

# Parsing Expression Grammars: A Recognition-Based Syntactic Foundation

Bryan Ford

Massachusetts Institute of Technology

January 14, 2004

# Designing a Language Syntax

# Designing a Language Syntax

## *Textbook Method*

1. Formalize syntax via context-free grammar
2. Write a YACC parser specification
3. Hack on grammar until “near-*LALR(1)*”
4. Use generated parser

# Designing a Language Syntax

## *Textbook Method*

1. Formalize syntax via context-free grammar
2. Write a YACC parser specification
3. Hack on grammar until “near-*LALR(1)*”
4. Use generated parser

## *Pragmatic Method*

1. Specify syntax informally
2. Write a recursive descent parser

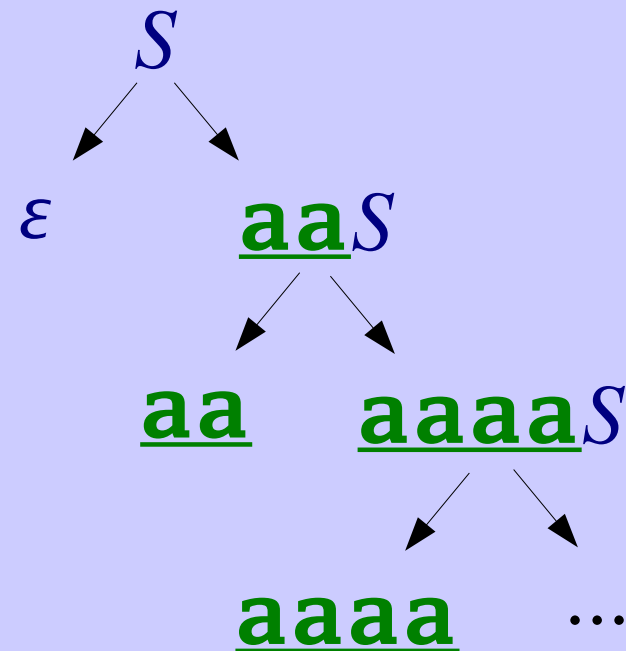
# What exactly does a CFG describe?

*Short answer:*

a rule system to **generate** language strings

*Example CFG:*

$$\begin{array}{l} S \rightarrow \underline{aa}S \\ S \rightarrow \varepsilon \end{array}$$



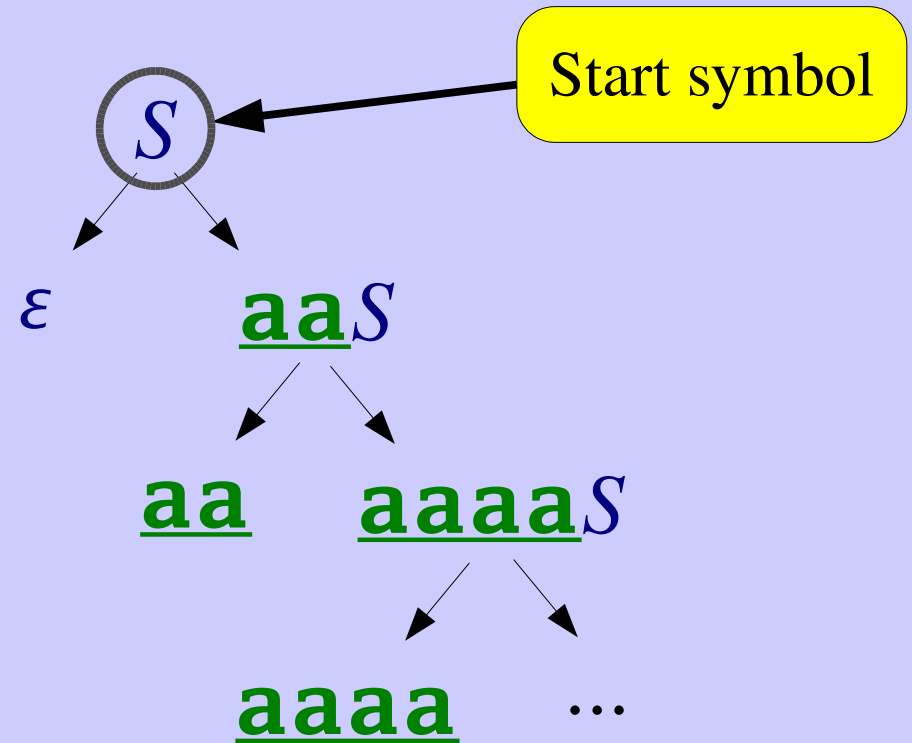
# What exactly does a CFG describe?

*Short answer:*

a rule system to **generate** language strings

*Example CFG:*

$$\begin{aligned} S &\rightarrow \underline{aa}S \\ S &\rightarrow \varepsilon \end{aligned}$$



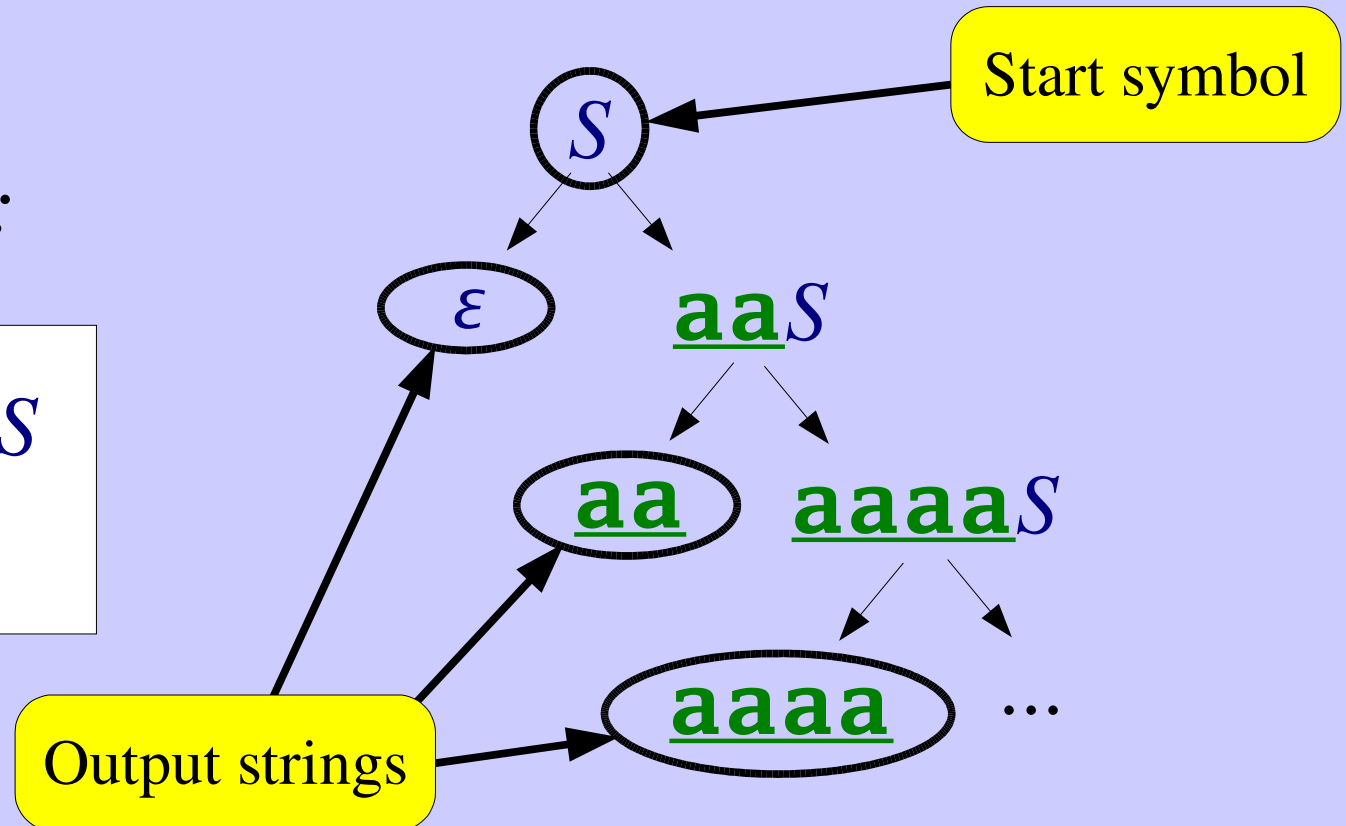
# What exactly does a CFG describe?

*Short answer:*

a rule system to **generate** language strings

*Example CFG:*

$S \rightarrow \underline{aa}S$   
 $S \rightarrow \varepsilon$



# What exactly do we *want* to describe?

*Proposed answer:*

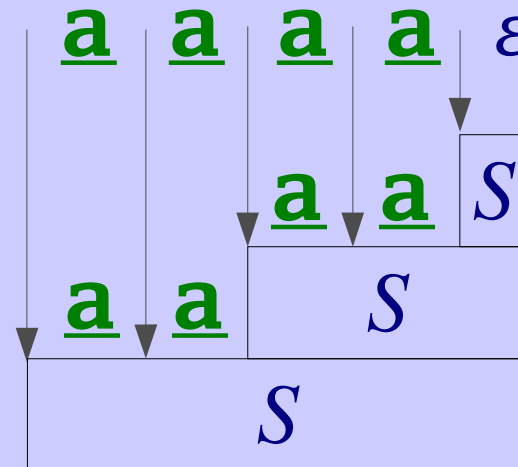
a rule system to **recognize** language strings

*Parsing Expression Grammar (PEG)*

models **recursive descent parsing practice**

*Example PEG:*

$$S \leftarrow \underline{aa}S / \varepsilon$$





# What exactly do we *want* to describe?

*Proposed answer:*

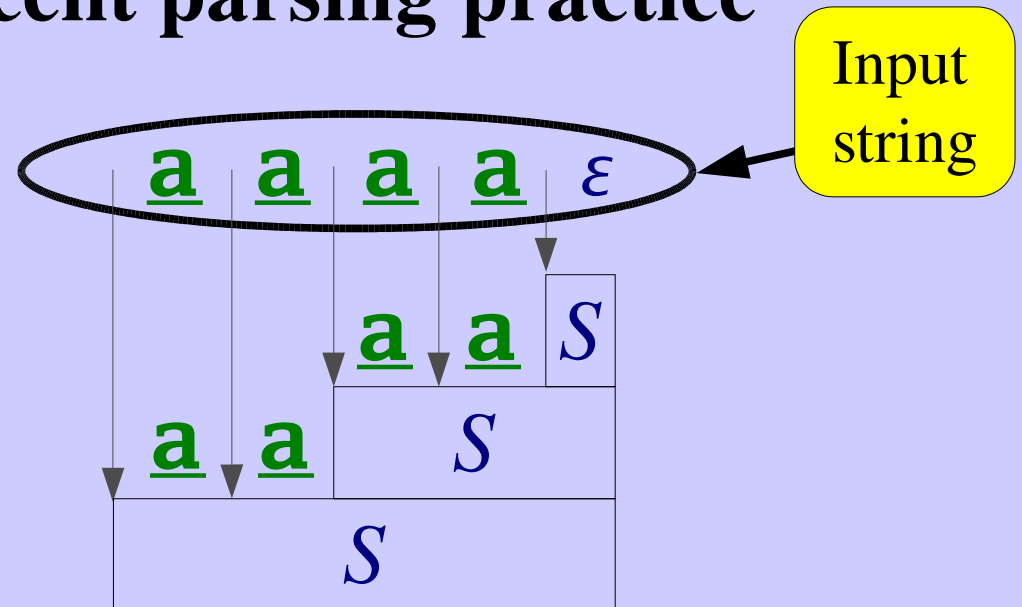
a rule system to **recognize** language strings

*Parsing Expression Grammar (PEG)*

models **recursive descent parsing practice**

*Example PEG:*

$$S \leftarrow \underline{aa}S / \varepsilon$$



# What exactly do we *want* to describe?

*Proposed answer:*

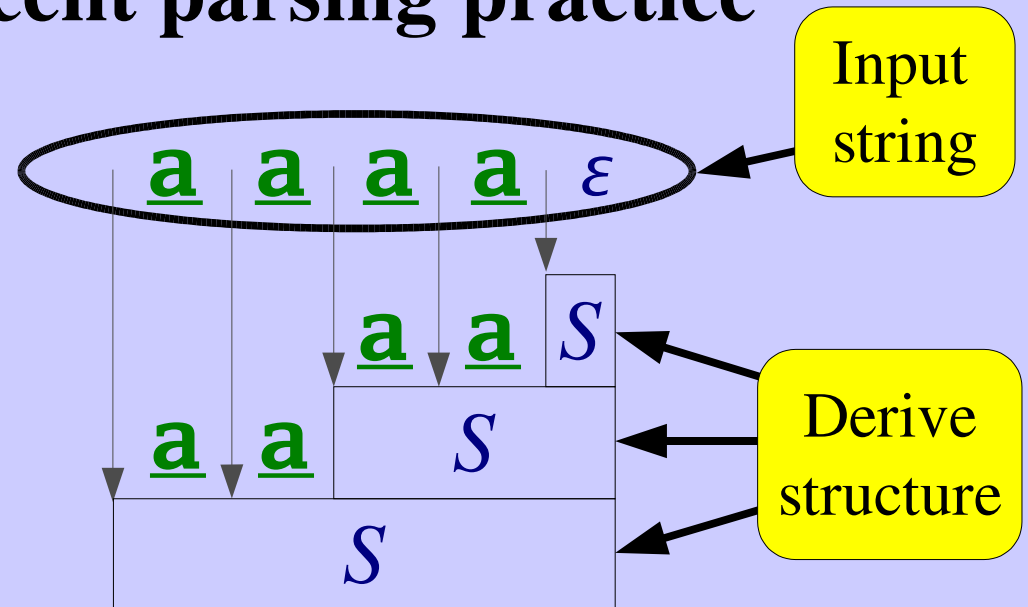
a rule system to **recognize** language strings

*Parsing Expression Grammar (PEG)*

models **recursive descent parsing practice**

*Example PEG:*

$$S \leftarrow \underline{aa}S / \varepsilon$$



# Take-Home Points

Key benefits of PEGs:

- **Simplicity, formalism, analyzability** of CFGs
- **Closer match** to syntax practices
  - More expressive than deterministic CFGs (*LL/LR*)
  - More of the “right kind” of expressiveness:  
*prioritized choice, greedy rules, syntactic predicates*
  - Unlimited lookahead, backtracking
- **Linear-time parsing** for *any* PEG

# *What kind of* recursive descent parsing?

Key assumptions:

- Parsing functions are **stateless**:  
*depend only on input string*
- Parsing functions **make decisions locally**:  
*return at most one result (success/failure)*

# Parsing Expression Grammars

Consists of:  $(\Sigma, N, R, e_S)$

- $\Sigma$ : finite set of *terminals* (character set)
- $N$ : finite set of *nonterminals*
- $R$ : finite set of rules of the form “ $A \leftarrow e$ ”,  
where  $A \in N$ ,  $e$  is a *parsing expression*.
- $e_S$ : a parsing expression called the *start expression*.

# Parsing Expressions

$\varepsilon$	the empty string
$\underline{\mathbf{a}}$	terminal ( $\mathbf{a} \in \Sigma$ )
$A$	nonterminal ( $A \in N$ )
$e_1 e_2$	a sequence of parsing expressions
$e_1 / e_2$	<i>prioritized choice</i> between alternatives
$e^?, e^*, e^+$	optional, zero-or-more, one-or-more
$\&e, !e$	syntactic predicates

# How PEGs Express Languages

Given input string  $s$ , a parsing expression either:

- **Matches** and consumes a prefix  $s'$  of  $s$ .
- **Fails** on  $s$ .

*Example:*

$S \leftarrow \underline{\text{bad}}$

$S$  matches “**badder**”

$S$  matches “**baddest**”

$S$  fails on “**abad**”

$S$  fails on “**babe**”

# Prioritized Choice with Backtracking

$S \leftarrow A / B$  means:

“To parse an  $S$ , *first* try to parse an  $A$ .

If  $A$  fails, *then* backtrack and try to parse a  $B$ .”

*Example:*

$S \leftarrow$  if C then S else S /  
if C then S

$S$  matches “if C then S foo”

$S$  matches “if C then S<sub>1</sub> else S<sub>2</sub>”

$S$  fails on “if C else S”



# Prioritized Choice with Backtracking

$S \leftarrow A / B$  means:

“To parse an  $S$ , *first* try to parse an  $A$ .

If  $A$  fails, *then* backtrack and try to parse a  $B$ .”

*Example* from the C++ standard:

“An *expression-statement* ... can be indistinguishable from a *declaration* ... In those cases the *statement* is a *declaration*.”

$statement \leftarrow declaration /$   
 $expression-statement$

# Greedy Option and Repetition

$A \leftarrow e?$  equivalent to  $\mathbf{A} \leftarrow e / \varepsilon$

$A \leftarrow e^*$  equivalent to  $\mathbf{A} \leftarrow e \mathbf{A} / \varepsilon$

$A \leftarrow e^+$  equivalent to  $\mathbf{A} \leftarrow e e^*$

*Example:*

$I \leftarrow L^+$

$L \leftarrow \underline{\mathbf{a}} / \underline{\mathbf{b}} / \underline{\mathbf{c}} / \dots$

$I$  matches “foobar”

$I$  matches “foo(bar)”

$I$  fails on “123”

# Syntactic Predicates

**And-predicate:**  $\&e$  succeeds whenever  $e$  does,  
*but consumes no input* [Parr '94, '95]

**Not-predicate:**  $!e$  succeeds whenever  $e$  fails

*Example:*

$A \leftarrow \underline{\text{foo}} \&(\underline{\text{bar}})$

$B \leftarrow \underline{\text{foo}} !(\underline{\text{bar}})$

$A$  matches “foobar”

$A$  fails on “foobie”

$B$  matches “foobie”

$B$  fails on “foobar”

# Syntactic Predicates

**And-predicate:**  $\&e$  succeeds whenever  $e$  does,  
*but consumes no input* [Parr '94, '95]

**Not-predicate:**  $!e$  succeeds whenever  $e$  fails

*Example:*

$$C \leftarrow B I^* E$$
$$I \leftarrow !E (C / T)$$
$$B \leftarrow \underline{(}^*$$
$$E \leftarrow \underline{*)}$$
$$T \leftarrow [any\ terminal]$$

$C$  matches “ $(^*ab^*)cd$ ”

$C$  matches “ $(^*a(^*b^*)c^*)$ ”

$C$  fails on “ $(^*a(^*b^*))$ ”

# Syntactic Predicates

**And-predicate:**  $\&e$  succeeds whenever  $e$  does,  
*but consumes no input* [Parr '94, '95]

**Not-predicate:**  $!e$  succeeds whenever  $e$  fails

*Example:*

$C \leftarrow \textcircled{B} I^* E$   
 $I \leftarrow !E (C / T)$   
 $B \leftarrow \underline{(}^*$   
 $E \leftarrow \underline{*)}$   
 $T \leftarrow [any\ terminal]$

Begin marker

$C$  matches “ $(^*ab^*)cd$ ”

$C$  matches “ $(^*a(^*b^*)c^*)$ ”

$C$  fails on “ $(^*a(^*b^*))$ ”

# Syntactic Predicates

**And-predicate:**  $\&e$  succeeds whenever  $e$  does,  
*but consumes no input* [Parr '94, '95]

**Not-predicate:**  $!e$  succeeds whenever  $e$  fails

*Example:*

Internal elements

$C \leftarrow B(I^*)E$

$I \leftarrow !E(C / T)$

$B \leftarrow \underline{C^*}$

$E \leftarrow \underline{^*}$

$T \leftarrow [any\ terminal]$

$C$  matches “ $(^*ab^*)cd$ ”

$C$  matches “ $(^*a(^*b^*)c^*)$ ”

$C$  fails on “ $(^*a(^*b^*))$ ”

# Syntactic Predicates

**And-predicate:**  $\&e$  succeeds whenever  $e$  does, but consumes no input [Parr '94, '95]

**Not-predicate:**  $!e$  succeeds whenever  $e$  fails

*Example:*

$C \leftarrow B I^* E$

$I \leftarrow !E (C / T)$

$B \leftarrow \underline{C^*}$

$E \leftarrow \underline{^*}$

$T \leftarrow [any\ terminal]$

End marker

$C$  matches “ $(^*ab^*)cd$ ”

$C$  matches “ $(^*a(^*b^*)c^*)$ ”

$C$  fails on “ $(^*a(^*b^*))$ ”

# Syntactic Predicates

**And-predicate:**  $\&e$  succeeds whenever  $e$  does,  
*but consumes no input* [Parr '94, '95]

**Not-predicate:**  $!e$  succeeds whenever  $e$  fails

*Example:*

$C \leftarrow B I^* E$   
 $\rightarrow I \leftarrow !E (C / T)$   
 $B \leftarrow \underline{(}^*$   
 $E \leftarrow \underline{*)}$   
 $T \leftarrow [any\ terminal]$

$C$  matches “ $(^*ab^*)cd$ ”

$C$  matches “ $(^*a(^*b^*)c^*)$ ”

$C$  fails on “ $(^*a(^*b^*))$ ”



# Syntactic Predicates

**And-predicate:**  $\&e$  succeeds whenever  $e$  does, but consumes no input [Parr '94, '95]

**Not-predicate:**  $!e$  succeeds whenever  $e$  fails

*Example:*

Only if an end marker *doesn't* start here...

$C \leftarrow B I^* E$   
 $\rightarrow I \leftarrow \textcircled{!E} (C / T)$   
 $B \leftarrow \underline{(}^*$   
 $E \leftarrow \underline{*)}$   
 $T \leftarrow [any\ terminal]$

$C$  matches “ $(^*ab^*)cd$ ”

$C$  matches “ $(^*a(^*b^*)c^*)$ ”

$C$  fails on “ $(^*a(^*b^*))$ ”

# Syntactic Predicates

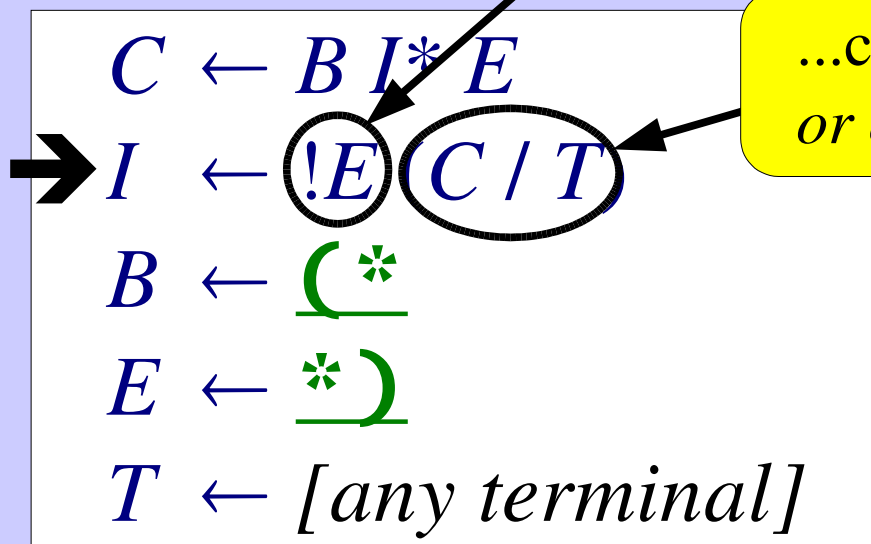
**And-predicate:**  $\&e$  succeeds whenever  $e$  does, but consumes no input [Parr '94, '95]

**Not-predicate:**  $!e$  succeeds whenever  $e$  fails

*Example:*

Only if an end marker *doesn't* start here...

...consume a nested comment, or else consume any single character.



$C$  matches  $( * a ( * b * ) c * )$   
 $C$  matches  $( * a ( * b * ) c * )$   
 $C$  fails on  $( * a ( * b * )$

# Syntactic Predicates

**And-predicate:**  $\&e$  succeeds whenever  $e$  does,  
*but consumes no input* [Parr '94, '95]

**Not-predicate:**  $!e$  succeeds whenever  $e$  fails

*Example:*

$C \leftarrow B I^* E$

$I \leftarrow !E (C / T)$

$B \leftarrow \underline{(}^*$

$E \leftarrow \underline{*)}$

$T \leftarrow [any\ terminal]$

$C$  matches “ $(^*ab^*)cd$ ”

$C$  matches “ $(^*a(^*b^*)c^*)$ ”

$C$  fails on “ $(^*a(^*b^*))$ ”

# Unified Grammars

PEGs can express both *lexical and hierarchical* syntax of realistic languages in one grammar

- *Example (in paper):*  
Complete self-describing PEG in 2/3 column
- *Example (on web):*  
Unified PEG for Java language

# Lexical/Hierarchical Interplay

Unified grammars create new design opportunities

*Example:*

$$\begin{aligned} E &\leftarrow S / \underline{( E )} / \dots \\ S &\leftarrow \underline{“ C* “} \\ C &\leftarrow \underline{\backslash ( E )} / \\ &\quad \underline{! “ ! \backslash T} \\ T &\leftarrow [any\ terminal] \end{aligned}$$

To get Unicode “ $\forall$ ”,  
instead of “`\u2200`”,  
write “`\(0x2200)`”  
or “`\(8704)`”  
or “`\(FOR_ALL)`”

# Lexical/Hierarchical Interplay

Unified grammars create new design opportunities

*Example:*

General-purpose expression syntax

$E \leftarrow S / ( E ) / \dots$

$S \leftarrow \_ C^* \_$

$C \leftarrow \_ ( E ) /$   
 $\_ ! \_ ! \_ T$

$T \leftarrow [any\ terminal]$

To get Unicode “ $\forall$ ”,  
instead of “ $\backslash u2200$ ”,  
write “ $\backslash (0x2200)$ ”  
or “ $\backslash (8704)$ ”  
or “ $\backslash (FOR\_ALL)$ ”

# Lexical/Hierarchical Interplay

Unified grammars create new design opportunities

*Example:*

String literals

$E \leftarrow S / ( E ) \dots$

$S \leftarrow \_ " C * \_ "$

$C \leftarrow \_ ( E ) /$   
 $\_ ! " ! \_ T$

$T \leftarrow [any\ terminal]$

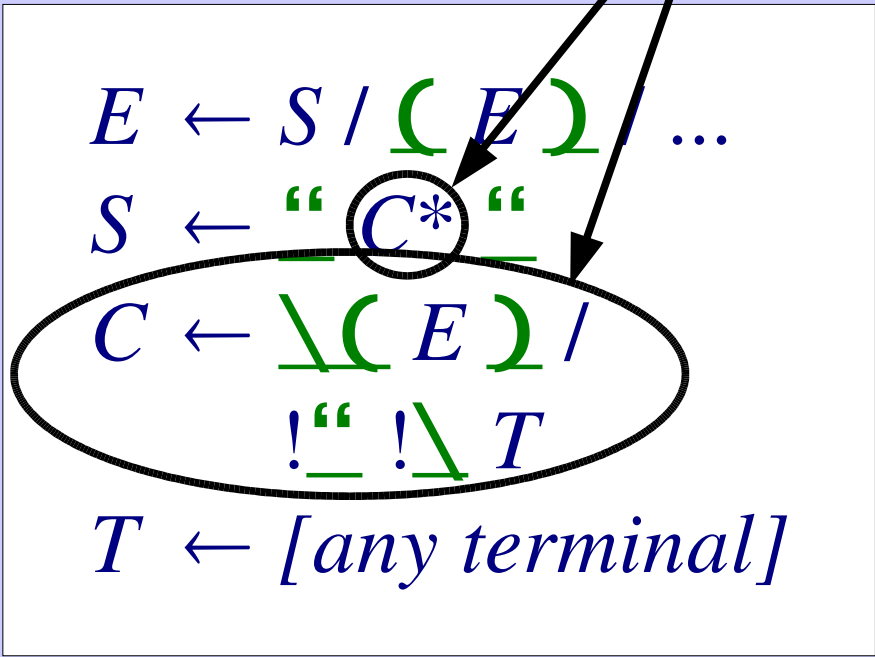
To get Unicode “ $\forall$ ”,  
instead of “ $\backslash u2200$ ”,  
write “ $\backslash (0x2200)$ ”  
or “ $\backslash (8704)$ ”  
or “ $\backslash (FOR\_ALL)$ ”

# Lexical/Hierarchical Interplay

Unified grammars create new design opportunities

*Example:*

Quotable characters



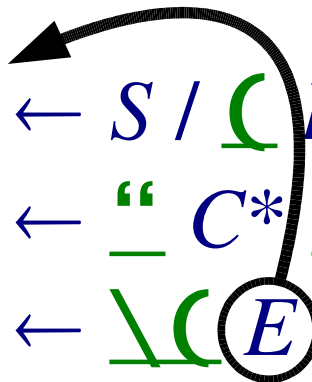
To get Unicode “ $\forall$ ”,  
instead of “ $\backslash u2200$ ”,  
write “ $\backslash (0x2200)$ ”  
or “ $\backslash (8704)$ ”  
or “ $\backslash (FOR\_ALL)$ ”



# Lexical/Hierarchical Interplay

Unified grammars create new design opportunities

*Example:*



$E \leftarrow S / (E) / \dots$   
 $S \leftarrow \_ C^* \_$   
 $C \leftarrow \_ (E) \_ /$   
 $\_ ! \_ ! \_ T$   
 $T \leftarrow [any\ terminal]$

To get Unicode “ $\forall$ ”,  
instead of “`\u2200`”,  
write “`\(0x2200)`”  
or “`\(8704)`”  
or “`\(FOR_ALL)`”

# Formal Properties of PEGs

- Express all deterministic languages -  $LR(k)$
- Closed under union, intersection, complement
- Some non-context free languages, e.g.,  $\underline{a}^n \underline{b}^n \underline{c}^n$
- Undecidable whether  $L(G) = \emptyset$
- Predicate operators can be eliminated
  - ...but the process is non-trivial!

# Minimalist Forms

Predicate-free PEG



TS [Birman '70/'73]

TDPL [Aho '72]

Any PEG



gTS [Birman '70/'73]

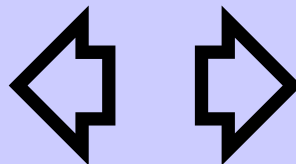
GTDPL [Aho '72]

$A \leftarrow \varepsilon$

$A \leftarrow \underline{\mathbf{a}}$

$A \leftarrow f$

$A \leftarrow BC / D$



$A \leftarrow \varepsilon$

$A \leftarrow \underline{\mathbf{a}}$

$A \leftarrow f$

$A \leftarrow B[C, D]$

# Formal Contributions

- Generalize TDPL/GTDPL with more expressive *structured parsing expression* syntax
- *Negative* syntactic predicate -  $!e$
- *Predicate elimination* transformation
  - Intermediate stages depend on generalized parsing expressions
- *Proof of equivalence* of TDPL and GTDPL

# What *can't* PEGs express directly?

- Ambiguous languages

*That's what CFGs were designed for!*

- Globally disambiguated languages?

–  $\{\underline{a}, \underline{b}\}^n \underline{a} \{\underline{a}, \underline{b}\}^n$  ?

- State- or semantic-dependent syntax

– C, C++ typedef symbol tables

– Python, Haskell, ML layout

# Generating Parsers from PEGs

## *Recursive-descent parsing*

- ☞ Simple & direct, but exponential-time if not careful

## *Packrat parsing [Birman '70/'73, Ford '02]*

- ☞ Linear-time, but can consume substantial storage

## *Classic LL/LR algorithms?*

- ☞ Grammar restrictions, but both time- & space-efficient

# Conclusion

PEGs model common parsing practices

- *Prioritized choice, greedy rules, syntactic predicates*

PEGs naturally complement CFGs

- CFG: *generative* system, for *ambiguous* languages
- PEG: *recognition-based*, for *unambiguous* languages

For more info:

**<http://pdos.lcs.mit.edu/~baford/packrat>**

(or **Google** for “Packrat Parsing”)