Representation

Symbols

```
Terminal : Set
Terminal = Char

data Nonterminal : Set where
  E : Nonterminal
  B : Nonterminal
  N : Nonterminal

data Symbol : Set where
  st : Terminal → Symbol
  sn : Nonterminal → Symbol
```

Semantic Types

- Parsers: every parser has a result type
- Grammars: every nonterminal has a semantic type

$$
\begin{aligned}
[\![\_]\!] &: \text{Nonterminal} \rightarrow \text{Set} \\
[\![\ \mathsf{E}\ ]\!] &= \mathbb{N} \\
[\![\ \mathsf{B}\ ]\!] &= \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\
[\![\ \mathsf{N}\ ]\!] &= \mathbb{N}
\end{aligned}
$$

Semantic Functions

- Type of semantic functions determined by $[\![\_]\!]$

$E \rightarrow E\ B\ N$	λ e b n $\rightarrow$ b e n	: $[\![\text{E}]\!] \rightarrow [\![\text{B}]\!] \rightarrow [\![\text{N}]\!] \rightarrow [\![\text{E}]\!]$
$E \rightarrow N$	id	: $[\![\text{N}]\!] \rightarrow [\![\text{E}]\!]$
$N \rightarrow 1$	1	: $[\![\text{N}]\!]$

Semantic Functions

- Type of semantic functions determined by $[\![\_]\!]$

$E \rightarrow E\ B\ N$	λ e b n $\rightarrow$ b e n	: $[\![$ E $]\!] \rightarrow [\![$ B $]\!] \rightarrow [\![$ N $]\!] \rightarrow [\![$ E $]\!]$
$E \rightarrow N$	id	: $[\![$ N $]\!] \rightarrow [\![$ E $]\!]$
$N \rightarrow 1$	1	: $[\![$ N $]\!]$

- Compute type of semantic function: $[\![\_\|\_]\!]$
- Production $A \rightarrow \beta$ has semantic function of type $[\![\beta \| A]\!]$

$[\![\_\|\_]\!]$: Symbols $\rightarrow$ Nonterminal $\rightarrow$ Set
$[\![$ [ ] $\|$ A $]\!]$ = $[\![$ A $]\!]$
$[\![$ st _ :: β $\|$ A $]\!]$ = $[\![$ β $\|$ A $]\!]$
$[\![$ sn B :: β $\|$ A $]\!]$ = $[\![$ B $]\!] \rightarrow [\![$ β $\|$ A $]\!]$

Productions

$$\textbf{data}\ \text{Production} : \text{Set}\ \textbf{where}$$
$$\quad \text{prod} : (\text{A} : \text{Nonterminal}) \rightarrow (\beta : \text{Symbols}) \rightarrow [\![\ \beta \parallel \text{A}\ ]\!] \rightarrow \text{Production}$$

Example:

$$\begin{aligned}
p_1 &= \text{prod E (sn E :: sn B :: sn N :: [ ])}\ (\lambda\ \text{e b n} \rightarrow \text{b e n}) \\
p_2 &= \text{prod E (sn N :: [ ])}\ \text{id} \\
p_3 &= \text{prod N (st '1' :: [ ])}\ 1
\end{aligned}$$

Of course it is desirable to devise a more convenient input syntax for grammars.

Generating a Parser

```
generateParser : Productions → (S : Nonterminal) → Parser ⟦ S ⟧
generateParser prods = gen where
  mutual
    gen : (A : Nonterminal) → Parser ⟦ A ⟧
    gen A = (foldr _<|>_ pFail ∘ map genAlt ∘ filterLHS A) prods

    genAlt : ∀ {A} → ProductionLHS A → Parser ⟦ A ⟧
    genAlt (prodlhs (prod A β sem)) = buildParser β (pSucceed sem)

    buildParser : ∀ {A} β → Parser ⟦ β ∥ A ⟧ → Parser ⟦ A ⟧
    buildParser []          p = p
    buildParser (st b :: β) p = buildParser β (p <*  pTerminal b)
    buildParser (sn B :: β) p = buildParser β (p <*> gen B)
```

Generating a Parser

```
generateParser : Productions → (S : Nonterminal) → Parser ⟦ S ⟧
generateParser prods = gen where
  mutual
    gen : (A : Nonterminal) → Parser ⟦ A ⟧
    gen A = (foldr _<|>_ pFail ∘ map genAlt ∘ filterLHS A) prods

    genAlt : ∀ {A} → ProductionLHS A → Parser ⟦ A ⟧
    genAlt (prodlhs (prod A β sem)) = buildParser β (pSucceed sem)

    buildParser : ∀ {A} β → Parser ⟦ β ∥ A ⟧ → Parser ⟦ A ⟧
    buildParser []          p = p
    buildParser (st b :: β) p = buildParser β (p <*  pTerminal b)
    buildParser (sn B :: β) p = buildParser β (p <*> gen B)
```

Generating a Parser

```
generateParser : Productions → (S : Nonterminal) → Parser ⟦ S ⟧
generateParser prods = gen where
  mutual
    gen : (A : Nonterminal) → Parser ⟦ A ⟧
    gen A = (foldr _<|>_ pFail ∘ map genAlt ∘ filterLHS A) prods

    genAlt : ∀ {A} → ProductionLHS A → Parser ⟦ A ⟧
    genAlt (prodlhs (prod A β sem)) = buildParser β (pSucceed sem)

    buildParser : ∀ {A} β → Parser ⟦ β ∥ A ⟧ → Parser ⟦ A ⟧
    buildParser []           p = p
    buildParser (st b  :: β) p = buildParser β (p <*  pTerminal b)
    buildParser (sn B :: β) p = buildParser β (p <*> gen B)
```

Left-Corner Transform

Left Corners

- Left corner: $A \stackrel{*}{\Rightarrow} X\beta$

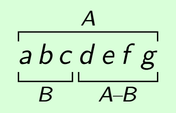
Left Corners

- Left corner: $A \stackrel{*}{\Rightarrow} {\color{red}X}\beta$

- Left-corner transform introduces new nonterminals "A–X"
- A–X represents the part of an A that follows an X.

- Example:

  $A \stackrel{*}{\Rightarrow} B\beta \stackrel{*}{\Rightarrow} a\,b\,c\,\beta \stackrel{*}{\Rightarrow} a\,b\,c\,d\,e\,f\,g$



Left-corner Transform

Transformation Rules (Johnson, 1998)

$$
\begin{array}{lll}
(1) & \forall A,\ b: & A \rightarrow b\ A\text{–}b \\
(2) & \forall C,\ A \rightarrow X\ \beta: & C\text{–}X \rightarrow \beta\ C\text{–}A \\
(3) & \forall A: & A\text{–}A \rightarrow \epsilon
\end{array}
$$

Example Transformation

Original:

$E \rightarrow E\ B\ N$

$E \rightarrow N$

$B \rightarrow +$

$B \rightarrow -$

$N \rightarrow 0$

$N \rightarrow 1$

Transformed:

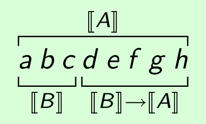
$E \rightarrow +\ E\text{–}+$	$B \rightarrow +\ B\text{–}+$	$N \rightarrow +\ N\text{–}+$
$E \rightarrow -\ E\text{–}-$	$B \rightarrow -\ B\text{–}-$	$N \rightarrow -\ N\text{–}-$
$E \rightarrow 0\ \ E\text{–}0$	$B \rightarrow 0\ \ B\text{–}0$	$N \rightarrow 0\ \ N\text{–}0$
$E \rightarrow 1\ \ E\text{–}1$	$B \rightarrow 1\ \ B\text{–}1$	$N \rightarrow 1\ \ N\text{–}1$
$E\text{–}E \rightarrow B\ N\ E\text{–}E$	$B\text{–}E \rightarrow B\ N\ B\text{–}E$	$N\text{–}E \rightarrow B\ N\ N\text{–}E$
$E\text{–}N \rightarrow E\text{–}E$	$B\text{–}N \rightarrow B\text{–}E$	$N\text{–}N \rightarrow N\text{–}E$
$E\text{–}+ \rightarrow E\text{–}B$	$B\text{–}+ \rightarrow B\text{–}B$	$N\text{–}+ \rightarrow N\text{–}B$
$E\text{–}- \rightarrow E\text{–}B$	$B\text{–}- \rightarrow B\text{–}B$	$N\text{–}- \rightarrow N\text{–}B$
$E\text{–}0 \rightarrow E\text{–}N$	$B\text{–}0 \rightarrow B\text{–}N$	$N\text{–}0 \rightarrow N\text{–}N$
$E\text{–}1 \rightarrow E\text{–}N$	$B\text{–}1 \rightarrow B\text{–}N$	$N\text{–}1 \rightarrow N\text{–}N$
$E\text{–}E \rightarrow \epsilon$	$B\text{–}B \rightarrow \epsilon$	$N\text{–}N \rightarrow \epsilon$

New nonterminals

(notation: Original "O. . .", Transformed "T. . .")

```
data TNonterminal : Set where
  n     : ONonterminal → TNonterminal
  n_─_  : ONonterminal → OSymbol → TNonterminal
```

```
T⟦_⟧ : TNonterminal → Set
T⟦ n A ⟧          = O⟦ A ⟧
T⟦ n A ─ st b ⟧   = O⟦ A ⟧
T⟦ n A ─ sn B ⟧   = O⟦ B ⟧ → O⟦ A ⟧
```



Transforming Grammars

Transformation Rules

$$
\begin{array}{lll}
(1) & \forall A,\ b: & A \rightarrow b\, A\text{–}b \\
(2) & \forall C,\ A \rightarrow X\,\beta: & C\text{–}X \rightarrow \beta\, C\text{–}A \\
(3) & \forall A: & A\text{–}A \rightarrow \epsilon
\end{array}
$$

```
lct : OProductions → TProductions
lct ps =
  concatMap (λ A → map (rule1 A) (terms ps)) (nonterms ps) ++
  concatMap (λ C → map (rule2 C) ps)         (nonterms ps) ++
  map rule3 (nonterms ps)
```

Transforming Productions

Rule (2): $A \rightarrow X\,\beta \quad \longrightarrow \quad C\text{–}X \rightarrow \beta\; C\text{–}A$

$$
\begin{array}{l}
\mathsf{rule2}\ :\ \mathsf{ONonterminal} \rightarrow \mathsf{OProduction} \rightarrow \mathsf{TProduction} \\
\mathsf{rule2}\ \mathsf{C}\ (\mathsf{O.prod}\ \mathsf{A}\ (\mathsf{X} :: \beta)\ \mathsf{sem})\ = \\
\qquad \mathsf{T.prod}\ (\mathsf{n}\ \mathsf{C} - \mathsf{X})\ (\mathsf{lift}\ \beta\ +\!\!+\ [\mathsf{T.sn}\ (\mathsf{n}\ \mathsf{C} - \mathsf{O.sn}\ \mathsf{A})]) \\
\qquad\qquad\qquad\qquad\qquad\ (\mathsf{semtrans}\ \mathsf{C}\ \mathsf{A}\ \mathsf{X}\ \beta\ \mathsf{sem})
\end{array}
$$

Transforming Semantics

Use semantic types as *specification* of semantic transformation

Semantic transformation

$$\text{production:} \quad A \rightarrow B\,\beta \quad \longrightarrow \quad C\text{–}B \rightarrow \beta\; C\text{–}A$$

$$\text{semantics:} \quad [\![B\,\beta \,\|\, A]\!] \quad \longrightarrow \quad [\![\beta\; C\text{–}A \,\|\, C\text{–}B]\!]$$

Transforming Semantics

Use semantic types as *specification* of semantic transformation

Semantic transformation

$$\text{production:}\quad A \rightarrow B\,\beta \quad\longrightarrow\quad C\text{–}B \rightarrow \beta\; C\text{–}A$$

$$\text{semantics:}\quad [\![B\,\beta \,\|\, A]\!] \quad\longrightarrow\quad [\![\beta\; C\text{–}A \,\|\, C\text{–}B]\!]$$

$$\begin{aligned}
\text{semtrans} : \;& \forall\; \mathsf{C}\;\mathsf{A}\;\mathsf{B}\;\beta \rightarrow \\
& \mathsf{O}[\![\; \mathsf{O.sn}\;\mathsf{B} :: \beta \;\|\; \mathsf{A} \;]\!] \rightarrow \\
& \mathsf{T}[\![\; \mathsf{lift}\;\beta \;+\!\!+\; [\mathsf{T.sn}\;(\mathsf{n}\;\mathsf{C} - \mathsf{O.sn}\;\mathsf{A})] \;\|\; \mathsf{n}\;\mathsf{C} - \mathsf{O.sn}\;\mathsf{B} \;]\!]
\end{aligned}$$

Transforming Semantics

Use semantic types as *specification* of semantic transformation

Semantic transformation

$$\text{production:} \quad A \to B\,\beta \quad \longrightarrow \quad C\text{–}B \to \beta\; C\text{–}A$$

$$\text{semantics:} \quad [\![B\,\beta \,\|\, A]\!] \quad \longrightarrow \quad [\![\beta\; C\text{–}A \,\|\, C\text{–}B]\!]$$

$$\begin{aligned}
\text{semtrans} \;:\; & \forall\; \text{C A B}\; \beta \to \\
& \text{O}[\![\; \text{O.sn B} :: \beta \;\|\; \text{A} \;]\!] \to \\
& \text{T}[\![\; \text{lift}\; \beta \;+\!\!+\; [\text{T.sn (n C} - \text{O.sn A)}] \;\|\; \text{n C} - \text{O.sn B} \;]\!]
\end{aligned}$$

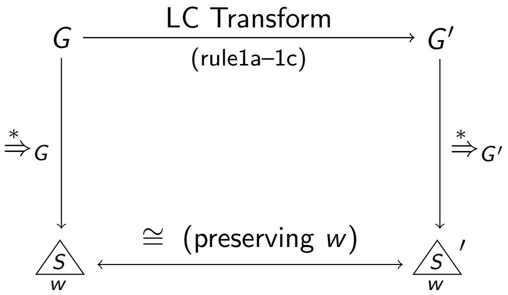
$$\begin{aligned}
\text{semtrans C A B}\; \beta \;=\; \text{O.foldSymbols}\; & (\lambda\; _\; \text{f} \to \text{f}) \\
& (\lambda\; _\; \text{f} \to \lambda\; \text{g} \to \text{f} \circ \text{flip g}) \\
& (\lambda\; \text{f}\;\; \text{g} \to \text{g} \circ \text{f}) \\
& \beta
\end{aligned}$$

Correctness

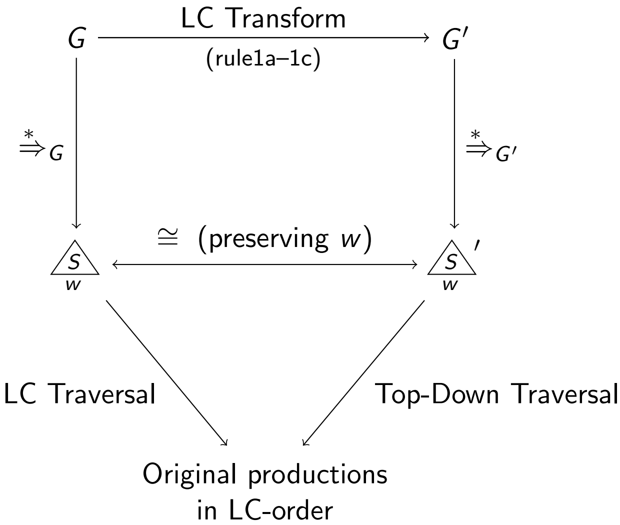
Correctness Criteria

- Correctness of the left-corner transform:
  - Transformed grammar recognizes the same language
  - No addition or removal of ambiguity
    (number of parse trees for each sentence is preserved)
  - Left recursion is removed
- What we proved (weaker):
  - Transformed grammar recognizes *at least* the original language:
    $\mathcal{L}(G) \subseteq \mathcal{L}(G')$

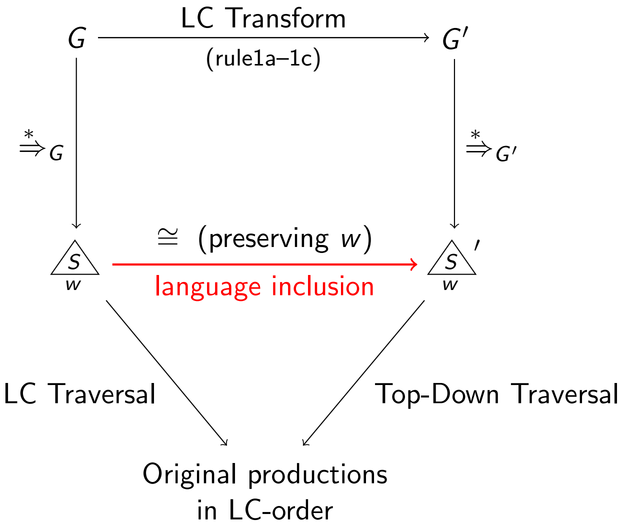
Concepts Involved in Proof



Concepts Involved in Proof



Concepts Involved in Proof

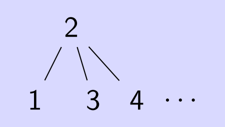


Parse Tree Traversals

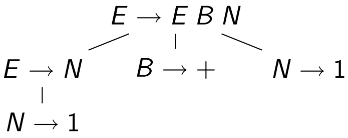
Top-down traversal: parent recognized *before* children

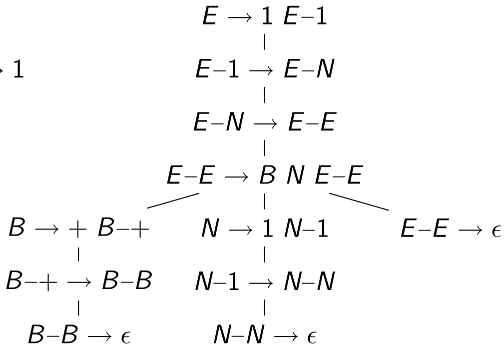
Bottom-up traversal: parent recognized *after* children

Left-corner traversal: parent recognized *after* left corner, and *before* other children



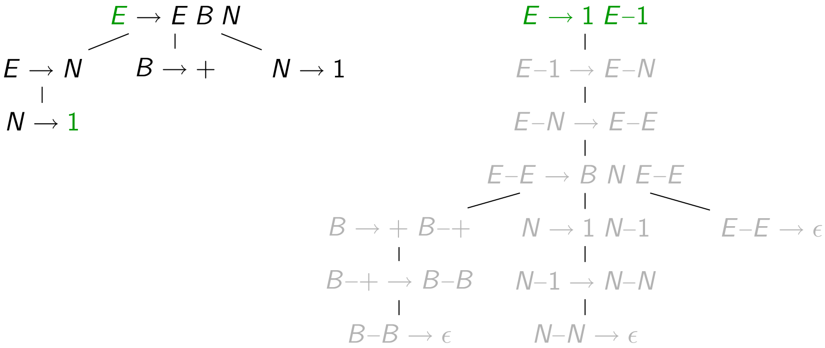
Left-Corner Traversal





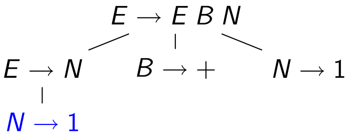
$1 + 1$

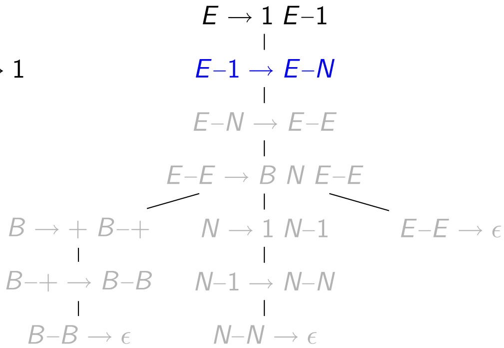
Left-Corner Traversal



$1 + 1$

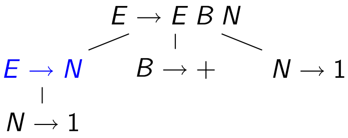
Left-Corner Traversal

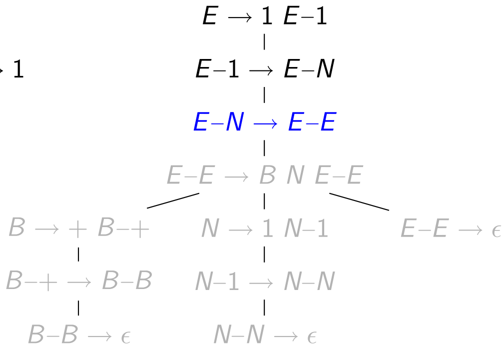




1 + 1

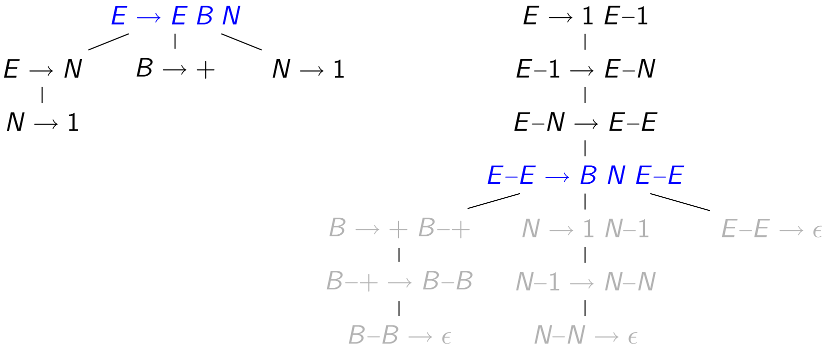
Left-Corner Traversal





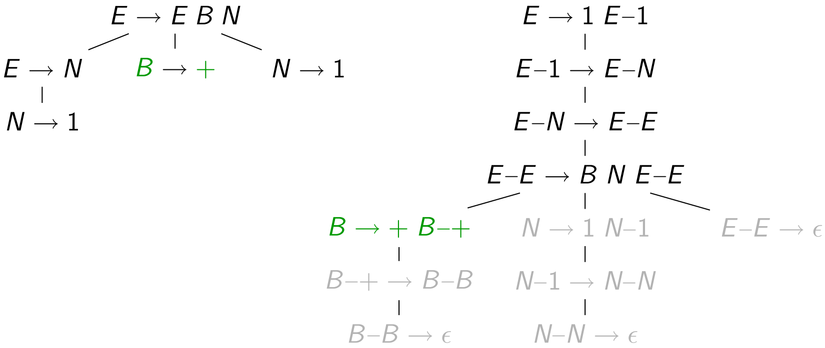
1 + 1

Left-Corner Traversal



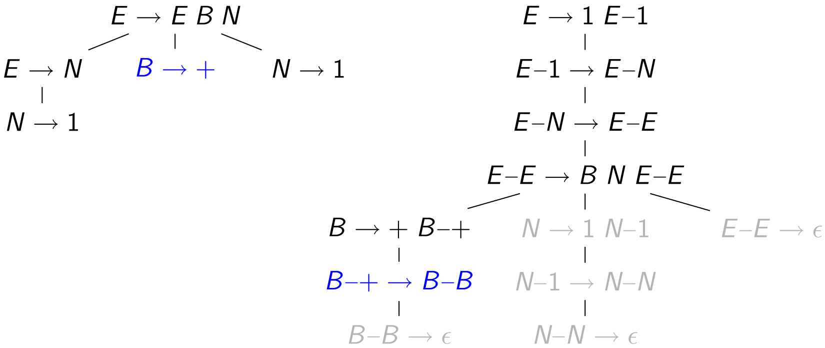
1 + 1

Left-Corner Traversal



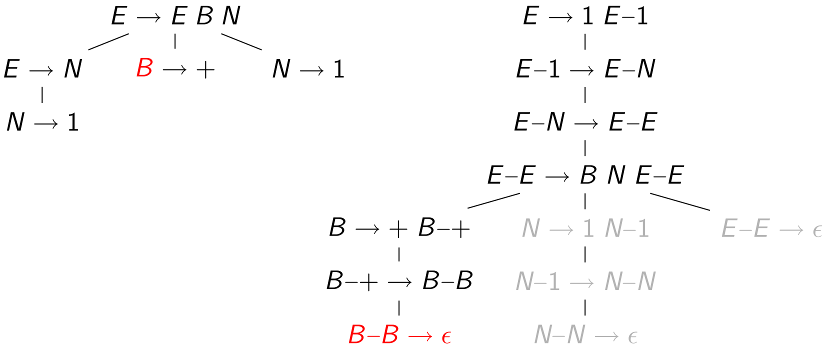
1 + 1

Left-Corner Traversal



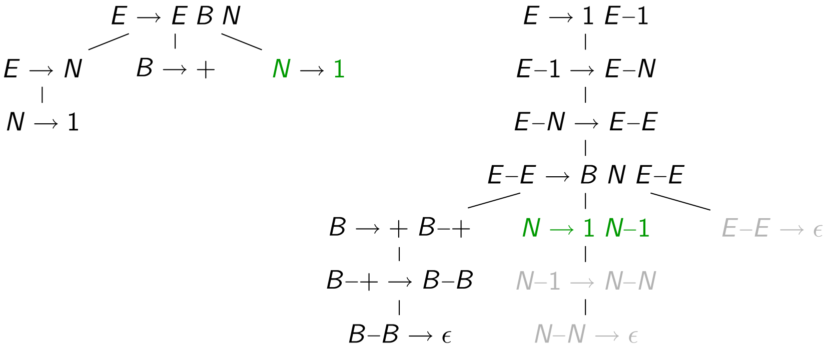
$1 + 1$

Left-Corner Traversal



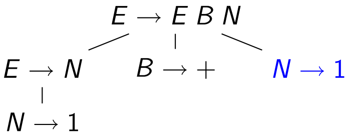
$1 + 1$

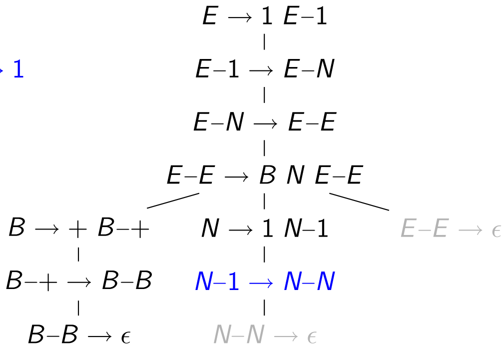
Left-Corner Traversal



$1 + 1$

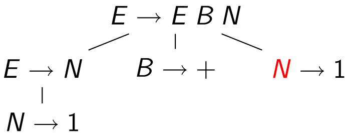
Left-Corner Traversal

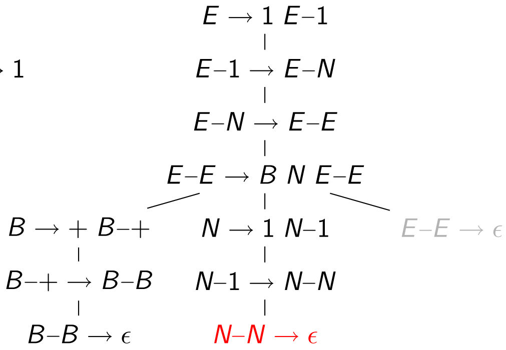




$1 + 1$

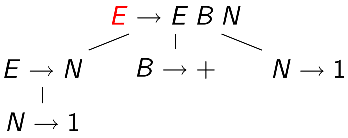
Left-Corner Traversal

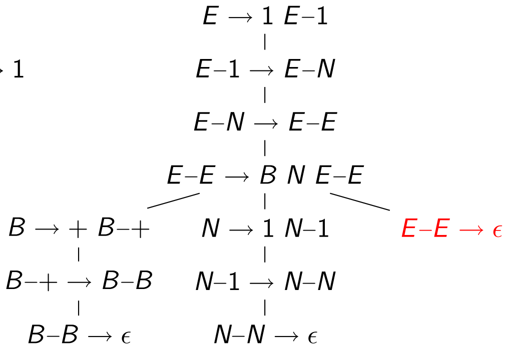




$1 + 1$

Left-Corner Traversal





$1 + 1$

Correctness Proof

Function $\underset{w}{\overset{S}{\triangle}} \rightarrow \underset{w}{\overset{S}{\triangle}}'$ *is* a proof that $\mathcal{L}(G) \subseteq \mathcal{L}(G')$

Proof Outline

- traverse $\underset{w}{\overset{S}{\triangle}}$ in LC-order
- transform productions
- add productions to $\underset{w}{\overset{S}{\triangle}}'$ in top-down order
- show that sentence w is preserved

Conclusion

Summary

Contributions

- Library for representing grammars and semantic functions, and generating parsers
- Implementation of the Left-Corner Transform
- Proof of a correctness property of our LCT implementation: $\mathcal{L}(G) \subseteq \mathcal{L}(G')$

Conclusions

- Dependent types are a natural fit for representing grammars.
- Proofs are possible, but a lot of work.
- This is just a start ...

Future Work

- Other grammar transformations
  (left factoring, ...)
- Grammar combinators
- Proof of non-left-recursion
  (total parser combinators, Danielsson and Norell)