Parsing Expression Grammars: A Recognition-Based Syntactic Foundation

Bryan Ford
Massachusetts Institute of Technology
Cambridge, MA
baford@mit.edu

Abstract

For decades we have been using Chomsky's generative system of grammars, particularly context-free grammars (CFGs) and regular expressions (REs), to express the syntax of programming languages and protocols. The power of generative grammars to express ambiguity is crucial to their original purpose of modelling natural languages, but this very power makes it unnecessarily difficult both to express and to parse machine-oriented languages using CFGs. Parsing Expression Grammars (PEGs) provide an alternative, recognition-based formal foundation for describing machine-oriented syntax, which solves the ambiguity problem by not introducing ambiguity in the first place. Where CFGs express nondeterministic choice between alternatives, PEGs instead use *prioritized choice*. PEGs address frequently felt expressiveness limitations of CFGs and REs, simplifying syntax definitions and making it unnecessary to separate their lexical and hierarchical components. A linear-time parser can be built for any PEG, avoiding both the complexity and fickleness of LR parsers and the inefficiency of generalized CFG parsing. While PEGs provide a rich set of operators for constructing grammars, they are reducible to two minimal recognition schemas developed around 1970, TS/TDPL and gTS/GTDPL, which are here proven equivalent in effective recognition power.

Categories and Subject Descriptors

F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars and Other Rewriting Systems—*Grammar types*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Syntax*; D.3.4 [**Programming Languages**]: Processors—*Parsing*

General Terms

Languages, Algorithms, Design, Theory

Keywords

Context-free grammars, regular expressions, parsing expression grammars, BNF, lexical analysis, unified grammars, scannerless parsing, packrat parsing, syntactic predicates, TDPL, GTDPL

POPL'04, January 14–16, 2004, Venice, Italy.

1 Introduction

Most language syntax theory and practice is based on *generative* systems, such as regular expressions and context-free grammars, in which a language is defined formally by a set of rules applied recursively to generate strings of the language. A *recognition-based* system, in contrast, defines a language in terms of rules or predicates that decide whether or not a given string is in the language. Simple languages can be expressed easily in either paradigm. For example, $\{s \in \mathtt{a}^* \mid s = (\mathtt{aa})^n\}$ is a generative definition of a trivial language over a unary character set, whose strings are "constructed" by concatenating pairs of a's. In contrast, $\{s \in \mathtt{a}^* \mid (|s| \ mod\ 2 = 0)\}$ is a recognition-based definition of the same language, in which a string of a's is "accepted" if its length is even.

While most language theory adopts the generative paradigm, most practical language applications in computer science involve the recognition and structural decomposition, or *parsing*, of strings. Bridging the gap from generative definitions to practical recognizers is the purpose of our ever-expanding library of parsing algorithms with diverse capabilities and trade-offs [9].

Chomsky's generative system of grammars, from which the ubiquitous context-free grammars (CFGs) and regular expressions (REs) arise, was originally designed as a formal tool for modelling and analyzing natural (human) languages. Due to their elegance and expressive power, computer scientists adopted generative grammars for describing machine-oriented languages as well. The ability of a CFG to express ambiguous syntax is an important and powerful tool for natural languages. Unfortunately, this power gets in the way when we use CFGs for machine-oriented languages that are intended to be precise and unambiguous. Ambiguity in CFGs is difficult to avoid even when we want to, and it makes general CFG parsing an inherently super-linear-time problem [14, 23].

This paper develops an alternative, recognition-based formal foundation for language syntax, *Parsing Expression Grammars* or PEGs. PEGs are stylistically similar to CFGs with RE-like features added, much like Extended Backus-Naur Form (EBNF) notation [30, 19]. A key difference is that in place of the unordered choice operator '|' used to indicate alternative expansions for a nonterminal in EBNF, PEGs use a *prioritized* choice operator '/'. This operator lists alternative patterns to be tested *in order*, unconditionally using the first successful match. The EBNF rules 'A → a b | a' and 'A → a | a b' are equivalent in a CFG, but the PEG rules 'A ← a b / a' and 'A ← a / a b' are different. The second alternative in the latter PEG rule will never succeed because the first choice is always taken if the input string to be recognized begins with 'a'.

A PEG may be viewed as a formal description of a top-down parser. Two closely related prior systems upon which this work is based were developed primarily for the purpose of studying top-down parsers [4, 5]. PEGs have far more syntactic expressiveness than the $LL(k)$ language class typically associated with top-down parsers, however, and can express all deterministic $LR(k)$ languages and many others, including some non-context-free languages. Despite their considerable expressive power, *all* PEGs can be parsed in linear time using a tabular or memoizing parser [8]. These properties strongly suggest that CFGs and PEGs define incomparable language classes, although a formal proof that there are context-free languages not expressible via PEGs appears surprisingly elusive.

Besides developing PEGs as a formal system, this paper presents pragmatic examples that demonstrate their suitability for describing realistic machine-oriented languages. Since these languages are generally designed to be unambiguous and linearly readable in the first place, the recognition-oriented nature of PEGs creates a natural affinity in terms of syntactic expressiveness and parsing efficiency.

The primary contribution of this work is to provide language and protocol designers with a new tool for describing syntax that is both practical and rigorously formalized. A secondary contribution is to render this formalism more amenable to further analysis by proving its equivalence to two simpler formal systems, originally named TS ("TMG recognition scheme") and gTS ("generalized TS") by Alexander Birman [4, 5], in reference to an early syntax-directed compiler-compiler. These systems were later called TDPL ("Top-Down Parsing Language") and GTDPL ("Generalized TDPL") respectively by Aho and Ullman [3]. By extension we prove that with minor caveats TS/TDPL and gTS/GTDPL are equivalent in recognition power, an unexpected result contrary to prior conjectures [5].

The rest of this paper is organized as follows. Section 2 first defines PEGs informally and presents examples of their usefulness for describing practical machine-oriented languages. Section 3 then defines PEGs formally and proves some of their important properties. Section 4 presents useful transformations on PEGs and proves the main result regarding the reducibility of PEGs to TDPL and GTDPL. Section 5 outlines open problems for future study, Section 6 describes related work, and Section 7 concludes.

```
# Hierarchical syntax
Grammar    <- Spacing Definition+ EndOfFile
Definition <- Identifier LEFTARROW Expression

Expression <- Sequence (SLASH Sequence)*
Sequence   <- Prefix*
Prefix     <- (AND / NOT)? Suffix
Suffix     <- Primary (QUESTION / STAR / PLUS)?
Primary    <- Identifier !LEFTARROW
            / OPEN Expression CLOSE
            / Literal / Class / DOT

# Lexical syntax
Identifier <- IdentStart IdentCont* Spacing
IdentStart <- [a-zA-Z_]
IdentCont  <- IdentStart / [0-9]

Literal    <- ['] (!['] Char)* ['] Spacing
            / ["] (!["] Char)* ["] Spacing
Class      <- '[' (!']' Range)* ']' Spacing
Range      <- Char '-' Char / Char
Char       <- '\\' [nrt'"\[\]\\]
            / '\\' [0-2][0-7][0-7]
            / '\\' [0-7][0-7]?
            / !'\\' .

LEFTARROW  <- '<-' Spacing
SLASH      <- '/' Spacing
AND        <- '&' Spacing
NOT        <- '!' Spacing
QUESTION   <- '?' Spacing
STAR       <- '*' Spacing
PLUS       <- '+' Spacing
OPEN       <- '(' Spacing
CLOSE      <- ')' Spacing
DOT        <- '.' Spacing

Spacing    <- (Space / Comment)*
Comment    <- '#' (!EndOfLine .)* EndOfLine
Space      <- ' ' / '\t' / EndOfLine
EndOfLine  <- '\r\n' / '\n' / '\r'
EndOfFile  <- !.
```

Figure 1. PEG formally describing its own ASCII syntax

2 Parsing Expression Grammars

Figure 1 shows an example PEG, which precisely specifies a practical syntax for PEGs using the ASCII character set. The example PEG describes its own complete syntax including all lexical characteristics. Most elements of the grammar should be immediately recognizable to anyone familiar with CFGs and regular expressions. The grammar consists of a set of definitions of the form 'A <- e', where A is a nonterminal and e is a *parsing expression*. The operators for constructing parsing expressions are summarized in Table 1.

Single or double quotes delimit string literals, and square brackets indicate character classes. Literals and character classes can contain C-like escape codes, and character classes can include ranges such as 'a-z'. The constant '.' matches any single character.

The sequence expression 'e_1 e_2' looks for a match of e_1 immediately followed by a match of e_2, backtracking to the starting point if either pattern fails. The choice expression 'e_1 / e_2' first attempts pattern e_1, then attempts e_2 from the same starting point if e_1 fails.

The ?, *, and + operators behave as in common regular expression syntax, except that they are "greedy" rather than nondeterministic. The option expression 'e?' unconditionally "consumes" the text matched by e if e succeeds, and the repetition expressions 'e*' and 'e+' always consume as many successive matches of e as possible. The expression 'a* a' for example can never match any string. Longest-match parsing is almost always the desired behavior where options or repetition occur in practical machine-oriented languages. Many forms of non-greedy behavior are still available in PEGs when desired, however, through the use of predicates.

The operators & and ! denote *syntactic predicates* [20], which provide much of the practical expressive power of PEGs. The expression '&e' attempts to match pattern e, then unconditionally backtracks to the starting point, preserving only the knowledge of whether e succeeded or failed to match. Conversely, the expression '!e' fails if e succeeds, but succeeds if e fails. For example, the subexpression '!EndOfLine .' in the definition for Comment in Figure 1, matches any single character as long as the nonter-

Operator	Type	Precedence	Description
`' '`	primary	5	Literal string
`" "`	primary	5	Literal string
`[ ]`	primary	5	Character class
`.`	primary	5	Any character
(e)	primary	5	Grouping
$e?$	unary suffix	4	Optional
$e*$	unary suffix	4	Zero-or-more
$e+$	unary suffix	4	One-or-more
$\&e$	unary prefix	3	And-predicate
$!e$	unary prefix	3	Not-predicate
$e_1\ e_2$	binary	2	Sequence
$e_1\ /\ e_2$	binary	1	Prioritized Choice

Table 1. Operators for Constructing Parsing Expressions

minal `EndOfLine` does *not* match starting at the same position. The expression '`Identifier !LEFTARROW`' in the definition for `Primary`, in contrast, matches any `Identifier` that is not *followed by* a `LEFTARROW`. This latter predicate prevents the right-hand-side `Expression` at the beginning of one `Definition` from consuming the left-hand-side `Identifier` of the next `Definition`, eliminating the need for an explicit delimiter. Predicates can involve arbitrary parsing expressions requiring any amount of "lookahead."

2.1 Unified Language Definitions

Most conventional syntax descriptions are split into two parts: a CFG to specify the hierarchical portion, and a set of regular expressions defining the lexical elements to serve as terminals for the CFG. CFGs are unsuitable for lexical syntax because they cannot directly express many common idioms, such as the greedy rule that usually applies to identifiers and numbers, or "negative" syntax such as the `Literal` rule above, in which quoted string literals may contain any character *except* the quote character. Regular expressions cannot describe recursive syntax, however, such as large expressions constructed inductively from smaller expressions.

Neither of these difficulties exist with PEGs, as demonstrated by the unified example grammar. The greedy nature of the repetition operator ensures that a sequence of letters can only be interpreted as a single `Identifier` and not as two or more immediately adjacent, shorter ones. Not-predicates describe the appropriate negative constraints on the elements that can appear in literals, character classes, and comments. The last '`!'\\' .`' alternative in the definition of `Char` ensures that the backslash cannot be used in a literal or character class except as part of an escape sequence.

Each definition in the example grammar that represents a distinct lexical "token," such as `Identifier`, `Literal`, or `LEFTARROW`, uses the `Spacing` nonterminal to "consume" any whitespace and/or comments immediately following the token. The definition of `Grammar` also starts with `Spacing` in order to allow whitespace at the beginning of the file. Associating whitespace with each immediately preceding token is a convenient convention for PEGs, but whitespace could just as easily be associated with the *following* token by referring to `Spacing` at the *beginning* of each token definition. Whitespace could even be treated as a separate kind of token, consistent with lexical traditions, but doing so in a unified grammar such as this one would require many explicit references to `Spacing` throughout the hierarchical portion of the syntax.

2.2 New Syntax Design Choices

Besides being able to express many existing machine-oriented languages in a concise and unified grammar, PEGs also create new possibilities for language syntax design. Consider for example a well-known problem with C++ syntax involving nested template type expressions:

```
vector<vector<float> > MyMatrix;
```

The space between the two right angle brackets is required because the C++ scanner is oblivious to the language's hierarchical syntax, and would otherwise interpret the '>>' incorrectly as a right shift operator. In a language described by a unified PEG, however, it is easy to define the language to permit a '>>' sequence to be interpreted as either one token or two depending on its context:

```
TemplType <- PrimType (LANGLE TemplType RANGLE)?
ShiftExpr <- PrimExpr (ShiftOper PrimExpr)*
ShiftOper <- LSHIFT / RSHIFT

LANGLE    <- '<' Spacing
RANGLE    <- '>' Spacing
LSHIFT    <- '<<' Spacing
RSHIFT    <- '>>' Spacing
```

Such permissiveness can create unexpected syntactic subtleties, of course, and caution and good taste are in order: a powerful syntax description paradigm also means more rope for the careless language designer to hang himself with. The traditional behavior for operator tokens is still easily expressible if desired, as follows:

```
LANGLE      <- !LSHIFT '<' Spacing
RANGLE      <- !RSHIFT '>' Spacing
LSHIFT      <- '<<' Spacing
RSHIFT      <- '>>' Spacing
```

Freeing lexical syntax from the restrictions of regular expressions also enables tokens to have hierarchical characteristics, or even to refer back to the hierarchical portion of the language. Pascal-like nestable comments, for example, cannot be described by a regular expression but are easily expressed in a PEG:

```
Comment <- '(*' (Comment / !'*)' .)* '*)'
```

Character and string literals in most programming languages permit escape sequences of some kind, to express either special characters or dynamic string substitutions. These escapes usually have a highly restrictive syntax, however. A language described by a unified PEG could permit the use of arbitrary expressions in such escapes, taking advantage of the full power of the language's expression syntax:

```
Expression <- ...
Primary    <- Literal / ...

Literal    <- ["] (!["] Char)* ["]
Char       <- '\\(' Expression ')'
            / !'\\' .
```

In place of the Java string literal `"\u2200"` containing the Unicode math symbol '∀', for example, the literal could be written `"\(0x2200)"`, `"\(8704)"`, or even `"\(Unicode.FOR_ALL)"`, where `FOR_ALL` is a constant defined in a class named `Unicode`.

2.3 Priorities, Not Ambiguities

The specification flexibility provided by PEGs, and the new syntax design choices they create, are not limited to the lexical portions of a language. Many sensible syntactic constructs are inherently ambiguous when expressed in a CFG, commonly leading language designers to abandon syntactic formality and rely on informal meta-rules to solve these problems. The ubiquitous "dangling ELSE" problem is a classic example, traditionally requiring either an informal meta-rule or severe expansion and obfuscation of the CFG. The correct behavior is easily expressed with the prioritized choice operator in a PEG:

```
Statement <- IF Cond THEN Statement ELSE Statement
           / IF Cond THEN Statement
           / ...
```

The syntax of C++ contains ambiguities that cannot be resolved with any amount of CFG rewriting, in which certain token sequences can be interpreted as either a statement or a definition. The language specification [25] resolves this problem with the informal meta-rule that such a sequence is always interpreted as a definition if possible. Similarly, the syntax of lambda abstractions, `let` expressions, and conditionals in Haskell [11] is unresolvably ambiguous in the CFG paradigm, and is handled in the Haskell specification with an informal "longest match" meta-rule. PEGs provide the necessary tools—prioritized choice, greedy repetition, and syntactic predicates—to define precisely how to resolve such ambiguities.

These tools do not make language syntax design easy, of course. In place of having to determine whether two possible alternatives in a CFG are ambiguous, PEGs present language designers with the analogous challenge of determining whether two alternatives in a '/' expression can be reordered without affecting the language. This question is often obvious, but sometimes is not, and is undecidable in general. As with discovering ambiguity in CFGs, however, we have the hope of finding automatic algorithms to identify order sensitivity or insensitivity conservatively in common situations.

2.4 Quirks and Limitations

If the definition of `Grammar` in Figure 1 did not reference `EndOfFile` at the end, then any ASCII file starting with at least one correct `Definition` would be interpreted as a "correct" grammar, even if the file has unreadable garbage at the end. This peculiarity arises from the fact that a parsing expression in a PEG can "succeed" without consuming all input text. We address this minor issue with the `EndOfFile` nonterminal, defined by the predicate expression '`!.`', which "matches" the end-of-file by failing if any character is available and succeeding otherwise.

Both *left* and *right recursion* are permissible in CFGs, but as with top-down parsing in general, left recursion is unavailable in PEGs because it represents a degenerate loop. For example, the CFG rules 'A → a A | a' and 'A → A a | a' represent a series of 'a's in a CFG, but the PEG rule 'A ← A a / a' is degenerate because it indicates that in order to recognize nonterminal A, a parser must first recognize nonterminal A... This restriction applies not only to direct left recursion as in this example, but also to indirect or mutual left recursion involving several nonterminals. Since both left and right recursion in a CFG merely represent repetition, however, and repetition is easier to express in a PEG using repetition operators, this limitation is not a serious problem in practice.

Like a CFG, a PEG is a purely syntactic formalism, not by itself capable of expressing languages whose syntax depends on *semantic predicates* [20]. Although the Java language can be described as a single unified PEG [7], C and C++ parsers require an incrementally constructed symbol table to distinguish between ordinary identifiers and `typedef`-defined type identifiers. Haskell uses a special stage in the "syntactic pipeline," inserted between the scanner and parser, to implement the language's layout-sensitive features.

3 Formal Development of PEGs

In this section we define PEGs formally and explore key properties. Many of these properties and their proofs were inspired by those of the closely related TS/TDPL and gTS/GTDPL systems [4, 5, 3], although the formulation of PEGs is substantially different.

3.1 Definition of a PEG

In Figure 1 we used a "concrete" ASCII-based syntax for PEGs to illustrate the characteristics of PEGs for practical language description purposes. For formal analysis, however, it is more convenient to use an abstract syntax for PEGs that represents only its essential structure. We begin therefore by defining this abstract syntax.

Definition: A *parsing expression grammar* (PEG) is a 4-tuple $G = (V_N, V_T, R, e_S)$, where V_N is a finite set of nonterminal symbols, V_T is a finite set of terminal symbols, R is a finite set of rules, e_S is a parsing expression termed the *start expression*, and $V_N \cap V_T = \emptyset$. Each rule $r \in R$ is a pair (A, e), which we write $A \leftarrow e$, where $A \in V_N$ and e is a parsing expression. For any nonterminal A, there is exactly one e such that $A \leftarrow e \in R$. R is therefore a function from nonterminals to expressions, and we write $R(A)$ to denote the unique expression e such that $A \leftarrow e \in R$.

We define *parsing expressions* inductively as follows. If e, e_1, and e_2 are parsing expressions, then so is:

1. ε, the empty string
2. a, any terminal, where $a \in V_T$.
3. A, any nonterminal, where $A \in V_N$.
4. $e_1 e_2$, a sequence.
5. e_1 / e_2, prioritized choice.
6. e^*, zero-or-more repetitions.
7. $!e$, a not-predicate.

All subsequent use of the unqualified term "grammar" refers specifically to parsing expression grammars as defined here, and the unqualified term "expression" refers to parsing expressions. We use the variables a, b, c, d to represent terminals, A, B, C, D for nonterminals, x, y, z for strings of terminals, and e for parsing expressions.

The structural requirement that R be a function, mapping each nonterminal in V_N to a unique parsing expression, precludes the possibility of expressions in the grammar containing "undefined references," or *subroutine failures* [5].

The *expression set* $E(G)$ of G is the set containing the start expression e_S, the expressions used in all grammar rules, and all subexpressions of those expressions.

A *repetition-free* grammar is a grammar whose expression set contains only expressions constructed without using rule 6 above. A *predicate-free* grammar is one whose expression set contains only expressions constructed without using rule 7.

3.2 Desugaring the Concrete Syntax

The abstract syntax does not include character classes, the "any character" constant '`.`', the option operator '`?`', the one-or-more-repetitions operator '`+`', or the and-predicate operator '`&`', all of which appear in the concrete syntax. We treat these features of the concrete syntax as "syntactic sugar," reducing them to abstract parsing expressions using local substitutions as follows:

- We consider the '`.`' expression in the concrete syntax to be a character class containing all of the terminals in V_T.
- If $a_1, a_2, \ldots, a_n$ are all of the terminals listed in a character class expression in the concrete syntax, after expanding any ranges, then we desugar this character class expression to the abstract syntax expression $a_1/a_2/\ldots/a_n$.
- We desugar an option expression e`?` in the concrete syntax to e_d/ε, where e_d is the desugaring of e.
- We desugar a one-or-more-repetitions expression e`+` to $e_d e_d^*$, where e_d is the desugaring of e.
- We desugar an and-predicate `&`e to $!(!e_d)$, where e_d is the desugaring of e.

3.3 Interpretation of a Grammar

Definition: To formalize the syntactic meaning of a grammar $G = (V_N, V_T, R, e_S)$, we define a relation $\Rightarrow_G$ from pairs of the form (e,x) to pairs of the form (n,o), where e is a parsing expression, $x \in V_T^*$ is an input string to be recognized, $n \geq 0$ serves as a "step counter," and $o \in V_T^* \cup \{f\}$ indicates the result of a recognition attempt. The "output" o of a successful match is the portion of the input string recognized and "consumed," while a distinguished symbol $f \notin V_T$ indicates failure. For $((e,x),(n,o)) \in \Rightarrow_G$ we will write $(e,x) \Rightarrow (n,o)$, with the reference to G being implied. We define $\Rightarrow_G$ inductively as follows:

1. **Empty:** $(\varepsilon, x) \Rightarrow (1, \varepsilon)$ for any $x \in V_T^*$.
2. **Terminal (success case):** $(a, ax) \Rightarrow (1, a)$ if $a \in V_T$, $x \in V_T^*$.
3. **Terminal (failure case):** $(a, bx) \Rightarrow (1, f)$ if $a \neq b$, and $(a, \varepsilon) \Rightarrow (1, f)$.
4. **Nonterminal:** $(A, x) \Rightarrow (n+1, o)$ if $A \leftarrow e \in R$ and $(e,x) \Rightarrow (n,o)$.
5. **Sequence (success case):** If $(e_1, x_1x_2y) \Rightarrow (n_1, x_1)$ and $(e_2, x_2y) \Rightarrow (n_2, x_2)$, then $(e_1e_2, x_1x_2y) \Rightarrow (n_1+n_2+1, x_1x_2)$. Expressions e_1 and e_2 are matched in sequence, and if each succeeds and consumes input portions x_1 and x_2 respectively, then the sequence succeeds and consumes the string x_1x_2.
6. **Sequence (failure case 1):** If $(e_1, x) \Rightarrow (n_1, f)$, then $(e_1e_2, x) \Rightarrow (n_1+1, f)$. If e_1 is tested and fails, then the sequence e_1e_2 fails without attempting e_2,
7. **Sequence (failure case 2):** If $(e_1, x_1y) \Rightarrow (n_1, x_1)$ and $(e_2, y) \Rightarrow (n_2, f)$, then $(e_1e_2, x_1y) \Rightarrow (n_1+n_2+1, f)$. If e_1 succeeds but e_2 fails, then the sequence expression fails.
8. **Alternation (case 1):** If $(e_1, xy) \Rightarrow (n_1, x)$, then $(e_1/e_2, xy) \Rightarrow (n_1+1, x)$. Alternative e_1 is first tested, and if it succeeds, the expression e_1/e_2 succeeds without testing e_2.
9. **Alternation (case 2):** If $(e_1, x) \Rightarrow (n_1, f)$ and $(e_2, x) \Rightarrow (n_2, o)$, then $(e_1/e_2, x) \Rightarrow (n_1+n_2+1, o)$. If e_1 fails, then e_2 is tested and its result is used instead.
10. **Zero-or-more repetitions (repetition case):** If $(e, x_1x_2y) \Rightarrow (n_1, x_1)$ and $(e^*, x_2y) \Rightarrow (n_2, x_2)$, then $(e^*, x_1x_2y) \Rightarrow (n_1 + n_2 + 1, x_1x_2)$.
11. **Zero-or-more repetitions (termination case):** If $(e,x) \Rightarrow (n_1, f)$, then $(e^*, x) \Rightarrow (n_1+1, \varepsilon)$.
12. **Not-predicate (case 1):** If $(e, xy) \Rightarrow (n, x)$, then $(!e, xy) \Rightarrow (n+1, f)$. If expression e succeeds consuming input x, then the syntactic predicate $!e$ fails.
13. **Not-predicate (case 2):** If $(e,x) \Rightarrow (n,f)$, then $(!e,x) \Rightarrow (n+1, \varepsilon)$. If e fails, then $!e$ succeeds but consumes nothing.

We define a relation $\Rightarrow_G^+$ from pairs (e,x) to outcomes o, such that $(e,x) \Rightarrow^+ o$ iff an n exists such that $(e,x) \Rightarrow (n,o)$.

If $(e,x) \Rightarrow^+ y$ for $y \in V_T^*$, we say that *e matches x* in G. If $(e,x) \Rightarrow^+ f$, we say that *e fails on x* in G. The *match set* $M_G(e)$ of expression e in G is the set of inputs x such that e matches x in G.

An expression e *handles* a string $x \in V_T^*$ if it either matches or fails on x in G. A grammar G *handles* string x if its start expression e_S handles x. G is *complete* if it handles all strings $x \in V_T^*$.

Two expressions e_1 and e_2 are *equivalent*, written $e_1 \asymp e_2$, if $(e_1, x) \Rightarrow^+ o$ implies $(e_2, x) \Rightarrow^+ o$ and vice versa. The resulting step counts need not be the same.

Theorem: If $(e,x) \Rightarrow (n,y)$, then y is a prefix of x: $\exists z (x = yz)$.

Proof: By induction on an integer variable $m \geq 0$, using as the induction hypothesis the proposition that the desired property holds for all $e, x, n \leq m$, and y.

Theorem: If $(e,x) \Rightarrow (n_1, o_1)$ and $(e,x) \Rightarrow (n_2, o_2)$, then $n_1 = n_2$ and $o_1 = o_2$. That is, the relation $\Rightarrow_G$ is a function.

Proof: By induction on a variable $m \geq 0$, using the induction hypothesis that the proposition holds for all $e, x, n_1 \leq m, n_2 \leq m, o_1$, and o_2. This induction technique will subsequently be referred to simply as *induction on step counts* of $\Rightarrow_G$.

Theorem: A repetition expression e^* does not handle any input string x on which e succeeds without consuming input: for any $x \in V_T^*$, if $(e,x) \Rightarrow (n_1, \varepsilon)$, then $(e^*, x) \not\Rightarrow (n_2, o_2)$ for any n_2, o_2. We call this the $*$*-loop condition*.

Proof: By induction on step counts.

3.4 Language Properties

This section describes properties of *parsing expression languages* (PELs), the class of languages that can be expressed by PEGs. PELs are closed under union, intersection, and complement. It is undecidable in general whether a PEG represents a nonempty language, or whether two PEGs represent the same language.

Definition: The *language* $L(G)$ of a PEG $G = (V_N, V_T, R, e_S)$ is the set of strings $x \in V_T^*$ for which the start expression e_S matches x.

Note that the start expression e_S only needs to *succeed* on input string x for x to be included in $L(G)$; e_S need not consume all of string x. For example, the trivial grammar $(\{\}, V_T, \{\}, \varepsilon)$ recognizes the language V_T^* and not just the empty string, because the start expression ε always succeeds even though it does not examine or consume any input. This definition contrasts with TS and gTS, in which partially consumed input strings are excluded from the language and classified as *partial-acceptance failures* [5].

Definition: A language L over an alphabet V_T is a *parsing expression language (PEL)* iff there exists a parsing expression grammar G whose language is L.

Theorem: The class of parsing expression languages is closed under union, intersection, and complement.

Proof: Suppose we have two grammars $G_1 = (V_N^1, V_T, R^1, e_S^1)$ and $G_2 = (V_N^2, V_T, R^2, e_S^2)$ respectively, describing languages $L(G_1)$ and $L(G_2)$. Assume without loss of generality, that $V_N^1 \cap V_N^2 = \emptyset$, by renaming nonterminals if necessary. We can form a new grammar $G' = (V_N^1 \cup V_N^2, V_T, R^1 \cup R^2, e'_S)$, where e'_S is one of the following:

- If $e'_S = e_S^1 \ / \ e_S^2$, then $L(G') = L(G_1) \cup L(G_2)$.
- If $e'_S = \&e_S^1 \ e_S^2$, then $L(G') = L(G_1) \cap L(G_2)$.
- If $e'_S = !e_S^1$, then $L(G') = V_T^* - L(G_1)$.

Theorem: The class of PELs includes non-context-free languages.

Proof: The classic example language $a^n b^n c^n$ is not context-free, but we can recognize it with a PEG $G = (\{A, B, D\}, \{a, b, c\}, R, D)$, where R contains the following definitions:

$$\begin{array}{lll} A & \leftarrow & a\,A\,b \ / \ \varepsilon \\ B & \leftarrow & b\,B\,c \ / \ \varepsilon \\ D & \leftarrow & \&(A\ !b)\ a^*\ B\ !. \end{array}$$

Theorem: It is undecidable in general whether the language $L(G)$ of an arbitrary parsing expression grammar G is empty.

Proof: We first prove in the same way as for CFGs [3] that it is undecidable whether the intersection of the languages of two PEGs is empty. Since PELs are closed under intersection, an algorithm to test the emptiness of the language $L(G)$ of any G could be used to test whether $L(G_1) \cap L(G_2)$ is empty, implying that emptiness is undecidable as well.

Given an instance $C = (x_1, y_1), \ldots, (x_n, y_n)$ of Post's correspondence problem over an alphabet Σ, it is known to be undecidable whether there is a non-empty string w that can be built from elements of C such that $w = x_{i_1} x_{i_2} \ldots x_{i_m} = y_{i_1} y_{i_2} \ldots y_{i_m}$, where $1 \le i_j \le n$ for each $1 \le j \le m$.

We build a grammar $G = (V_N, V_T, R, D)$ where $V_N = \{A, B, D\}$, and $V_T = \Sigma \cup \{a_1, \ldots, a_n\}$. The a_i in V_T are distinct terminals not in Σ, which will serve as markers associated with the elements of C. R contains the following three rules:

- $A \leftarrow x_1 A a_1 / x_2 A a_2 / \ldots / x_n A a_n / \varepsilon$
- $B \leftarrow y_1 B a_1 / y_2 B a_2 / \ldots / y_n B a_n / \varepsilon$
- $D \leftarrow \&.\ \&(A\ !.)\ B\ !.$

Nonterminal A matches strings of the form $x_{i_1} x_{i_2} \ldots x_{i_m} a_{i_m} \ldots a_{i_2} a_{i_1}$, while B matches strings of the form $y_{i_1} y_{i_2} \ldots y_{i_m} a_{i_m} \ldots a_{i_2} a_{i_1}$. The nonterminal D uses the and-predicate operator to match only strings matching both A and B, representing solutions to the correspondence problem. The &. at the beginning of the definition of D (desugared appropriately) ensures that empty solutions are not allowed, and the !. after the references to A and B ensure that the complete input is consumed in each case. An algorithm to decide whether $L(G)$ is nonempty could therefore be used to solve the correspondence problem C, yielding the desired result.

Definition: Two PEGs G_1 and G_2 are *equivalent* if they recognize the same language: $L(G_1) = L(G_2)$.

Theorem: The equivalence of two arbitrary PEGs is undecidable.

Proof: An algorithm to decide the equivalence of two PEGs could also be used to decide the non-emptiness problem above, simply by comparing the grammar to be tested against a trivial grammar for the empty language.

3.5 Analysis of Grammars

We often would like to analyze the behavior of a particular grammar over arbitrary input strings. While many interesting properties of PEGs are undecidable in general, conservative analysis proves useful and adequate for many practical purposes.

Theorem: It is undecidable whether an arbitrary grammar is complete: that is, whether it either succeeds or fails on all input strings.

Proof: Suppose we have an arbitrary grammar $G = (V_N, V_T, R, e_S)$, and we define a new grammar $G' = (V'_N, V_T, R', e'_S)$, where $V'_N = V_N \cup \{A\}$, $A \notin V_N$, $R' = R \cup \{A \leftarrow \&e_S\ A\}$, and $e'_S = A$. If G's start expression e_S succeeds on any input string x, then this input will cause a degenerate loop in G' via nonterminal A, so G' is incomplete. If $L(G)$ is empty, however, then G' is complete and also fails on all inputs. An algorithm to decide whether G' is complete would therefore allow us to decide whether G is empty, which has already been shown undecidable.

Definition: We define a relation $\rightharpoonup_G$ consisting of pairs of the form (e, o), where e is an expression and $o \in \{0, 1, f\}$. We will write $e \rightharpoonup o$ for $(e, o) \in \rightharpoonup_G$, with the reference to G being implied. This relation represents an abstract simulation of the $\Rightarrow_G$ relation. If $e \rightharpoonup 0$, then e might succeed on some input string while consuming no input. If $e \rightharpoonup 1$, then e might succeed while consuming at least one terminal. If $e \rightharpoonup f$, then e might fail on some input. We will use the variable s to represent a $\rightharpoonup_G$ outcome of either 0 or 1. We define the simulation relation $\rightharpoonup_G$ inductively as follows:

1. $\varepsilon \rightharpoonup 0$.
2. $a \rightharpoonup 1$.
3. $a \rightharpoonup f$.
4. $A \rightharpoonup o$ if $R_G(A) \rightharpoonup o$.
5. $e_1 e_2 \rightharpoonup 0$ if $e_1 \rightharpoonup 0$ and $e_2 \rightharpoonup 0$.
   $e_1 e_2 \rightharpoonup 1$ if $e_1 \rightharpoonup 1$ and $e_2 \rightharpoonup s$.
   $e_1 e_2 \rightharpoonup 1$ if $e_1 \rightharpoonup s$ and $e_2 \rightharpoonup 1$.
6. $e_1 e_2 \rightharpoonup f$ if $e_1 \rightharpoonup f$.
7. $e_1 e_2 \rightharpoonup f$ if $e_1 \rightharpoonup s$ and $e_2 \rightharpoonup f$.

8. $e_1/e_2 \rightharpoonup s$ if $e_1 \rightharpoonup s$.
9. $e_1/e_2 \rightharpoonup o$ if $e_1 \rightharpoonup f$ and $e_2 \rightharpoonup o$.
10. $e^* \rightharpoonup 1$ if $e \rightharpoonup 1$,
11. $e^* \rightharpoonup 0$ if $e \rightharpoonup f$.
12. $!e \rightharpoonup f$ if $e \rightharpoonup s$.
13. $!e \rightharpoonup 0$ if $e \rightharpoonup f$.

Because this relation does not depend on the input string, and there are a finite number of relevant expressions in a grammar, we can compute this relation over any grammar by applying the above rules iteratively until we reach a fixed point.

Theorem: The relation $\rightharpoonup_G$ summarizes $\Rightarrow_G$ as follows:

- If $(e,x) \Rightarrow_G (n,\varepsilon)$, then $e \rightharpoonup 0$.
- If $(e,x) \Rightarrow_G (n,y)$ and $|y| > 0$, then $e \rightharpoonup 1$.
- If $(e,x) \Rightarrow_G (n,f)$, then $e \rightharpoonup f$.

Proof: By induction over the step counts of the relation $\Rightarrow_G$. The definition rules for $\rightharpoonup_G$ above correspond one-to-one to the rules for $\Rightarrow_G$. The conclusion in each case follows immediately from the inductive hypothesis, except in the cases for the repetition operator, which require the $*$-loop condition theorem from Section 3.3.

3.6 Well-Formed Grammars

A *well-formed* grammar is a grammar that contains no directly or mutually left-recursive rules, such as 'A ← A a / a', which could prevent the grammar from handling any input string. This checkable structural property implies completeness, while being permissive enough for most purposes. A grammar can have left-recursive rules but still be complete if its degenerate loops are actually unreachable, but we have little need for such grammars in practice.

Definition: We define an inductive set WF_G as follows. We write $WF(e)$ for $e \in WF_G$, with the reference to G being implied, to mean that expression e is *well-formed* in G.

1. $WF(\varepsilon)$.
2. $WF(a)$.
3. $WF(A)$ if $WF(R_G(A))$.
4. $WF(e_1e_2)$ if $WF(e_1)$ and $e_1 \rightharpoonup 0$ implies $WF(e_2)$.
5. $WF(e_1/e_2)$ if $WF(e_1)$ and $WF(e_2)$.
6. $WF(e^*)$ if $WF(e)$ and $e \not\rightharpoonup 0$.
7. $WF(!e)$ if $WF(e)$.

A grammar G is *well-formed* if all of the expressions in its expression set $E(G)$ are well-formed in G. As with the $\rightharpoonup_G$ relation, the WF_G set can be computed by iteration to a fixed point.

Lemma: Assume that grammar G is well-formed, and that all expressions in $E(G)$ handle all strings $x \in V_T^*$ of length n or less. Then the expressions in $E(G)$ also handle all strings of length $n+1$.

Proof: By induction over the step counts of $\Rightarrow_G$. The interesting cases are as follows:

- For a nonterminal A, the induction hypothesis allows us to assume that $R_G(A)$ handles all strings of length $n+1$; therefore so does A by the definition of $\Rightarrow_G$.
- For a sequence e_1e_2, we can assume that e_1 handles all strings x of length $n+1$. If $(e_1,x) \Rightarrow (n,\varepsilon)$, then $e_1 \rightharpoonup 0$, so $WF(e_2)$ applies and e_2 also handles x. If $(e_1,x) \Rightarrow (n,y)$ for $|y| > 0$, then e_2 only needs to handle strings of length n or less, which is given. If $(e_1,x) \Rightarrow (n,f)$, then e_2 is not used.
- For e^*, the $WF(e)$ condition ensures that e_1 handles inputs of length $n+1$, and the $e \not\rightharpoonup 0$ condition ensures that the recursive dependency on e^* in the success case only needs to handle strings of length n or less.

Theorem: A well-formed grammar G is complete.

Proof: By induction over the length of input strings, each expression in $E(G)$ handles every input string. Since G's start expression e_S is in $E(G)$, the conclusion follows.

3.7 Grammar Identities

A number of important identities allow PEGs to be transformed without changing the language they represent. We will use these identities in subsequent results.

Theorem: The sequence and alternation operators are associative under expression equivalence: $e_1(e_2e_3) \asymp (e_1e_2)e_3$ and $e_1/(e_2/e_3) \asymp (e_1/e_2)/e_3$.

Proof: Trivial, from the definition of $\Rightarrow_G$.

Theorem: Sequence operators can be distributed into choice operators on the left but *not* on the right: $e_1(e_2/e_3) \asymp (e_1e_2)/(e_1e_3)$, but $(e_1/e_2)e_3 \not\asymp (e_1e_3)/(e_2e_3)$.

Proof: In the left-side case, the expression $(e_1e_2)/(e_1e_3)$ invokes e_1 twice from the *same* starting point—on the same input string—making its result the same as the factored $e_1(e_2/e_3)$ expression. In the right-side case, however, suppose that e_1 succeeds but e_3 fails. In the expression $(e_1/e_2)e_3$, the failure of e_3 causes the whole expression to fail. In $(e_1e_3)/(e_2e_3)$, however, the first instance of e_3 only causes the first alternative to fail; the second alternative will then be tried, in which the e_3 might succeed if e_2 consumes a different amount of input than e_1 did.

Theorem: Predicates can be moved left within sequences distributively as follows: $e_1\ !e_2 \asymp\ !(e_1e_2)\ e_1$.

Proof: If e_1 succeeds, then e_2 is tested starting at the same point in each case, resulting in the same overall behavior; the second case merely invokes e_1 twice at the same position. If e_1 fails, then the predicate in $e_1!e_2$ is not tested at all. The predicate in $!(e_1e_2)e_1$ is tested, and succeeds because of the first e_1's failure, but the overall result is still failure due to the second instance of e_1.

Definition: Two expressions e_1 and e_2 are *disjoint* if they succeed on disjoint sets of input strings: $M_G(e_1) \cap M_G(e_2) = \emptyset$.

Theorem: A choice expression e_1/e_2 is commutative if its subexpressions are disjoint.

Proof: If either e_1 or e_2 fails on a string x, it does not matter which is tested first. The only way the language can be affected by changing their order is if e_1 and e_2 both succeed on x and consume different amounts of input. Disjointness precludes this possibility.

Although the results from Section 3.4 imply that disjointness is undecidable in general, it is easy to "force" a choice expression to be disjoint via the following simple transformation:

Theorem: $e_1/e_2 \asymp e_1/!e_1\ e_2 \asymp !e_1\ e_2/e_1$, and the latter two equivalent choice expressions are disjoint.

Proof: Trivial, by case analysis.

4 Reductions on PEGs

In this section we present methods of reducing PEGs to simpler forms that may be more useful for implementation or easier to reason about formally. First we describe how to eliminate repetition and predicate operators, then we show how PEGs can be mapped into the much more restrictive TS/TDPL and gTS/GTDPL systems.

4.1 Eliminating Repetition Operators

As in CFGs, repetition expressions can be eliminated from a PEG by converting them into recursive nonterminals. Unlike in CFGs, the substitute nonterminal in a PEG must be right-recursive.

Theorem: Any repetition expression e^* can be eliminated by replacing it with a new nonterminal A with the definition $A \leftarrow e\ A/\varepsilon$.

Proof: By induction on the length of the input string.

Theorem: For any PEG G, an equivalent repetition-free grammar G' can be created.

Proof: Simply eliminate all repetition expressions throughout G's nonterminal definitions and start expression.

4.2 Eliminating Predicates

In this section we show how to eliminate all predicate operators from any well-formed grammar whose language does not include the empty string. The restriction to grammars that do not accept the empty string is a minor but unavoidable problem: we will show later that it is impossible for a predicate-free grammar to accept the empty string without accepting *all* input strings.

Given a well-formed, repetition-free grammar $G = (V_N, V_T, R, e_S)$ where $\varepsilon \notin L(G)$, we will create an equivalent grammar $G' = (V'_N, V_T, R', e'_S)$ that is well-formed, repetition-free, and predicate-free. This process occurs in three normalization stages. In the first stage, we rewrite the grammar so that sequence and predicate expressions only contain nonterminals and choice expressions are disjoint. In the second stage, we further rewrite the grammar so that nonterminals never succeed without consuming any input. In the third stage we finally eliminate predicates.

4.2.1 Stage 1

In this stage we rewrite the existing definitions in R and the original start expression e_S, adding some new nonterminals and corresponding definitions in the process, to produce V'_N, R_1, and e_{S1}.

We first add three special nonterminals, T, Z, and F, with corresponding rules as follows. The nonterminal T matches any single terminal, and has the definition '`T <- .`' in concrete PEG syntax, before desugaring. The nonterminal Z matches and consumes any input string; to avoid introducing repetition operators, we define it $Z \leftarrow TZ/\varepsilon$. The nonterminal F always fails; to avoid using predicates we define it $F \leftarrow ZT$.

We define a function f recursively as follows, to convert expressions in our original grammar G into our first normal form:

1. $f(e) = e$ if $e \in \{\varepsilon\} \cup V_N \cup V_T$.
2. $f(e_1e_2) = AB$, adding $A \leftarrow f(e_1)$ and $B \leftarrow f(e_2)$ to R_1.
3. $f(e_1/e_2) = A\,/\,!A\ f(e_2)$, adding $A \leftarrow f(e_1)$ to R_1.
4. $f(!e) = !A$, adding $A \leftarrow f(e)$ to R_1.

Definition: The *stage 1 grammar* G_1 of G is (V'_N, V_T, R_1, e_{S1}), where $e_{S1} = f(e_S)$, $R_1 = \{A \leftarrow f(e) \mid A \leftarrow e \in R\} \cup \{$new definitions resulting from application of $f\}$, and $V'_N = V_N \cup \{$new nonterminals resulting from application of $f\}$.

Lemma: For any expression e, $f(e) \asymp_{G_1} e$.

Proof: By structural induction over e. The only interesting case is for choice expressions, which uses the identity $e_1/e_2 \asymp e_1/!e_1\ e_2$.

Theorem: $G_1 \asymp G$, all sequence and predicate expressions in the expression set of G_1 contain only nonterminals as their subexpressions, and all choice expressions are disjoint.

Proof: Direct from the construction of f.

4.2.2 Stage 2

We now rewrite the stage 1 grammar G_1 into another equivalent grammar $G_2 = (V'_N, V_T, R_2, e_{S2})$, in which all nonterminals either succeed and consume a nonempty input prefix, or fail: $\forall A \in V'_N (A \not\rightharpoonup_{G_2} 0)$. This transformation is analogous to ε-reduction on CFGs, though the details are different due to predicates.

We use two functions g_0 and g_1, to "split" expressions into ε-*only* and ε-*free* parts, respectively. The ε-*only part* $g_0(e)$ of an expression e is an expression that yields the same result as e on all input strings for which e succeeds without consuming any input, and fails otherwise. The ε-*free part* $g_1(e)$ of e likewise yields the same result as e on all inputs for which e succeeds and consumes at least one terminal, and fails otherwise.

We first define g_0 recursively as follows:

1. $g_0(\varepsilon) = \varepsilon$.
2. $g_0(a) = F$.
3. $g_0(A) = g_0(R_G(A))$.
4. $g_0(AB) = g_0(A)g_0(B)$ if $A \rightharpoonup 0$, otherwise $g_0(AB) = F$.
5. $g_0(e_1/e_2) = g_0(e_1)\,/\,g_0(e_2)$.
6. $g_0(!A) = !(A\,/\,g_0(A))$.

Lemma: The function g_0 terminates if G is well-formed.

Proof: By structural induction over the WF_G relation. Termination relies on $g_0(AB)$ not recursively invoking $g_0(B)$ if $A \not\rightharpoonup 0$.

We now define the function g_1 primitive-recursively as follows:

1. $g_1(\varepsilon) = F$.
2. $g_1(a) = a$.
3. $g_1(A) = A$.
4. $g_1(AB) = g_0(A)B \ / \ Ag_0(B) \ / \ AB$.
5. $g_1(e_1/e_2) = g_1(e_1) \, / \, g_1(e_2)$.
6. $g_1(!e) = F$.

Definition: The *stage 2 grammar* G_2 is (V'_N, V_T, R_2, e_{S2}), where $R_2 = \{A \leftarrow g_1(e) \mid A \leftarrow e \in R_1\}$, and $e_{S2} = g_1(e_{S1}) \ / \ g_0(e_{S1})$. We effectively split all of the nonterminal definitions in R_1, retaining only the ε-free parts in the definitions of R_2, while substituting the corresponding ε-only parts at the points where these nonterminals are referenced in order to preserve the original behavior. There are only two such points: case 6 of g_0, where we rewrite the operands of predicate expressions, and case 4 of g_1, for ε-free sequences.

We say that the *splitting invariant* holds if the following is true:

- If $(e,x) \Rightarrow^+_{G_1} \varepsilon$, then $(g_0(e),x) \Rightarrow^+_{G_2} \varepsilon$ and $(g_1(e),x) \Rightarrow^+_{G_2} f$.
- If $(e,x) \Rightarrow^+_{G_1} y$ for $|y| > 0$, then $(g_0(e),x) \Rightarrow^+_{G_2} f$ and $(g_1(e),x) \Rightarrow^+_{G_2} y$.
- If $(e,x) \Rightarrow^+_{G_1} f$, then $(g_0(e),x) \Rightarrow^+_{G_2} f$ and $(g_1(e),x) \Rightarrow^+_{G_2} f$.

Lemma: Assume that the splitting invariant holds for all input strings of length n or less. Then the splitting invariant holds for strings of length $n+1$.

Proof: By induction over the step counts of $\Rightarrow_{G_1}$ and $\Rightarrow_{G_2}$.

Theorem: G_2 is well-formed and equivalent to G, and for all nonterminals $A \in V_N$, $A \not\rightharpoonup_{G_2} 0$.

Proof: A direct consequence of the splitting invariant and the fact that G is well-formed.

4.2.3 Stage 3

Finally we rewrite G_2 into the final grammar $G' = (V'_N, V_T, R', e'_S)$.

Definition: We define a function d, such that $d(A,e)$ "distributes" a nonterminal A into an ε-only expression e resulting from the stage 2 function g_0:

1. $d(A,e) = e$, if $e \in \{\varepsilon, F\}$.
2. $d(A,e_1e_2) = d(A,e_1) \ d(A,e_2)$.
3. $d(A,e_1/e_2) = d(A,e_1) \ / \ d(A,e_2)$.
4. $d(A,!e) = !(A \ e)$.

Lemma: If $e = g_0(e')$ and $e' \in E(G_2)$, then $A \ e \asymp d(A,e) \ A$. That is, we can use $d(A,e)$ to move e leftward across a nonterminal reference in a sequence expression.

Proof: Structural induction on e and the identities in Section 3.7.

Now define a function $n(e,C) = (e \ (Z/\varepsilon) \ / \ \varepsilon) \ C$.

Lemma: If e is an ε-only expression in G_2, then $!e \ C \asymp n(e,C)$.

Proof: If e succeeds, then the (Z/ε) also succeeds and consumes *the entire remaining input*. (The nonterminal Z alone is not sufficient because it was rewritten in stage 2 to be ε-free.) Since any nonterminal C is ε-free, the overall expression will therefore fail. If e fails, however, then $(e \ (Z/\varepsilon) \ / \ \varepsilon)$ succeeds without consuming anything, making the overall expression behave according to C.

We now define a function h_0 to eliminate predicates from ε-producing expressions resulting from the g_0 or d functions:

1. $h_0(\varepsilon, C) = C$.
2. $h_0(F, C) = F$.
3. $h_0(e_1e_2, C) = n(n(h_0(e_1,C),C) \ / \ n(h_0(e_2,C),C),C)$.
4. $h_0(e_1/e_2, C) = h_0(e_1,C) \ / \ h_0(e_2,C)$.
5. $h_0(!(B/e), C) = n(B/h_0(e,C),C)$.
6. $h_0(!(A \ (B/e)), C) = n(A \ (B/h_0(e,C)),C)$.

Lemma: If $e = g_0(e')$ or $e = d(A, g_0(e'))$ and $e' \in E(G_2)$, then $h_0(e,C)$ is a predicate-free expression equivalent to $e \ C$.

Proof: By structural induction over e. Case 5 handles predicates resulting directly from g_0, which always have the form $!(B/e_1)$, where e_1 is likewise an expression resulting from g_0. Case 6 similarly handles the situation $e = d(A, g_0(e'))$. Case 3 rewrites a sequence e_1e_2 using the not-predicate analog of DeMorgan's Law: if e_1 and e_2 are ε-only expressions, then $e_1e_2 \asymp !(!e_1/!e_2)$. We cannot simply use $h_0(e_1e_2,C) = h_0(e_1,C) \ h_0(e_2,C)$ because $h_0(e_1,C)$ consumes input if C succeeds, which would cause the $h_0(e_2,C)$ part to start at the wrong position.

We now define a corresponding function h_1 to eliminate predicates from the ε-free expressions generated by the stage 2 function g_1:

1. $h_1(e) = e$, if $e \in \{a, A\}$.
2. $h_1(AB) = AB$.
3. $h_1(e_1B) = h_0(e_1,B)$, if e_1 is not a nonterminal.
4. $h_1(Ae_2) = h_0(d(A,e_2),A)$, if e_2 is not a nonterminal.
5. $h_1(e_1/e_2) = h_1(e_1) \ / \ h_1(e_2)$.

Lemma: If $e = g_1(e')$ and $e' \in E(G_2)$, then $h_1(e)$ is a predicate-free expression equivalent to e in G_2.

Proof: By structural induction over e. In case 3, we know from the definition of g_1 that e_1 is an ε-only expression resulting from g_0, so we use the function h_0 to combine it with the subsequent (ε-free) nonterminal and eliminate predicates from e_1. Case 4 is similar, except that we must first move e_2 to the left of A using the d function before applying the predicate transformation.

Definition: The *predicate-reduced* grammar G' of G is (V'_N, V_T, R', e'_S), where V'_N is the set of nonterminals produced in stage 1, $R' = \{A \leftarrow h_1(e) \mid A \leftarrow e \in R_2\}$, and $e'_S = h_1(g_1(e_{S1})) \ / \ h_0(g_0(e_{S1}),T)$.

Theorem: G' is well-formed, repetition-free, predicate-free, and equivalent to G.

Proof: G' is repetition-free because G is repetition-free and we never introduced any repetition operators. From the previous result, each nonterminal $A \in V'_N$ is equivalent to the corresponding nonterminal in the stage 2 grammar. By the same result, the $h_1(g_1(e_{S1}))$

part of the new start expression e'_S is equivalent to the ε-free part of the original start expression e_S. The $h_0(g_0(e_{S1}),T)$ in the new start expression succeeds and consumes exactly one terminal whenever the input string is nonempty and the ε-only part of the original start expression e_S succeeds. Finally, since we made the assumption at the start that the original grammar does not accept the empty string, the transformed grammar behaves identically for this degenerate case. Since the acceptance of a string into the language of a grammar only depends on the success or failure of the start expression, and not on how much of the input the start expression consumes, the new grammar G' accepts exactly the same strings as G.

4.2.4 The Empty String Limitation

To show that we have no hope of avoiding the restriction that the original grammar cannot accept the empty input string, we prove that any predicate-free grammar cannot accept the empty input string without accepting all input strings.

Lemma: Assume that G is a predicate-free grammar, and that for any expression e and input x of length n or less, $(e,\varepsilon) \Rightarrow^+ \varepsilon$ iff $(e,x) \Rightarrow^+ \varepsilon$. Then the same holds for input strings of length $n+1$.

Proof: By induction over step counts in $\Rightarrow_G$.

Theorem: In a repetition-free grammar G, an expression e matches the empty string iff it matches *all* input strings and produces only ε results. In consequence, $\varepsilon \in L(G)$ implies $L(G) = V_T^*$.

Proof: By induction over string length.

We could work around the empty string limitation by defining PEGs to require all recognized strings to include a designated end marker terminal, as Birman does in the original TS and gTS systems [5].

4.3 Reduction to TS/TDPL

We can reduce any predicate-free PEG to an instance of Birman's TS system [4, 5], renamed "Top-Down Parsing Language" (TDPL) by Aho and Ullman [3]. We will use the latter term for its descriptiveness. TDPL uses a set of grammar-like definitions, but these definitions have only a few fixed forms in place of open-ended hierarchical parsing expressions. We can view TDPL as the PEG analog of Chomsky Normal Form (CNF) for context-free grammars. Instead of defining TDPL "from the ground up" as Birman does, we simply define it as a restricted form of PEG.

Definition: A *TDPL grammar* is a PEG $G = (V_N, V_T, R, S)$ in which S is a nonterminal in V_N and all of the definitions in R have one of the following forms:

1. $A \leftarrow \varepsilon$.
2. $A \leftarrow a$, where $a \in V_T$.
3. $A \leftarrow f$, where $f \equiv\ !\varepsilon$.
4. $A \leftarrow BC/D$, where $B, C, D \in V_N$.

The third form, $A \leftarrow f$, representing unconditional failure, is considered "primitive" in TDPL, although we define it here in terms of the parsing expression $!\varepsilon$. The fourth form, $A \leftarrow BC/D$, combines the functions of nonterminals, sequencing, and choice. A TDPL grammar G is interpreted according to the usual $\Rightarrow_G$ relation.

Theorem: Any predicate-free PEG $G = (V_N, V_T, R, e_S)$ can be reduced to an equivalent TDPL grammar $G' = V'_N, V_T, R', S)$.

Proof: First we add a new nonterminal S with definition $S \leftarrow e_S$, representing the original start expression. We then add two nonterminals E and F with definitions $E \leftarrow \varepsilon$ and $F \leftarrow f$ respectively. Finally, we rewrite each definition that does not conform to one of the TDPL forms above using the following rules:

$A \leftarrow B$	$\longmapsto$	$A \leftarrow BE/F$
$A \leftarrow e_1 e_2$	$\longmapsto$	$A \leftarrow BC/F$ $B \leftarrow e_1$ $C \leftarrow e_2$
$A \leftarrow e_1/e_2$	$\longmapsto$	$A \leftarrow BE/C$ $B \leftarrow e_1$ $C \leftarrow e_2$
$A \leftarrow e^*$	$\longmapsto$	$A \leftarrow BA/E$ $B \leftarrow e$

Aho and Ullman define an "extended TDPL" notation [3] equivalent in expressiveness to repetition-free, predicate-free PEGs, with reduction rules almost identical to those above.

4.4 Reduction to gTS/GTDPL

Birman's "generalized TS" (gTS) system, named "generalized TDPL" (GTDPL) by Aho and Ullman, is similar to TDPL, but uses slightly different basic rule forms that effectively provide the functionality of predicates in PEGs.

Definition: A *GTDPL grammar* is a PEG $G = (V_N, V_T, R, S)$ in which S is a nonterminal and all of the definitions in R have one of the following forms:

1. $A \leftarrow \varepsilon$.
2. $A \leftarrow a$, where $a \in V_T$.
3. $A \leftarrow f$, where $f \equiv\ !\varepsilon$.
4. $A \leftarrow B[C,D]$, where $B[C,D] \equiv B\,C\,/\,!B\,D$, and $B, C, D \in V_N$.

Theorem: Any PEG $G = (V_N, V_T, R, e_S)$ can be reduced to an equivalent GTDPL grammar $G' = V'_N, V_T, R', S)$.

Proof: First we add the definitions $S \leftarrow e_S$, $E \leftarrow \varepsilon$, and $F \leftarrow f$, as above for TDPL. Then we rewrite all non-conforming definitions using the following transformations:

$A \leftarrow B$	$\longmapsto$	$A \leftarrow B[E,F]$
$A \leftarrow e_1 e_2$	$\longmapsto$	$A \leftarrow B[C,F]$ $B \leftarrow e_1$ $C \leftarrow e_2$
$A \leftarrow e_1/e_2$	$\longmapsto$	$A \leftarrow B[E,C]$ $B \leftarrow e_1$ $C \leftarrow e_2$
$A \leftarrow e^*$	$\longmapsto$	$A \leftarrow B[A,E]$ $B \leftarrow e$
$A \leftarrow\ !e$	$\longmapsto$	$A \leftarrow B[F,E]$ $B \leftarrow e$

4.4.1 Parsing PEGs

Corollary: It is possible to construct a linear-time parser for any PEG on a reasonable random-access memory machine.

Proof: Reduce the PEG to a GTDPL grammar and then use the tabular parsing technique described by Aho and Ullman [3].

In practice it is not necessary to reduce a PEG all the way to TDPL or GTDPL form, though it is typically necessary at least to eliminate repetition operators. Practical methods for constructing such linear-time parsers both manually and automatically, particularly using modern functional programming languages such as Haskell [11], are discussed in prior work [8, 7].

4.4.2 Equivalence of TDPL and GTDPL

Theorem: Any well-formed GTDPL grammar that does not accept the empty string can be reduced to an equivalent TDPL grammar.

Proof: Treating the original GTDPL grammar as a repetition-free PEG, first eliminate predicates (Section 4.2), then reduce the resulting predicate-free grammar to TDPL (Section 4.3).

5 Open Problems

This section briefly outlines some promising directions for future work on PEGs and related syntactic formalisms.

Birman defined a transformation on gTS that converts loop failures caused by grammar circularities into ordinary recognition failures [5]. By extension it is possible to convert any PEG into a complete PEG. It is probably possible to transform any PEG into an equivalent *well-formed* PEG, but this conjecture is unverified; Birman did not define a structural well-formedness property for gTS. Such a transformation on PEGs is conceivable despite the undecidability of a grammar's completeness, since the transformation works essentially by building "run-time" circularity checks into the grammar instead of trying to decide statically at "compile-time" whether any circular conditions are reachable.

Perhaps of more practical interest, we would like a useful conservative algorithm to determine if a choice expression e_1/e_2 in a grammar is definitely disjoint, and therefore commutative. Such an algorithm would enable us to extend PEG syntax with an unordered choice operator '$|$' analogous to the choice operator used in EBNF syntax for CFGs. The '$|$' operator would be semantically identical to '$/$', but would express the language designer's assertion that the alternatives are disjoint and therefore order-independent, and tools such PEG analyzers and PEG-based parser generators [7] could verify these assertions automatically.

A final open problem is the relationship and inter-convertibility of CFGs and PEGs. Birman proved that TS and gTS can simulate any deterministic pushdown automata (DPDA) [5], implying that PEGs can express any deterministic LR-class context-free language. There is informal evidence, however, that a much larger class of CFGs might be recognizable with PEGs, including many CFGs for which no conventional linear-time parsing algorithm is known [7]. It is not even proven yet that CFLs exist that cannot be recognized by a PEG, though recent work in lower bounds on the complexity of general CFG parsing [14] and matrix product [23] shows at least that general CFG parsing is inherently super-linear.

6 Related Work

This work is inspired by and heavily based on Birman's TS/TDPL and gTS/GTDPL systems [4, 5, 3]. The $\Rightarrow_G$ relation and the basic properties in Sections 3.3 and 3.4 are direct adaptations of Birman's work. The major new features of the present work are the extension to support general parsing expressions with repetition and predicate operators, the structural analysis and identity results in Sections 3.5 through 3.7, and the predicate elimination procedure in Section 4.2. While parsing expressions could conceivably be treated merely as "syntactic sugar" for GTDPL grammars, it is not clear that the predicate elimination transformation, and hence the reduction from GTDPL to TDPL, could be accomplished without the use of more general expression-like forms in the intermediate stages. For this reason it appears that PEGs represent a useful formal notation in its own right, complementary to the minimalist TDPL and GTDPL systems.

Unfortunately it appears TDPL and GTDPL have not seen much practical use, perhaps in large measure because they were originally developed and presented as formal models for certain types of top-down parsers, rather than as a useful syntactic foundation in its own right. Adams [1] used TDPL in a modular language prototyping framework, however. In addition, many practical top-down parsing libraries and toolkits, including the popular ANTLR [21] and the PARSEC combinator library for Haskell [15], provide backtracking capabilities that conform to this model in practice, if perhaps unintentionally. These existing systems generally use "naive" backtracking methods that risk exponential runtime in worst-case scenarios, but the same features can be implemented in strictly linear time using a memoizing "packrat parser" [8, 7].

The positive form of syntactic predicate (the "and-predicate") was introduced by Parr [20] for use in ANTLR [21], and later incorporated into JavaCC under the name "syntactic lookahead" [16]. The `metafront` system includes a limited, fixed-lookahead form of syntactic predicates under the terms "attractors" and "traps" [6]. The negative form of syntactic predicate (the "not-predicate") appears to be new, but its effect can be achieved in practical parsing systems such as ANTLR and JavaCC using semantic predicates [17].

Many extensions and variations of context-free grammars have been developed, such as indexed grammars [2], W-grammars [28], affix grammars [13], tree-adjoining grammars [12], minimalist grammars [24], and conjunctive grammars [18]. Most of these extensions are motivated by the requirements of expressing natural languages, and all are at least as difficult to parse as CFGs.

Since machine-oriented language translators often need to process large inputs in linear or near-linear time, and there appears to be no hope of general CFG parsing in much better than $O(n^3)$ time [14], most parsing algorithms for machine-oriented languages focus on handling subclasses of the CFGs. Classic deterministic top-down and bottom-up techniques [3] are widely used, but their limitations are frequently felt by language designers and implementors.

The syntax definition formalism SDF increases the expressiveness of CFGs with explicit disambiguation rules, and supports unified language descriptions by combining lexical and context-free syntax definitions into a "two-level" formalism [10]. The nondeterministic linear-time $NSLR(1)$ parsing algorithm [26] is powerful enough to generate "scannerless" parsers from unified syntax definitions without treating lexical analysis separately [22], but the algorithm severely restricts the form in which such CFGs can be written. Other machine-oriented syntax formalisms and tools use CFGs extended with explicit disambiguation rules to express both lexical and hierarchical syntax, supporting unified syntax definitions more cleanly while giving up strictly linear-time parsing [21, 29, 27]. These systems graft recognition-based functionality onto generative CFGs, resulting in a "hybrid" generative/recognition-based syntactic model. PEGs provide similar features in a simpler syntactic foundation by adopting the recognition paradigm from the start.

7 Conclusion

Parsing expression grammars provide a powerful, formally rigorous, and efficiently implementable foundation for expressing the syntax of machine-oriented languages that are designed to be unambiguous. Because of their implicit longest-match recognition capability coupled with explicit predicates, PEGs allow both the lexical and hierarchical syntax of a language to be described in one concise grammar. The expressiveness of PEGs also introduces new syntax design choices for future languages. Birman's GTDPL system serves as a natural "normal form" to which any PEG can easily be reduced. With minor restrictions, PEGs can be rewritten to eliminate predicates and reduced to TDPL, an even more minimalist form. In consequence, we have shown TDPL and GTDPL to be essentially equivalent in recognition power. Finally, despite their ability to express language constructs requiring unlimited lookahead and backtracking, all PEGs are parseable in linear time with a suitable tabular or memoizing algorithm.

Acknowledgments

I would like to thank my advisor Frans Kaashoek, as well as François Pottier, Robert Grimm, Terence Parr, Arnar Birgisson, and the POPL reviewers, for valuable feedback and discussion and for pointing out several errors in the original draft.

8 References

[1] Stephen Robert Adams. *Modular Grammars for Programming Language Prototyping*. PhD thesis, University of Southampton, 1991.

[2] Alfred V. Aho. Indexed grammars—an extension of context-free grammars. *Journal of the ACM*, 15(4):647–671, October 1968.

[3] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation and Compiling - Vol. I: Parsing*. Prentice Hall, Englewood Cliffs, N.J., 1972.

[4] Alexander Birman. *The TMG Recognition Schema*. PhD thesis, Princeton University, February 1970.

[5] Alexander Birman and Jeffrey D. Ullman. Parsing algorithms with backtrack. *Information and Control*, 23(1):1–34, August 1973.

[6] Claus Brabrand, Michael I. Schwartzbach, and Mads Vanggaard. The `metafront` system: Extensible parsing and transformation. In *Third Workshop on Language Descriptions, Tools and Applications*, Warsaw, Poland, April 2003.

[7] Bryan Ford. Packrat parsing: a practical linear-time algorithm with backtracking. Master's thesis, Massachusetts Institute of Technology, Sep 2002.

[8] Bryan Ford. Packrat parsing: Simple, powerful, lazy, linear time. In *Proceedings of the 2002 International Conference on Functional Programming*, Oct 2002.

[9] Dick Grune and Ceriel J.H. Jacobs. *Parsing Techniques—A Practical Guide*. Ellis Horwood, Chichester, England, 1990.

[10] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF—reference manual—. *SIGPLAN Notices*, 24(11):43–75, 1989.

[11] Simon Peyton Jones and John Hughes (editors). *Haskell 98 Report*, 1998. `http://www.haskell.org`.

[12] Aravind K. Joshi and Yves Schabes. Tree-adjoining grammars. *Handbook of Formal Languages*, 3:69–124, 1997.

[13] C.H.A. Koster. Affix grammars. In J.E.L. Peck, editor, *ALGOL 68 Implementation*, pages 95–109, Amsterdam, 1971. North-Holland Publ. Co.

[14] Lillian Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *Journal of the ACM*, 49(1):1–15, 2002.

[15] Daan Leijen. Parsec, a fast combinator parser. `http://www.cs.uu.nl/~daan`.

[16] Sun Microsystems. Java compiler compiler (JavaCC). `https://javacc.dev.java.net/`.

[17] Sun Microsystems. JavaCC: LOOKAHEAD minitutorial. `https://javacc.dev.java.net/doc/lookahead.html`.

[18] Alexander Okhotin. Conjunctive grammars. *Journal of Automata, Languages and Combinatorics*, 6(4):519–535, 2001.

[19] International Standards Organization. Syntactic metalanguage — Extended BNF, 1996. ISO/IEC 14977.

[20] Terence J. Parr and Russell W. Quong. Adding semantic and syntactic predicates to $LL(k)$—*pred-*$LL(k)$. In *Proceedings of the International Conference on Compiler Construction*, Edinburgh, Scotland, April 1994.

[21] Terence J. Parr and Russell W. Quong. ANTLR: A Predicated-$LL(k)$ parser generator. *Software Practice and Experience*, 25(7):789–810, 1995.

[22] Daniel J. Salomon and Gordon V. Cormack. Scannerless NSLR(1) parsing of programming languages. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI)*, pages 170–178, Jul 1989.

[23] Amir Shpilka. Lower bounds for matrix product. In *IEEE Symposium on Foundations of Computer Science*, pages 358–367, 2001.

[24] Edward Stabler. Derivational minimalism. *Logical Aspects of Computational Linguistics*, pages 68–95, 1997.

[25] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, June 1997.

[26] Kuo-Chung Tai. Noncanonical SLR(1) grammars. *ACM Transactions on Programming Languages and Systems*, 1(2):295–320, Oct 1979.

[27] M.G.J. van den Brand, J. Scheerder, J.J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In *Compiler Construction*, 2002.

[28] A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens, and R.G. Fisker. Report on the algorithmic language ALGOL 68. *Numer. Math.*, 14:79–218, 1969.

[29] Eelco Visser. A family of syntax definition formalisms. Technical Report P9706, Programming Research Group, University of Amsterdam, 1997.

[30] Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic descriptions. *Communications of the ACM*, 20(11):822–823, November 1977.