Pika parsing: parsing in reverse solves the left recursion and error recovery problems

LUKE A. D. HUTCHISON, Massachusetts Institute of Technology (alum)

A *recursive descent parser* is built from a set of mutually-recursive functions, where each function directly implements one of the nonterminals of a grammar, causing the structure of recursive calls to directly parallel the structure of the grammar. Recursive descent parsers can take time exponential in the length of the input and the depth of the parse tree, however a *memoized recursive descent parser* or *packrat parser* is able to parse in time linear in the length of the input and the depth of the parse tree. Recursive descent parsers are extremely simple to write, but suffer from two significant problems: (i) left-recursive grammars cause the parser to get stuck in infinite recursion, and (ii) it is difficult or impossible to optimally recover the parse state and continue parsing after a syntax error. Surprisingly, both problems can be solved *by parsing the input in reverse*. The *pika parser* is a new type of packrat parser that employs dynamic programming to parse the input from right to left, bottom-up – the reverse of the standard recursive descent order of top-down, left to right. This reversed parsing order enables pika parsers to directly handle left-recursive grammars, simplifying grammar writing, and enables pika parsers to directly and optimally recover from syntax errors, which is a crucial property for IDEs and compilers. Pika parsing maintains the linear-time performance characteristics of packrat parsing. Several new insights into precedence, associativity, and left recursion are presented.

CCS Concepts: • **Theory of computation → Grammars and context-free languages**.

Additional Key Words and Phrases: parsing, packrat parsing, recursive descent parsing, top-down parsing, bottom-up parsing, PEG parsing, PEG grammars, precedence, associativity, left-recursive grammars, parsing error recovery, memoization, dynamic programming

ACM Reference Format:
Luke A. D. Hutchison. 2020. Pika parsing: parsing in reverse solves the left recursion and error recovery problems. *ACM Trans. Program. Lang. Syst.* 0, 0, Article 0 (August 2020), 26 pages. https://doi.org/00.0000/0000000.0000000

1 INTRODUCTION

1.1 Parsing generative grammars

Since the earliest beginnings of computer science, researchers have sought to identify robust algorithms for parsing formal languages [1, 2]. Careful work to establish the theory of computation [26] provided further understanding into the expressive power of different classes of languages, the relationships between different classes of languages, and the relative computational power of algorithms that can parse different classes of languages.

Generative systems of grammars, particularly Chomsky's context-free grammars (CFGs) [4] and Kleene's regular expressions [17] became the workhorses of parsing for many data formats, and have been used nearly ubiquitously over several decades. However, inherent ambiguities and nondeterminism in allowable grammars increase the implementation

Author's address: Luke A. D. Hutchison, luke.hutch@alum.mit.edu, Massachusetts Institute of Technology (alum).

Manuscript submitted to ACM

complexity or decrease the runtime efficiency of parsers for these classes of language [21, 27], due to parsing conflicts and/or exponential blowup in parsing time or space due to nondeterminism [17, 22].

1.2 Bottom-up parsers

Much early work in language recognition focused on *bottom-up parsing*, in which, as the input is processed in order, a forest of parse trees is built upwards from terminals towards the root. These parse trees are joined together into successively larger trees as higher-level *productions* or *reductions* are applied. This parsing pattern is termed *shift-reduce* [1]. Examples of shift-reduce parsers include early work on precedence parsers and operator precedence parsers [3, 18, 25], the later development of LR [19] and LALR (lookahead) [10] parsers, and the more complex GLR parser [21] designed to handle ambiguous and nondeterministic grammars. Some parsers in this class run into issues with shift-reduce conflicts and/or reduce-reduce conflicts caused by ambiguous grammars; the GLR parser requires careful and complex data structure maintenance to avoid exponential blowup in memory usage, and to keep parsing time complexity to $O(n^3)$ in the length of the input. Optimal error recovery in shift-reduce parsers can be complex and difficult to implement.

A particularly interesting bottom-up parser is the CYK algorithm [6, 7, 16] for context-free grammars. This parsing algorithm employs dynamic programming to efficiently recursively subdivide the input string into all substrings, and determines whether each substring can be generated from each grammar rule. This results in $O(n^3|G|)$ performance in the length of the input n and the size of the grammar $|G|$. The algorithm is robust and simple, and has optimal error recovery capabilities since all rules are matched against all substrings of the input. However, the CYK algorithm is inefficient for very large input length n due to scaling as the cube of the input length, and additionally, the algorithm requires the grammar to be reduced to Chomsky Normal Form (CNF) [5] as a preprocessing step, which can result in a large or even an exponential blowup in grammar size in the worst case – so in practice, $|G|$ may also be large.

1.3 Top-down parsers

More recently, *recursive descent parsing* or *top-down parsing* has increased in popularity, due to its simplicity of implementation, and due to the helpful property that the structure of mutually-recursive functions of the parser directly parallels the structure of the corresponding grammar. However, recursive descent parsers suffer from three significant problems:

(1) In the simplest implementation, in the worst case the time complexity of a recursive descent parser is exponential in the length of the input, due to unlimited lookahead, and in the complexity of the grammar, due to the lack of termination of recursion at already-visited nodes during depth-first search over the Directed Acyclic Graph (DAG) of the grammar.
(2) Without special handling [23], left-recursive grammars cause naïve recursive descent parsers to get stuck in infinite recursion, resulting in stack overflow.
(3) If there is a syntax error partway through a parse, it is difficult or impossible to optimally recover the parsing state to allow the parser to continue parsing beyond the error. This makes recursive descent parsing particularly problematic for use in Integrated Development Environments (IDEs), where syntax highlighting and code assist need to continue to work after syntax errors are encountered during parsing, and also in compilers, which to be helpful need to be able to show more than one error message per compilation unit.

These problems have been gradually addressed over the years:

(1) The discovery of *packrat parsing*, employing memoization to avoid duplication of work [24], reduced the time complexity for recursive descent parsing from potentially exponential in the length of the input to linear in the length of the input and the depth of the parse tree.
(2) It was found that direct or indirect left recursive rules could be rewritten into right recursive form (with only a weak equivalence to the left recursive form) [12], although the rewriting rules can be complex, especially when semantic actions are involved, and the resulting parse tree can diverge quite significantly from the structure of the original grammar, making it more difficult to create an abstract syntax tree from the parse tree. With some extensions, packrat parsing can be made to handle left recursion directly [12, 14, 15, 28], although usually with loss of linear-time performance, which is one of the primary reasons packrat parsing is chosen over other parsing algorithms [28].
(3) Progress has been made in error recovery for recursive descent parsers, by identifying synchronization points [9], or by employing heuristic algorithms to attempt to recover after an error, yielding an "acceptable recovery rate" from 41% to 76% [8] in recent work. Up to and including this recent state-of-art, optimal error recovery has remained elusive.

1.4 The pika parser

The *pika parser*[1] is a new type of packrat parser that applies PEG operators in reverse, from right to left (i.e. from the end of the input back to the beginning) and bottom-up (i.e. from terminals towards the root of the parse tree) – see Section 2. This is the opposite of the top-down, left-to-right order of recursive descent parsing.

Pika parsers are able to solve each of the predominant problems of other parsers:

(1) As a direct inversion of top-down packrat parsing, pika parsing has the same big-Oh time complexity as packrat parsing, meaning pika parsing scales linearly as a function of the size of the input and the depth of the parse tree.
(2) Right-to-left, bottom-up parsing directly handles left-recursive grammars, and grammars containing arbitrary numbers of cycles, without modification. This can greatly simplify grammar design.
(3) Right-to-left, bottom-up parsing conveys near-perfect error recovery capabilities on the parser, because when parsing from right to left, or from the end of a document towards the beginning, if an error is encountered, the span of the input string to the right of the syntax error has already been completely parsed before the syntax error was encountered. It is straightforward to recover the parse tree at any desired level of the grammar after a syntax error, by skipping ahead through the memo table to the next valid match of any grammar rule of interest (this can be done with a single lookup in logarithmic time). This is particularly useful for IDEs, allowing syntax highlighting and code assist features to continue to work after syntax errors are encountered, and for compilers, allowing all syntax errors for a source code file to be displayed in a single pass.

The pika parser is defined to work with PEG grammars (Section 1.5), which are unambiguous and deterministic, eliminating the ambiguity and nondeterminism that complicate and slow down other bottom-up parsers.

1.5 PEG grammars

A significant refinement of recursive descent parsing known as Parsing Expression Grammars (PEGs, or more commonly, but redundantly, "PEG grammars") were proposed by Ford in his 2002 PhD thesis [12]. In contrast to most bottom-up

[1] A pika ("PIE-kah") is a small, mountain-dwelling, rabbit-like mammal found in Asia and North America, typically inhabiting moraine fields near the treeline. Like a packrat, it stores food in caches or "haystacks" during the warmer months to survive the winter.

parsers, which represent grammar rules as *generative productions*, PEG parsers were defined to operate top down, and PEG grammar rules represent *greedily-recognized patterns*. PEG grammar rules replace ambiguity with deterministic choice, i.e. all PEG grammars are unambiguous. There are only a handful of PEG rule types, and they can be implemented in a straightforward way. As a subset of all possible recursive rule types, all non-left-recursive PEG grammars can be parsed directly by a recursive descent parser, and can be parsed efficiently with a packrat parser. PEG grammars are more powerful than regular expressions, but it is conjectured (though as yet unproved) that there exist context-free languages that cannot be recognized by a PEG parser [13]). Top-down PEG parsers suffer from the same limitations as other packrat parsers: they cannot parse left-recursive grammars without special handling, and they have poor error recovery properties.

A PEG grammar consists of a finite set of *rules*. Each rule has the form $A \leftarrow e$, where A is a unique *rule name*, and e is a *parsing expression*, also referred to as a *clause*. One rule is designated as the *starting rule*, which becomes the root for top-down recursive parsing. PEG clauses are constructed from *subclauses*, *terminals*, and *rule references*. The subclauses within a rule's clause may be composed into a tree, with *PEG operators* forming the non-leaf nodes, and terminals or rule references forming the leaf nodes.

Terminals can match individual characters, strings, or (in a more complex parser) even regular expressions. One special type of terminal is the *empty string* ϵ (with ASCII notation `()`), which always matches at any position, even beyond the end of the input string, and consumes zero characters.

The PEG operators (Table 1) are defined as follows:

- A `Seq` clause matches the input at a given start position if all of its subclauses match the input in order, with the first subclause match starting at the initial start position, and each subsequent subclause match starting immediately after the previous subclause match. Matching stops if a single subclause fails to match the input at its start position.
- A `First` clause (*ordered choice*) matches if any of its subclauses match, iterating through subclauses in left to right order, with all match attempts starting at the current parsing position. After the first match is found, no other subclauses are matched, and the `First` clause matches. If no subclauses match, the `First` clause does not match. The `First` operator gives a top-down PEG parser *limited backtracking capability*.
- A `OneOrMore` clause matches if its subclause matches at least once, greedily consuming as many matches as possible until the subclause no longer matches.
- A `FollowedBy` clause matches if its subclause matches at the current parsing position, but no characters are consumed even if the subclause does match. This operator provides *lookahead*.
- A `NotFollowedBy` clause matches if its subclause does not match at the current parsing position, but it does not consume any characters if the subclause does not match. This operator provides *lookahead, logically negated*.

Two additional PEG operators may be offered for convenience, and can be defined in terms of the basic PEG operators:

- An `Optional` clause matches if its subclause matches. If the subclause does not match, the `Optional` clause still matches (i.e. this operator always matches), but consumes zero characters. The clause $e?$ can be automatically transformed into e/ϵ, reducing the number of PEG operators that need to be implemented directly.
- A `ZeroOrMore` clause matches if its subclause matches zero or more times, greedily consuming as many matches as possible. If its subclause does not match (in other words if the subclause matches zero times), then the `ZeroOrMore` still matches the input, consuming zero characters (i.e. this operator always matches). The clause $e*$

Name	Num subclauses	Notation	Equivalent to	Example rule in ASCII notation
`Seq`	2+	$e_1\ e_2\ e_3$		`Sum <- Prod '+' Prod;`
`First`	2+	$e_1 \,/\, e_2 \,/\, e_3$		`ArithOp <- '+' / '-';`
`OneOrMore`	1	$e+$		`Whitespace <- [ \t\n\r]+;`
`FollowedBy`	1	$\&e$		`Exclamation <- Word &'!';`
`NotFollowedBy`	1	$!e$		`VarName <- !ReservedWord [a-z]+;`
`Optional`	1	$e?$	e/ϵ	`Value <- Name ('[' Num ']')?;`
`ZeroOrMore`	1	$e*$	$(e+)?$ **[or]** $e+/\epsilon$	`Program <- Statement*;`
`Longest`	2+	$e_1 \mid e_2 \mid e_3$		`Expr <- E1 | E2 | E3;`

Table 1. PEG operators, defined in terms of subclauses e and e_i, and the empty string ϵ.

can be automatically transformed into $(e+)?$, which can be further transformed into $e+/\epsilon$, again reducing the number of operators that need to be implemented.

One new PEG operator is proposed here as a convenience feature when writing grammars:

- A `Longest` clause matches if any of its subclauses match at the current parsing position. If multiple subclauses match, the subclause that consumes the longest span of input is used as the match, tiebreaking in left-to right order. Without this new `Longest` operator, it can be difficult to get a standard PEG grammar to match the input as intended, particularly because the greedy nature of `First` can cause parsing failures if its subclauses were not listed in the correct order by the grammar-writer. An earlier subclause of a `First` clause may inadvertently match a prefix of the input that was intended to be matched by a later subclause, because the earlier clause matches prefixes of strings matched by the later clause. To resolve this, the grammar writer must understand the partial ordering induced by all possible prefix matches between all subclauses of the `First` clause *across all valid inputs*, and then this partial ordering must be serialized into a total ordering using a topological sort, so that subclauses that match longer sequences are listed before subclauses that match prefixes of those longer sequences. If it is not easy or possible to determine the partial ordering of prefix matches of all subclauses across all inputs, then in many cases, a `Longest` operator may be able to be used as a shortcut to obtain the desired parsing behavior. `Longest` will often match the grammar writer's intent and intuition better than `First`, since the PEG philosophy is one of greedily matching as many characters in the input as possible with a grammar rule. However note that `Longest` will always try to match all subclauses, whereas `First` only tries to match as many of its subclauses as needed before a match is found – so in general, `First` will be more efficient than `Longest`.

1.6 Applying PEG rules bottom-up

PEG operators were originally defined for use in recursive descent parsers, so can be viewed as recurrence relations. Many recurrences can be solved bottom-up using dynamic programming (DP), often with dramatic savings in computational work compared to computing the same recurrences top-down. The type of memoization employed by packrat parsers is related to but not the same as dynamic programming.

It is difficult to modify top-down parsers so that they can handle left recursion, and so that they have robust error recovery characteristics. However, both of these problems are solved, or at least simplified, in almost any type of bottom-up parser. Therefore, it is worth considering how a PEG parser could be built to work bottom-up using dynamic programming.

1.7 Inversion of recurrence relations in dynamic programming algorithms

In DP algorithms, there is always an underlying recurrence relation, and this recurrence relation is evaluated bottom-up, from the base cases of recursion up to the recursion root, with the purpose of duplicating as little work as possible in computing the value of the topmost invocation of the recurrence relation. This is implemented using lookups of earlier-computed entries in a *DP table* (which is an indexed memo table), and then choosing the correct order in which to populate the DP table entries so that values are always computed before they are needed by higher invocations of the recurrence relation.

There are three different methods for implementing dynamic programming given a recurrence relation. For example, the Fibonacci sequence, defined by the recurrence $F_i = F_{i-1} + F_{i-2}$, with base cases $F_0 = 0$ and $F_1 = 1$, can be evaluated using any of the four following methods:

(1) *Inefficiently, top-down, using unmemoized recursion.* Recursion terminates at the base cases F_0 and F_1. Top-down unmemoized recursion duplicates work exponentially as a function of i, since the call graph for top-down direct evaluation of the recurrence relation is a Directed Acyclic Graph (DAG), and depth-first traversal of a DAG without terminating recursion at previously-visited nodes can result in running time exponential in the height of the DAG.

(2) *Efficiently, top-down, using memoized recursion.* This avoids duplicating work by using a *memo table* that contains previously-calculated values so they can be reused. When recursing down from F_i to F_0, if a value F_j for $(1 < j < i)$ has previously been computed on a different branch of recursion, then that value is used as the return value for the current branch of recursion, pruning any further recursion. Recursion initially continues downward on the left branch until F_1 and F_0 are reached, producing the value for F_2. Subsequently, each frame of recursion memoizes its return value and returns from its left branch to the parent frame of recursion. The parent checks the memo table for the right branch of recursion, and finds the value has already been computed and memoized, so the memoized value is used instead of recursing further, etc. This is analogous to the memoization method of top-down packrat parsing.

(3) *Efficiently, bottom-up, using standard dynamic programming.* This is the standard method used to compute the Fibonacci sequence, even though it is not normally thought of as an example of dynamic programming. This algorithm starts by storing the two base cases in a two-element memo table, $m_0 = F_0$ and $m_1 = F_1$. Then the algorithm iterates from F_2 to F_i, adding the two previous memo table entries at each step, and then dropping the oldest value m_0, moving m_1 to m_0, and moving F_j to m_1.

(4) *Efficiently, bottom-up, using the "recurrence inversion" method of dynamic programming.* This algorithm starts by seeding the dynamic programming table (the memo table) with the base cases (also termed the terminals or leaf nodes) F_0 and F_1. Both these recurrence terms are added to an initially-empty "active set" (also known as the "dirty set" or "DP wavefront"). Defining F_i as the "parent clause" of the recurrence, and each of F_{i-1} and F_{i-2} as "subclauses" of the recurrence, each new subclause calculation that is added to the active set triggers the evaluation of its corresponding parent clause (this is a reversal of the evaluation order used in recursion, where the parent clause recurses down to its subclauses). For the Fibonacci example, each F_{i-1} that is computed should trigger the evaluation of F_i, since F_{i-1} is a subclause of F_i, and each F_{i-2} in the active set should also trigger the evaluation of F_i. These two "seeding" actions (the inverse of recursion) can be collected together: for a subclause $F_{i'}$ the function mapping the parent clause index to the subclause index can be *inverted* to find the parent clause index in terms of the subclause index. When a subclause $F_{i'}$ is computed and added to the

active set, it should trigger the evaluation of both the parent clauses $F_{i'+1}$ and $F_{i'+2}$, since $F_{i'}$ is a subclause of the recurrence $F_{i'+1} = F_{i'} + F_{i'-1}$ and also of the recurrence $F_{i'+2} = F_{i'+1} + F_{i'}$. In other words, triggering parent clauses when subclauses are evaluated requires not only reversing the arrow direction for the edges in the call graph between a parent clause and its subclauses, but also *inverting the indexing function* that maps parent indices to subclause indices, producing a mapping from subclause index to parent index. For the Fibonacci example, the parent-to-subclause relationship $F_i \rightarrow F_{i-1}$ is inverted to $F_{i'+1} \leftarrow F_{i'}$, and the parent-to-subclause relationship $F_i \rightarrow F_{i-2}$ is inverted to $F_{i'+2} \leftarrow F_{i'}$. These are collected to form the subclause-to-parent inverted relationship $\{F_{i'+1}, F_{i'+2}\} \leftarrow F_{i'}$. Care must be taken to ensure that DP table values are computed before they are needed by any recurrence evaluation. Once a parent clause $F_{i'}$ has been computed, its value is memoized, and its parent clauses are triggered for evaluation. This DP wavefront expands upwards until no more parent clauses are triggered and the active set is empty.

Almost all DP algorithms taught in an algorithms classes use method (3), which requires the writing out of the recurrence relation and the selection of an evaluation order that will populate the DP table in the correct order so that values are always available before they are needed (e.g. any of row-major, column-major or trailing-diagonal order can be used for populating the 2D DP table for the string edit distance algorithm [20]). Despite the recurrence table being populated bottom-up, the recurrence is always applied top-down for a single step for each new table entry, which does not require inverting the recurrence.

Method (4) is not typically taught in universities, even in advanced algorithms classes. Many problems in international programming competitions can only be solved efficiently using dynamic programming, but after writing out the recurrence, figuring out the correct evaluation order using method (3) can be exceptionally difficult. The simplest efficient solution to many of these problems requires method (4), the inversion of the recurrence relation, and the use of an expanding and possibly irregular DP wavefront or active set to gradually populate the DP table. Experienced programming competition participants will be familiar with this technique for DP recurrence inversion and wavefront propagation, and the technique is typically passed on by word of mouth or by studying previous winning solutions.

With some specialization to the parsing problem, the method used to implement the pika parser is a hybrid of method (3) and method (4) (Section 2).

1.8 The graph-theoretic relationship between recursive descent parsing and packrat parsing

Parsing a specific input with a recursive descent parser for a specific grammar results in a *call graph* of recursive calls. Without memoization, in the general case this call graph is a DAG, resulting in unmemoized recursive descent parsers taking as much as exponential time in the length of the input and/or the depth of the parse tree. The use of memoization in packrat parsing effectively breaks edges in this DAG, traversing only a *minimum spanning tree* of the call graph, by terminating recursion at the second and subsequent incoming edge to any node in the call graph. This can result in up to an exponential speedup of graph traversal.

The call graph of a recursive descent parsing algorithm on a specific input can be inverted from top-down to bottom-up by reversing the arrow direction on the DAG edges, and by ensuring that the start position of the match of a parent clause can be accurately predicted from the start position of the match of a subclause (which is precisely the index inversion referred to in Section 1.7, method (4)).

With this background, it is now possible to describe the pika parsing algorithm.

2 THE PIKA PARSER: RIGHT-TO-LEFT, BOTTOM-UP PACKRAT PARSING

2.1 Overview of pika parsing

2.1.1 Premise. The premise of pika parsing is very simple: take the structure of the recursive call graph of a top-down parse of a given input with a given grammar, and invert this downwards-directed DAG so that parsing happens bottom-up from terminals towards the root. Instead of shallower frames of recursion recursing down into deeper frames of recursion, in bottom-up parsing, deeper frames of recursion trigger the execution of shallower frames of recursion using back-references from subclauses to their parent clauses. Running a recursive algorithm "backwards" from leaves to root may seem strange; however, combined with memoization, this is exactly the definition of dynamic programming.

2.1.2 Initial incorrect DP table population order: bottom-up, in parallel across the input. With bottom-up parsing formulated as a dynamic programming problem, it may seem possible to parallelize the parser across the entire input string (or, more accurately, across the DP wavefront). The parser would work on extending matches upwards in parallel across the input, building the parse tree from terminal matches up towards the root. In fact, an early version of the pika parser worked this way, and was able to correctly parse many grammars and inputs in parallel – seemingly a first for any parser. However, when dealing with precedence grammars, particularly when the grammar contained left-recursive rules, the parser was not always able to parse the input. As an example, consider the input string `1+2+3*4*5`. A parallel parser that uses precedence climbing to evaluate a left-recursive grammar (Section 3) would produce the two parse tree fragments `(((1)+2)+3)` and `(((3)*4)*5)` in parallel, but then could not join these two fragments together into a single parse tree with the structure `(((1)+2)+(((3)*4)*5))`, because the two fragments overlap at the input character `3`. The memo entry for `3` was consumed as a subclause match of the first fragment before precedence climbing replaced the match in that memo entry with the higher-precedence match `(((3)*4)*5)`. This failure of parallelized bottom-up parsing is caused by trying to evaluate PEG clauses before all subclauses of those PEG clauses are fully matched and memoized. This is an example of a race condition, whether or not the parser is actually parallelized (i.e. even in a single-threaded implementation, the parser can fail if the memo table entries are not updated in the right order). All DP algorithms fail if the DP table is not populated in the correct order.

2.1.3 Correct DP table population order: right to left, bottom to top. Whether or not a PEG operator matches at a given start position in the input depends *only* upon (i) the match results for lower-level (higher-precedence) subclauses at the same start position; or, if the PEG clause is a `Seq` or `OneOrMore` operator, (ii) the match result for the second or subsequent subclause match at or to the right of the current start position. In particular, by definition, *PEG operators never match characters (or depend upon memo table entries) to the left of the current parsing position.* The correct DP table population order must ensure that all subclause matches are already in the memo table before a parent clause is matched, *and that any already-memoized match is the highest-level match that it is possible to find for that subclause and start position.* This is required to replicate top-down parsing semantics, since top-down parsing always starts at the highest-possible match for a subclause, and recurses down from there. The only way to guarantee that the memo table already contains the highest possible match for a memo table entry lower in the table or to the right of the current table entry is to populate the table *from right to left (i.e. from the end of the input towards the beginning), and from the lowest-level/highest-precedence clauses towards highest-level/lowest-precedence clauses (i.e. from terminals up towards the starting expression in the grammar).* Simply parsing bottom-up is insufficient to replicate top-down parsing semantics for a memoized parser; it is also necessary to parse from right to left.

This additional requirement of parsing right to left is highly surprising and unusual in the domain of parsing algorithms, but this is the only correct population order for the DP table. There is a symmetry here: whereas a packrat parser operates top to bottom and left to right, a pika parser operates in the direct mirror image order – right to left and bottom to top. Without fully inverting the parsing order in this manner, the semantics of top-down parsing cannot be replicated by a bottom-up parser, leading to an incomplete parse tree.

2.1.4 Inversion of DP indexing function. Complex DP algorithms are best implemented using the recurrence inversion method (Section 1.7, method (4)). Inverting the recurrence relations represented by a set of PEG grammar rules would appear to require the use of *dynamic back-references* from subclauses to their parent clauses, where each time a subclause's memo entry is consulted by a parent clause, the subclause memo entry stores a back-reference to the parent clause, so that if a better match is ever found for the subclause memo entry, the parent clause can be triggered for re-evaluation at the correct start position. For example, if a parent `Seq` clause at input position i consumes m characters via the match of its first subclause, then its second subclause will need to be matched in the memo table starting at input position $(i + m)$, which is a dynamic function of the number of characters consumed by the previous subclause match.

However, under the assumption that everything to the right of the current parse position has already been completely parsed and memoized, given the chosen DP evaluation order, no changes will ever be made to the memo table to the right of the current parse position, therefore these dynamic back-references do not need to be written to subclause memo table entries. In the inverted view, there is never any need to trigger parent matches to the left of the current match position when a subclause matches at the current position, because parent clauses to the left of the current position will be evaluated in due course as the parser moves from right to left.

In fact, when a subclause matches at start position i in the input, the only parent clauses that need to be scheduled for matching are those parent clauses that share the same start position i as the subclause; a dynamic offset mapping the subclause start position to the parent start position is never needed. The recurrence indexing function that maps the parent start position to the subclause start position is $i \rightarrow i$, which is the identity function, therefore inversion has no effect. This makes the expansion of the DP wavefront very simple.

A PEG rule is of course still free to look up memo table entries for the second or subsequent subclause match of a `Seq` or `OneOrMore` clause by looking to the right of or below the current position in the memo table, since all these memo table entries will already represent fully-parsed suffixes of the input and fully-generated subtrees of the final parse tree.

2.1.5 Proof of termination of left recursion in a pika parser. One issue that needs to be addressed is whether left recursion will terminate when executed bottom-up, since left recursion does not terminate when executed top-down.

Define a match (Section 2.2.4) to consist of a memo key (i.e. the position of the entry in the memo table – Section 2.2.1), the length of the match (i.e. the number of characters consumed by the match), and an array of subclause matches (i.e. the child nodes of this match node in the parse tree). Match M_1 is said to be a *better match* than match M_2 when the two matches share the same memo key (meaning they are matches for the same clause at the same start position), and when:

- (for `First` clause matches) the index of the first matching subclause of M_1 is less than the index of the first matching subclause of M_2, as required by the semantics of `First`; or
- the length of M_1 is greater than the length of M_2. (In any path upwards through the final parse tree from a terminal or leaf node to the root, the length of the match corresponding to each successively higher node should increase monotonically, since top-down PEG parsing is defined such that the length of a given clause match is

the sum of the length of all its subclause matches – therefore a parent clause cannot match fewer characters than any of its subclauses.)

A memo entry is only updated with a new match if there is no previous match for that memo entry, or if there does exist a previous match for the entry, but the new match is a better match than the old match, as defined above. Notably, a memo entry containing a previous match is not updated with a new match if the two matches have the same length (and if they have the same first matching subclause index, if the matches are for a `First` clause). The parent clauses of a memo entry's clause are only triggered for evaluation (expanding the DP wavefront upwards) when the entry is updated with a better match.

For a memo entry's match to be updated, requiring that the length of any new (non-`First`) match be strictly greater than the length of any old match for the same memo entry accomplishes two things:

- *Left recursion is guaranteed to terminate at some point*, because the input string is of finite length, and each loop around a cycle in the grammar must increase the length of match by at least one character.
- *Left recursion is able to terminate early (before consuming the entire span of the input string to the right of the match position).* Specifically, left recursion terminates as soon as no more input can be consumed by the left-recursive rule. This occurs when the DP wavefront is expanded upwards through a left-recursive cycle, causing the same clause in the grammar to be matched twice at the same start position, and both matches have the same length. Since the length of the newer match is the same as the length of the older match, the newer match is not found to be better than the older match. Therefore, the newer match is not written to the memo entry, and no parent clauses are triggered for subsequent re-evaluation. This terminates the upwards expansion of the DP wavefront for that grammar rule.

As an aside, and more generally, lower-level matches in the parse tree match monotonically fewer characters than higher-level matches on the same path in the parse tree (since the length of a match is the sum of the lengths of its subclause matches). Lower-level matches in the parse tree also represent higher precedence than higher-level matches on the same path. Given the following definition:

Definition 1. *The* absolute precedence P_i^m *of a specific match* $m = \langle \text{clause}, \text{startPos} \rangle$ *of a rule* R_i <- clause *is an integer chosen such that for two matches* $P_a^s < P_b^t$*, whenever* s *is an ancestor of* t *in the parse tree, regardless of the number of loops around a cycle in the grammar on the path between the two parse tree nodes.*

the following corollary becomes evident:

Corollary 1. *Given two matches from the same precedence hierarchy, where the matches have the same starting position and different match lengths, the shorter match must be of higher absolute precedence than the longer match.*

Restated, the match length of two different rules in the same precedence hierarchy, where the matches have the same start position, are monotonically reverse-ordered with respect to the absolute precedence level of the matches, and are monotonically ordered with respect to depth in the parse tree. Longer matches correspond to lower absolute precedence, and are higher in the parse tree; shorter matches correspond to higher absolute precedence, and are lower in the parse tree. This is relevant to the newly-proposed `Longest` PEG operator (Section 1.5).

2.1.6 Proof of optimality of error recovery characteristics in a pika parser. PEG clause matches only depend upon the span of input from their start position to their end position, and do not depend upon any characters before their start position or after their end position. Because the input is parsed from right to left, meaning the memo table is always

fully populated to the right of the current parse position, the span of input to the right of a syntax error is always parsed optimally, because the syntax error was not even encountered yet during right-to-left parsing to the right of the syntax error. To the left of the syntax error, parsing recovers for a given clause in the grammar as soon as a complete match of the clause can be found where the match ends before the beginning of the syntax error.

Restated, parsing of input to the right of a syntax error is optimal for bottom-up parsing, and parsing of input to the left of a syntax error has the same characteristics for bottom-up parsing as it does for top-down parsing: rules whose matches end before the syntax error starts will match the input, whereas rules whose matches would overlap with the location of the syntax error in a correct input will fail to match.

Therefore, error recovery to the left or right of any syntax error, or between any closest pair of syntax errors, is optimal for pika parsing.

2.2 Implementation details

The key classes of the pika parser are as follows. The code shown is simplified to illustrate only the most important aspects of the parsing algorithm and data structures (e.g. fields are not always shown as initialized, trivial assignment constructors are omitted, optimizations are not shown, etc.). Listings are given in Java notation for precise semantic clarity compared to an invented pseudocode. See the reference parser for the full implementation (Section 8).

2.2.1 The `MemoKey` *class (Listing 1).* This class has two fields, `clause` and `startPos`, corresponding to the row and column of the memo table respectively (there is one memo table row per clause or subclause in the grammar, and one memo table column per character in the input string to be parsed).

The *natural ordering*[2] (or the default sort order) is defined for `MemoKey` to sort instances into decreasing order of `startPos`, and then into bottom-up topological order of `clause`, based on a topological sort of the DAG structure of clauses in the grammar performed during initialization. The topological sort produces a clause index, `Clause.clauseIdx`, representing the position of the clause in the topological order of all clauses in the grammar, ordered from terminals up to the toplevel clause.

Instances of `MemoKey` are added to a priority queue to implement both the DP wavefront and the DP table population order (Section 2.2.2). The priority queue uses this natural ordering to define the order of removal of elements from the queue. The elements in the priority queue are the wavefront or active set of the DP algorithm, and the head element in the priority queue is the index of the next element of the DP table that should be matched and memoized.

2.2.2 The `Grammar` *class (Listing 2).* This class has one primary method, `parse`, which parses the input string. The `Grammar` class has a field that contains a collection of `Rule` objects for the rules of the grammar, and also contains the field `allClauses`, which is a list of all unique clauses and sub-clauses across all rules, in topological sort order from a lowest clause (a terminal) to the highest clause (the start rule for top-down parsing).

The `parse` method creates the memo table and the priority queue for ordering the expansion of the DP wavefront, then initializes the memo table, populating the table with all positions at which each terminal matches the input. The empty string match, ϵ (written `"()"` in ASCII notation, and implemented using the `Nothing` class in the reference parser), is not memoized in order to keep the memo table small, since this clause matches everywhere (Section 4.1). The memo table initialization process can be optimized using an optional lex preprocessing step, which reduces the initial size of the memo table by minimizing the number of spurious matches (Section 4.3).

[2]The Java mechanism for defining the natural ordering requires implementing the `Comparable` interface, which requires `this.compareTo(other)` to return a negative number if (`this < other`), zero if (`this == other`), and a positive number if (`this > other`).

During initialization, as each matching terminal is found by matching the terminal clause top-down (i.e. unmemoized), `memoTable.addMatch(match, priorityQueue)` is called to add the match to the memo entry designated by `memoKey` (also stored at `match.memoKey`). This `addMatch` method call also adds the seed parent clauses of each terminal match to the priority queue, paired with the current start position (Sections 2.2.3, 2.2.5).

After initialization, the `parse` method enters the main parsing loop. This loop successively removes the head of the priority queue until the priority queue is empty. For each `MemoKey` removed from the priority queue, an attempt is made to match `memoKey.clause` at position `memoKey.startPos`. If the clause matches, the match is added to the memo table. Again this may result in new higher-level entries being added to the priority queue for seed parent clauses of the matching clause.

Somewhat surprisingly, the `parse` method as shown is the entire pika parsing algorithm. However, `Clause.match` (Section 2.2.3) and `MemoTable.addMatch` (Section 2.2.6) still need to be detailed.

2.2.3 The `Clause` class (Listing 3). This class represents a PEG operator or clause, and any nested PEG operators or subclauses. This class has a subclass `Terminal`, which is the superclass of all terminal clause implementations (nonterminals simply extend `Clause` directly). The `Clause` subclass for each PEG operator type provides a concrete implementation of the `match` method that follows the PEG operator's semantics.

The `seedParentClauses` field contains a list of which parent clauses try to match this clause as a subclause at the same start position as the parent clause (i.e. before the parent clause has consumed any previous characters). This list of clauses is used to produce new `MemoKey` entries that are added to the priority queue to trigger parent clauses to be matched when the subclause is found to match at a given position (Listing 7).

The `canMatchZeroChars` field is set on initialization, and used to record whether a clause can match zero characters. Clauses that can match zero characters always match at every input position, which can dramatically increase the size of the memo table, so in an optimized pika parser, these clauses are handled differently in order to keep the memo table small (Section 4.1).

The `match` method is used to look up a subclause match in the memo table if `matchDirection == BOTTOM_UP`, or to do top-down recursive descent parsing for the optional lex preprocessing step (Section 4.3) if `matchDirection == TOP_DOWN`.

The `Char` subclass of `Terminal` (Listing 4) is a simple terminal clause that can match a single character in the input. Terminals directly check whether they match the input string, rather than looking in the memo table as with nonterminal clauses (compare to Listing 5).

The `Seq` subclass of `Clause` (Listing 5) implements the `Seq` PEG operator. The `match` method of this class requires all subclauses to match in order for the `Seq` clause to match, and the match start position must be updated for each successive subclause based on the number of characters consumed by the previous subclause matches. If top-down match direction is requested (by the optional lex preprocessing step, to initialize the memo table), then simple unmemoized recursive descent parsing is applied. If the match is bottom-up, i.e. in the normal case, subclause matches are instead looked up in the memo table. The `subclauseMatches` array is allocated if at least one subclause matches, and if all subclauses match in order, then the `subclauseMatches` array is passed into the `new Match` constructor to serve as the child nodes of the new match node, which forms part of the parse tree.

The other subclasses of `Clause` (implementing the other PEG operator types) can be found in the reference implementation (Section 8).

2.2.4 The `Match` *class (Listing 6).* This class is used to store matches (parse tree nodes) and their lengths (the number of characters consumed by the match) in memo entries, and to link matches to their subclause matches (i.e. to link parse tree nodes to their child nodes). The subclause index of the matching clause is also stored for `First` clauses in the `firstMatchingSubClauseIdx` field, so that matches of earlier subclauses can take priority over matches of later subclauses, as required by the semantics of the `First` PEG operator. Two matches can be compared using the `isBetterThan` method to determine whether one match is better than the second match, as defined in Section 2.1.5.

2.2.5 The `MemoEntry` *class (Listing 7).* This class stores the current best match for a specific `MemoKey` (i.e. for a specific clause and start position). The `addMatch` method checks whether `newMatch` is a better match than any current-best match stored in `bestMatch` for the memo entry. If it is, this method sets `bestMatch` to `newMatch`, and then since `bestMatch` was updated, any parent clauses of the matching clause are triggered to be matched at the same start position. This is accomplished by pairing each of the *seed parent clauses* of the matching clause with the current start position (Section 2.2.3), creating parent `MemoKey` instances, which are added the priority queue to expand the DP wavefront.

2.2.6 The `MemoTable` *class (Listing 8).* This class contains a two-level map, `memoTable`, which maps a clause to a start position to a `MemoEntry`. This two-level nesting of maps, and the use of the the sorted map interface `NavigableMap` (and its concrete implementation, `TreeMap`) for the inner map, allows for error recovery in $O(\log N)$ time after a syntax error, using a single lookup to find the next complete match of any grammar rule of interest.

The `lookUpBestMatch` method looks up a memo key in the memo table, and returns the current best match of the memo key's clause at the memo key's start position, or returns `null` if there was no match. The body of this method could be rewritten in a single line in a language that has fail-fast `null` handling, such as the "safe call" operator (`?.`) in Kotlin, in the following form:

```
return memoTable.get(memoKey.clause)?.get(memoKey.startPos)?.bestMatch;
```

The `addMatch` method looks up a memo entry in the two-level memo table using the memo key stored in a field of the provided `Match` object, adding a new entry to the memo table if one doesn't already exist for that memo key. (The function `Map.computeIfAbsent` is used rather than `Map.putIfAbsent` so that the constructors for `TreeMap` and `MemoEntry` are only called if there is no entry yet in the map for a given key.) This method then adds the provided `Match` object to the memo entry, which may trigger parent memo keys to be added to the priority queue (Section 2.2.5).

3 PRECEDENCE PARSING WITH PEG GRAMMARS

A *precedence-climbing grammar* tries to match a grammar rule at a given level of precedence, and, if that rule fails to match, defers or "fails over" to the next highest level of precedence (using a `First` clause, in the case of a PEG grammar). With top-down recursion, this means that deeper recursion frames (corresponding to nodes lower in the parse tree) typically represent higher levels of precedence[3]. Precedence climbing is implemented for all but the highest precedence clause, which uses a *precedence override pattern* such as parentheses to support multiple nested loops around the precedence hierarchy cycle. A primitive precedence-climbing grammar is shown in Listing 11.

The rule for each level of precedence in this example grammar has subclauses (corresponding to operands) that are references to rules at the next highest level of precedence. If the operands and/or the operator don't match, then the

[3]Therefore, technically, pika parsing implements *precedence descent* rather than precedence climbing, because it operates bottom-up, but the process will still be termed precedence climbing here, because the grammar rules are still specified using the top-down convention in the pika parser.

entire rule fails over to the next highest level of precedence. The root of the parse tree for any fully-parsed expression will be a match of the lowest-precedence clause, in this case `E0`.

This grammar can match nested unequal levels of precedence, e.g. `"1*2+3*4"`, but cannot match runs of equal precedence, e.g. `"1+2+3"`, since these cannot be parsed unambiguously without specifying the associativity of the operator. The above grammar also cannot handle direct nesting of equal levels of precedence, e.g. `"--4"` or `"((5*6))"`.

Since pika parsers can handle left recursion directly, these issues can be fixed by modifying the grammar into the form shown in Listing 12, allowing unary minus and parentheses to self-nest, and resolving ambiguity in the binary (two-argument) rules `E1` and `E0` by rewriting them into left-recursive form, which generates a parse tree with left-associative structure. A right-associative rule `Y1` with binary operator `OP` would move the self-recursive subclause to the right side, yielding the right-recursive form `Y1 <- (Y2 OP Y1) / Y2`, which would generate a parse tree with right-associative structure.

The reference pika parser (Section 8) can automatically rewrite grammars into this correct precedence-climbing form using the syntax shown in Listing 13, which can include an optional "`,L`" or "`,R`" associativity-specifying parameter after the precedence level, within the square brackets after the rule name. Passing the reference parser the grammar description in Listing 13 will automatically build the grammar shown in Listing 12.

4 OPTIMIZATIONS

4.1 Reducing the size of the memo table by not memoizing zero-length matches

The implementation of the parser as given so far is inefficient, because of it may create many entries in the memo table for zero-length matches.

The empty-string terminal ϵ (implemented in the reference parser by the class `Nothing`) always matches zero characters everywhere in the input, including beyond the end of the input string. `Nothing` is the simplest example of a clause that can match zero characters, but there are numerous other PEG clause types that can match zero characters, for example any instance of `Optional`, such as `(X?)`; any instance of `ZeroOrMore`, such as `(X*)`; instances of `First` where the last subclause can match zero characters[4], such as `(X / Y / Z?)`; and instances of `Seq` where all subclauses can match zero characters, such as `(X? Y* Z?)`. Since `Nothing` extends `Terminal`, in the implementation shown so far (Listing 2), one entry is added to the memo table for `Nothing` for every input position. Zero-length matches can cascade up the grammar from these terminals towards the root, adding match entries to the memo table at every input position for many grammar clauses. Mitigations should be put in place to avoid this.

It is safe to assume that no clause will never have the empty string ϵ as its first subclause, since this would be useless for any type of clause. If ϵ is disallowed in the first subclause position, then it is unnecessary to trigger upwards expansion of the DP wavefront by seeding the memo table with zero-length ϵ matches at every input position during initialization, since if ϵ is only ever used in the second or subsequent subclause of a clause, earlier subclauses will fill the role of triggering parent clauses. Preventing the creation of these initial ϵ matches requires changing the following line in `Grammar.parse` (Listing 2):

```
if (clause instanceof Terminal) { ... }
```

such that the terminal clause type `Nothing` (representing ϵ) is skipped for seeding into the memo table:

```
if (clause instanceof Terminal && !(clause instanceof Nothing)) { ... } .
```

[4]If a `First` clause has any subclause that can match zero characters, then all subsequent subclauses of the `First` clause will be ignored, because the subclause that can match zero characters will always match.

This change allows us to eliminate $O(n)$ memo table entries for ϵ, given input length n, or more likely, $O(n|G|)$ memo table entries for grammar of size $|G|$, since these unused zero-length matches can propagate upwards through multiple levels of the grammar. However, failing to start seeding the DP wavefront with ϵ matches can result in a document not being able to be parsed, since grammar rules that would be evaluated with top-down recursion may not be triggered for bottom-up matching with dynamic programming if zero-length matches are not seeded. Therefore, for the above optimization to work, some other changes are also necessary.

Given a grammar consisting of a single grammar rule `(P <- 'x'? 'y'+)`, which matches an optional `'x'` character followed by one or more `'y'` characters, we can rewrite the rule as `(P <- ('x' / ()) 'y'+)`, highlighting the fact that the first subclause `('x'?)` can match zero characters. This rewritten single-rule grammar will match the string `"xyyy"` and also the string `"yyy"` without any problems, using an unoptimized pika parser. However, after the above change is made to eliminate the seeding of matches of `Nothing`, this grammar will no longer match `"yyy"`, because `('x')` never matches at input position 0, so its parent clause `('x' / ())` is never triggered via the `seedParentClauses` mechanism. The second subclause of this parent clause, `()`, will never be matched since `Nothing` matches are not seeded. Therefore, the grandparent clause `(('x' / ()) 'y'+)` is not triggered for matching either.

The way to solve this problem is to modify the `MemoTable.lookUpBestMatch` method so that if a memo table entry does not contain a known match for a given memo key, *but the clause for that memo entry can match zero characters*, then a zero-length match is returned anyway (Listing 9). (This zero-length match does not need to be memoized, saving on memo table space.) Returning a zero-length match for clauses that can match zero characters whenever there is no match for the clause in the memo table at a given position fixes most of the issues with triggering parent clauses for matching.

This mechanism requires the `Clause.canMatchZeroChars` field to be set for each clause during initialization (Listing 10). The DAG of all grammar clauses is topologically sorted during initialization, then the initialization code iterates upwards through the DAG of clauses from terminals towards the root grammar rule, determining which clauses can match zero characters, given whether or not their subclauses can match zero characters.

One additional modification is needed to complete the optimization: on initialization, each PEG operator type must look at which subclauses of a parent clause can match zero characters in determining which parent clauses need to be seeded if the PEG operator matches. Specifically, for all clauses, any subclause that can start at the same start position as the parent clause must seed the parent clause at the same start position if the subclause matches. For the `Seq` PEG clause type, this is determined by finding all subclauses *up to and including the first subclause that matches one or more characters,* and adding the parent `Seq` clause to the subclause's `seedParentClauses` collection (Listing 10).

With each of these changes in place, the above example grammar, `(P <- ('x' / ()) 'y'+)` can now match the string `"yyy"`, because `('y'+)` is the first subclause of `(('x' / ()) 'y'+)` that cannot match zero characters. When `('y'+)` matches, this now triggers the parent clause `(('x' / ()) 'y'+)` to be matched at the same start position. The parent clause will then try to look up the memo entry for `('x'/ ())`, which fails to find a match for any input position, since `('x')` never matched at any input position, so its parent clause was never triggered for matching. Therefore, there are no matches for `('x'/ ())` in the memo table in any input position. However, the updated code for `MemoTable.lookUpBestMatch` (Listing 9) now returns a zero-length match when that memo lookup fails to find a memo entry for this subclause, which allows the toplevel clause `(('x' / ()) 'y'+)` to match at each `('y'+)` position, including at the first character position. This allows the parse to complete successfully and correctly.

The full implementation of these initialization and optimization steps can be seen in the reference parser (Section 8).

4.2 Reducing the size of the memo table by rewriting `OneOrMore` into right-recursive form

A naïve implementation of the `OneOrMore` PEG operator is iterative, matching as many instances of the operator's single subclause as possible, and assembling all subclause matches into an array, which is stored in the `subClauseMatches` field of the resulting `Match` object. However, when parsing the input from beginning to end, this causes a rule like `(Ident <- [a-z]+)` to add $O(m^2)$ match nodes to the memo table as a function of m, the maximum number of subclause matches of a `OneOrMore` clause. For example, given the input `"hello"`, this rule will store the following matches and subclause matches in the memo table: `[ [h,e,l,l,o], [e,l,l,o], [l,l,o], [l,o], [o] ]`.

This can be resolved by rewriting `OneOrMore` clauses into right-recursive form, e.g. the `OneOrMore` rule `(X <- Y+)` can be rewritten as `(X <- Y X?)`. Correspondingly, a `ZeroOrMore` rule `(X <- Y*)` can be rewritten as `(X <- (Y X?)?)`. The effect of this rewriting pattern is to turn runs of matches of the `Y` subclause into a linked list in the memo table, which means that successively shorter suffix matches are reused as each subsequent list tail, rather than being duplicated. Each match of `X` consists of either one or two subclause matches: a single match of subclause `Y`, and, optionally, a nested match of `X`, representing the tail of the list. With this modification, the number of subclause matches created and added to the memo table becomes linear in the maximum number of subclause matches of a `OneOrMore` match, i.e. $O(m)$.

Rewriting `OneOrMore` clauses into right-recursive form changes the structure of the parse tree into right-associative form. This transformation can be easily and automatically reversed once parsing is complete, by flattening each right-recursive linked list of `OneOrMore` subclause matches into an array of subclause matches of a single `OneOrMore` node. This is implemented in the reference parser by adding a `Match.getSubClauseMatches` method that flattens the right-recursive list for `OneOrMore` matches, and just returns the subclause match array without modification for other clause types.

4.3 Reducing the number of spurious matches using a lex preprocessing pass

Parsing an input right-to-left and then bottom-to-top produces the same parse tree as parsing an input top-to-bottom and then left-to-right, except that for the bottom-up variant, extra entries may be added to the memo table as a result of spurious matches. For example, if the grammar contains rules that match quoted strings or comments in a programming language's syntax, a bottom-up parser will not know the context to the left of the match when encountering apparent tokens inside the quoted string or comment, so the parser may add entries to the memo table for matches of random rules inside a quoted string or comment. These spurious matches will not be linked into the final parse tree, since when the entire document has been parsed, no higher clause in the grammar will be looking for the spurious match as a subclause match. Therefore, these spurious matches do not affect parsing, but they do cause parsing to take longer, and they create an extra memory overhead by increasing the size of the memo table.

Because of this potential for wasted work and wasted memory (which may be significant, e.g. in the case of long comments), bottom-up parsing will always be less efficient than top-down parsing. However, this can be mitigated using a *lex preprocessing step*.

Lexing (or tokenization) is a common preliminary step taken by many parsers. In the reference pika parser (Section 8), lexing is enabled by adding a rule called `Lex` that matches all necessary terminals (or tokens, quoted strings, comments, etc.) using a single `First` clause. The `Lex` rule matches a single token at a time, and the rule is repeatedly applied by the parser until all input is consumed. This step is run as an alternative to the standard memo table initialization step, which matches every terminal against every input position.

If a `Lex` rule is used, all terminals should be included somewhere in the `Lex` rule, so that the memo table is properly seeded with any terminal match that could end up as a leaf of the final parse tree. Otherwise, the appropriate parent clauses will not be triggered for subsequent bottom-up parsing, and the parser will not parse the entire input. Additionally, care should be taken to ensure that any given token in the input only matches one `Lex` subclause, otherwise some tokens may not properly seed all required terminal matches.

Each `Lex` subclause should be a shallow PEG clause, and should not be left-recursive, since top-down lex preprocessing runs unmemoized in order to minimize the size of the number of lowest-level entries in the memo table. Only after a complete `Lex` subclause is found to match is the tree of subclause matches for the subclause added to the memo table.

With lex preprocessing in place, the overhead of bottom-up parsing relative to top-down parsing will typically only amount to a small constant factor. Any remaining wasted work from spurious matches will be minimal, and this overhead is the price that can be paid for the benefits of direct support for left recursive grammars, and optimal error recovery.

5 AUTOMATIC CONVERSION OF PARSE TREE INTO AN ABSTRACT SYNTAX TREE (AST)

The structure of the parse tree resulting from a parse is directly induced by the structure of the grammar, i.e. there is one node in the parse tree for each clause and subclause of the grammar that matched the input. Many of these nodes can be ignored (e.g. nodes that match semantically-irrelevant whitespace or comments in the input), and could be suppressed in the output tree. Mapping the parse tree into a simpler structure that contains only the nodes of interest results in the Abstract Syntax Tree (AST).

The reference parser supports the syntax `(ASTNodeLabel ':' Clause)` for labeling any clause in the grammar. After parsing is complete, all matches in the parse tree without an AST node label are simply dropped. The resulting simplified tree is the AST, containing only nodes of interest (Fig. 1).

6 FUTURE WORK

The pika parser algorithm could be extended to enable *incremental parsing* [11], where when a document is parsed and then subsequently edited, only the minimum necessary subset of parsing work is performed, and the maximal amount of work from the prior parsing operation is reused.

To enable incremental parsing in a pika parser, first *dynamic back-references* must be added to memo-table entries (Section 2.1.4). The `MemoTable` class would add a field `backRefs` of type `Set<MemoKey>`, and then the `lookUpBestMatch` method (Listing 9) would be modified to always create a memo entry when a memo key is looked up in the hash table, if there is no memo entry in the memo table yet for that memo key, just as already done in *addMatch* (Listing 8). A new parameter `MemoKey parentMemoKey` would be added to the `lookupBestMatch` method, and after looking up or creating the memo entry for `memoKey`, the method would call `backRefs.add(parentMemoKey)`. This would add `parentMemoKey` to the set of back-refs for the memo entry, whether or not there was already a match in the memo table for `memoKey`.

Now given a set S of spans of the input that are modified through insertions and deletions:

(1) For each modified span S_i:
   (a) Delete all memo table entries that contain matches that are fully contained within S_i.
   (b) For any newly-added characters in S_i, match all terminals at all newly-added character positions, and if any matches are found, add the memo key of the match to `priorityQueue`. to start seeding the expansion of the DP wavefront upwards from newly-matching terminals.

(c) For all memo table entries M_j whose match partially overlaps with S_i (identified efficiently using an interval tree or similar), add the memo key of M_j to a new set E of entries that need to be re-matched.
(d) Find the transitive closure T of the memo keys of all matches reachable upwards from the memo entries whose memo keys are in in E, by following the links in `MemoEntry.backRefs`.
(e) Delete the memo entries for all memo keys in T.
(f) Add all memo keys in T to `priorityQueue` to be re-matched.

(2) Run the parsing algorithm as normal, until `priorityQueue` is empty.

Furthermore, the input string should be modified to be a linked list (or skip list), such that *relative addressing* is possible, otherwise after incrementally re-parsing using the above steps, the memo keys of all matches to the right of an insertion or deletion point in the input will point to the wrong character in the new input string. PEG rules always consume spans of the input in sequential order, so only the next input character after the current parsing position is needed – the algorithm actually does not need to be able to address characters using an absolute character position within the input. Therefore, the `MemoKey` class should replace `startPos` with a direct reference to the node in the linked list of input characters, and this list should be updated as the input is edited.

Because the memo table tends to accumulate spurious matches that are not linked into the final parse tree, and because `backRefs` is a set that may grow over time, to keep memory usage low in an incremental pika parser that is built into an editor or IDE, periodically the memo table could be garbage collected by running a complete parse from scratch.

There is some complexity to the above steps, and pika parsing is already extremely fast (scaling linearly in the length of the input and the depth of the parse tree), therefore, unless it is necessary to enable the editing of enormous documents in realtime, with the parse tree updated after every keystroke, the standard optimized pika parser is probably fast enough to use to re-parse the whole document after every edit operation, rather than implementing incremental parsing as described above. To make the effect of edit operations incremental, the abstract syntax tree can be "diffed" between parses of successive edit operations to find only the nodes in the abstract syntax tree that have changed as a result of the edit.

7 BROADER APPLICATIONS OF INVERSION OF RECURSION

The techniques presented in this paper for inverting a top-down recursive algorithm to turn it into a bottom-up dynamic programming algorithm have applications beyond parsing, and could bring similar benefits to other non-parsing recursive descent algorithms.

8 REFERENCE IMPLEMENTATION

An MIT-licensed reference implementation of the pika parser is available at: http://github.com/lukehutch/pikaparser

The reference parser is significantly optimized, for example all clauses are *interned*, so that rules that share a subclause do not result in duplicate rows in the memo table. Additionally, rule references are replaced with direct references to the rule, to save on lookup time while traversing the grammar during parsing. These optimizations make preprocessing the grammar more complicated, but result in significant performance gains.

The reference implementation includes a *meta-grammar* that is able to parse a PEG grammar written in ASCII notation, making it easy to write new grammars using the reference parser.

9 CONCLUSION

The *pika parser* is a new type of PEG parser that employs dynamic programming to parse a document in reverse (from the end to the beginning of the input), and bottom-up (from individual characters up to the root of the parse tree). Bottom-up parsing enables left-recursive grammars to be parsed directly, making grammar writing simpler, and enables almost perfect error recovery after syntax errors, making pika parsers useful for implementing IDEs. The pika parser operates within a moderately small constant performance factor of packrat parsing, i.e. is linear in the length of the input and the depth of the grammar. Several optimizations were presented to reduce the size of the memo table, and to reduce the amount of wasted work that must be performed due to spurious matches caused by bottom-up parsing. Mechanisms for implementing precedence climbing and left or right associativity were demonstrated, and several new insights were provided into precedence, associativity, and left recursion. The mechanism for inverting a top-down dynamic recursive algorithm to produce a bottom-up dynamic programming algorithm may have broader applications. A reference implementation of the pika parser is available under the MIT license.

REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. Compilers: principles, techniques, and tools.

[2] Alfred V Aho and Jeffrey D Ullman. 1972. *The theory of parsing, translation, and compiling*. Vol. 1. Prentice-Hall Englewood Cliffs, NJ.

[3] Alfred V. Aho and Jeffrey D. Ullman. 1977. *Principles of Compiler Design*. Addison-Wesley.

[4] Noam Chomsky. 1956. Three models for the description of language. *IRE Transactions on information theory* 2, 3 (1956), 113–124.

[5] Noam Chomsky. 1959. On certain formal properties of grammars. *Information and control* 2, 2 (1959), 137–167.

[6] William John Cocke, Jacob T Schwartz, and Courant Institute of Mathematical Sciences. 1970. *Programming languages and their compilers*. Courant Institute of Mathematical Sciences.

[7] H Daniel. 1967. Younger. Recognition and parsing of context-free languages in time n^3. *Information and control* 10, 2 (1967), 189–208.

[8] Sérgio Queiroz de Medeiros, Gilney de Azevedo Alvez Junior, and Fabio Mascarenhas. 2020. Automatic syntax error reporting and recovery in parsing expression grammars. *Science of Computer Programming* 187 (2020), 102373.

[9] Sérgio Queiroz de Medeiros and Fabio Mascarenhas. 2018. Syntax error recovery in parsing expression grammars. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. 1195–1202.

[10] Franklin Lewis DeRemer. 1969. *Practical translators for LR (k) languages*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[11] Patrick Dubroy and Alessandro Warth. 2017. Incremental packrat parsing. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. 14–25.

[12] Bryan Ford. 2002. *Packrat parsing: a practical linear-time algorithm with backtracking*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[13] Bryan Ford. 2004. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 111–122.

[14] Richard A Frost and Rahmatullah Hafiz. 2006. A new top-down parsing algorithm to accommodate ambiguity and left recursion in polynomial time. *ACM SIGPLAN Notices* 41, 5 (2006), 46–54.

[15] Richard A. Frost, Rahmatullah Hafiz, and Paul C. Callaghan. 2007. Modular and Efficient Top-down Parsing for Ambiguous Left-Recursive Grammars. In *Proceedings of the 10th International Conference on Parsing Technologies* (Prague, Czech REpublic) *(IWPT âĂŹ07)*. Association for Computational Linguistics, USA, 109âĂŞ120.

[16] Tadao Kasami. 1966. An efficient recognition and syntax-analysis algorithm for context-free languages. *Coordinated Science Laboratory Report no. R-257* (1966).

[17] Stephen Cole Kleene. 1951. Representation of events in nerve nets and finite automata. *RAND Research Memorandum* RM-704 (1951).

[18] Donald E Knuth. 1962. History of writing compilers. In *Proceedings of the 1962 ACM National Conference on Digest of Technical Papers*. 43.

[19] Donald E Knuth. 1965. On the translation of languages from left to right. *Information and control* 8, 6 (1965), 607–639.

[20] Joseph B Kruskal. 1983. An overview of sequence comparison: Time warps, string edits, and macromolecules. *SIAM review* 25, 2 (1983), 201–237.

[21] Bernard Lang. 1974. Deterministic techniques for efficient non-deterministic parsers. In *International Colloquium on Automata, Languages, and Programming*. Springer, 255–269.

[22] Robert McNaughton and Hisao Yamada. 1960. Regular expressions and state graphs for automata. *IRE transactions on Electronic Computers* 1 (1960), 39–47.

[23] Sérgio Medeiros, Fabio Mascarenhas, and Roberto Ierusalimschy. 2014. Left recursion in parsing expression grammars. *Science of Computer Programming* 96 (2014), 177–190.

[24] Peter Norvig. 1991. Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics* 17, 1 (1991), 91–98.

[25] Klaus Samelson and Friedrich L Bauer. 1960. Sequential formula translation. *Commun. ACM* 3, 2 (1960), 76–83.
[26] Michael Sipser. 2012. *Introduction to the Theory of Computation.* Cengage Learning.
[27] Masaru Tomita. 2013. *Efficient parsing for natural language: a fast algorithm for practical systems.* Vol. 8. Springer Science & Business Media.
[28] Alessandro Warth, James R Douglass, and Todd Millstein. 2008. Packrat parsers can support left recursion. In *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation.* 103–110.

FIGURES

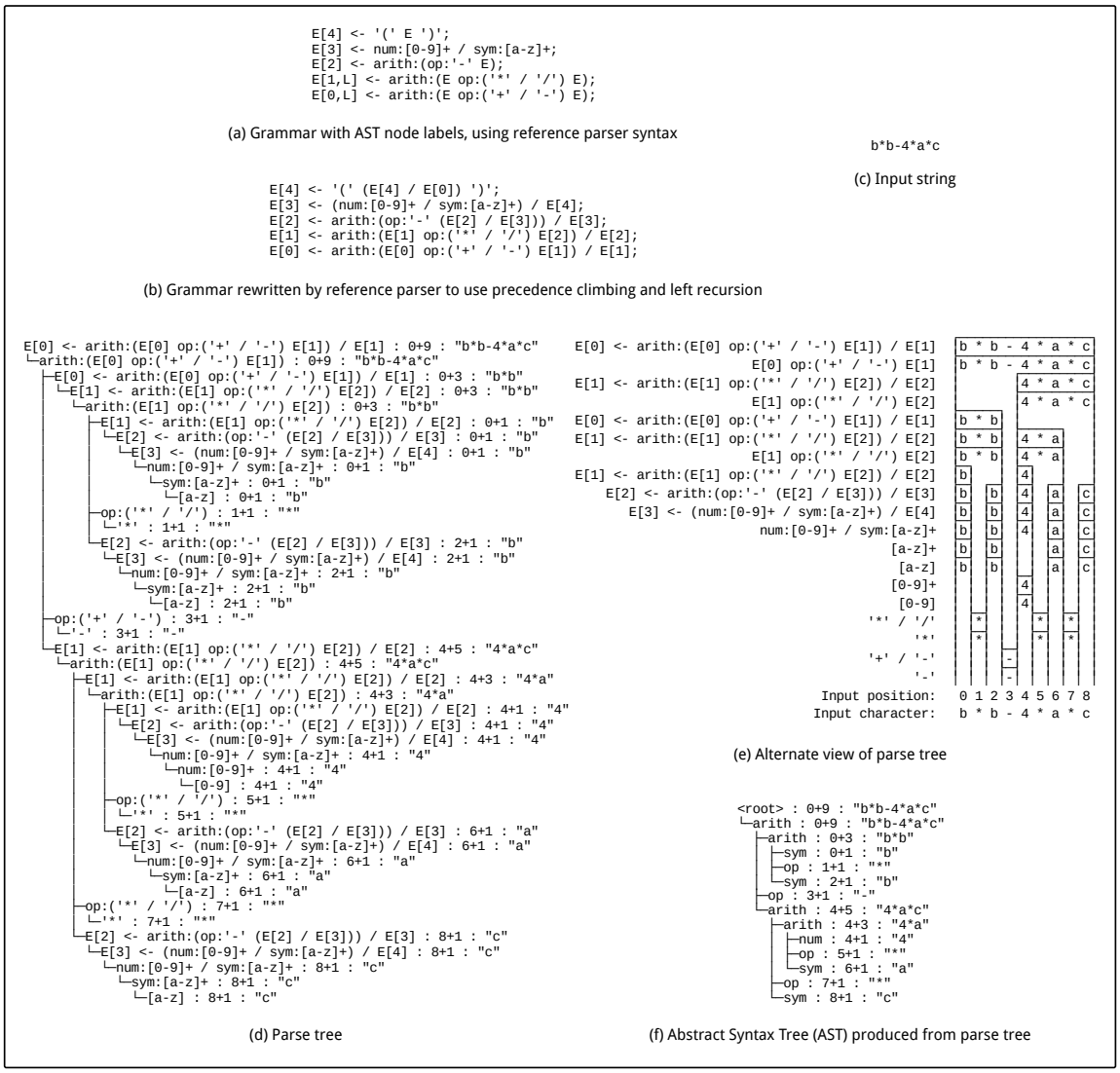

Fig. 1. Example of parse tree and abstract syntax tree produced by the reference pika parser

LISTINGS

```
class MemoKey implements Comparable<MemoKey> {
    Clause clause;
    int startPos;

    @Override
    public int compareTo(MemoKey other) {
        // Sort MemoKeys in reverse order of startPos
        int diff = -(this.startPos - other.startPos);
        if (diff != 0) {
            return diff;
        }
        // Break ties using topological sort index of clause, sorting bottom-up
        return this.clause.clauseIdx - other.clause.clauseIdx;
    }
}
```

Listing 1. The MemoKey class

```
class Grammar {
    List<Rule> allRules;
    List<Clause> allClauses;

    private static void matchAndMemoize(MemoKey memoKey, MemoTable memoTable, String input,
                PriorityQueue<MemoKey> priorityQueue) {
        var match = memoKey.clause.match(MatchDirection.BOTTOM_UP, memoTable, memoKey, input);
        if (match != null) {
            memoTable.addMatch(match, priorityQueue);
        }
    }

    MemoTable parse(String input) {
        var memoTable = new MemoTable();
        var priorityQueue = new PriorityQueue<MemoKey>();

        // Initialize the memo table: seed with terminal matches, generating initial priority queue
        for (var clause : allClauses) {
            if (clause instanceof Terminal) {
                for (int startPos = 0; startPos < input.length(); startPos++) {
                    matchAndMemoize(new MemoKey(clause, startPos), memoTable, input, priorityQueue);
                }
            }
        }

        // Main parsing loop: remove head element from the priority queue until empty; match against
        // the input; memoize if head element matches; add seed parent clauses to priority queue
        while (!priorityQueue.isEmpty()) {
            matchAndMemoize(priorityQueue.remove(), memoTable, input, priorityQueue);
        }
        return memoTable;
    }
}
```

Listing 2. The Grammar class

```
abstract class Clause {
    Clause[] subClauses;
    Set<Clause> seedParentClauses;
    boolean canMatchZeroChars;
    int clauseIdx;

    abstract Match match(MatchDirection matchDirection, MemoTable memoTable, MemoKey memoKey,
            String input);
}
```

Listing 3. The Clause class

```
class Char extends Terminal {
    char c;

    @Override
    Match match(MatchDirection matchDirection, MemoTable memoTable, MemoKey memoKey, String input) {
        return memoKey.startPos < input.length() && input.charAt(memoKey.startPos) == c
            ? new Match(memoKey, /* firstMatchingSubClauseIdx = */ 0, /* len = */ 1,
                    /* subclauseMatches = */ new Match[0]);
            : null;
    }
}
```

Listing 4. The implementation of the Char terminal (Terminal extends Clause)

```
class Seq extends Clause {
    @Override
    Match match(MatchDirection matchDirection, MemoTable memoTable, MemoKey memoKey, String input) {
        Match[] subClauseMatches = null;
        var currStartPos = memoKey.startPos;
        for (int subClauseIdx = 0; subClauseIdx < subClauses.length; subClauseIdx++) {
            var subClause = subClauses[subClauseIdx];
            var subClauseMemoKey = new MemoKey(subClause, currStartPos);
            var subClauseMatch = matchDirection == MatchDirection.TOP_DOWN
                    // Match top-down, unmemoized
                    ? subClause.match(MatchDirection.TOP_DOWN, memoTable, subClauseMemoKey, input)
                    // Match bottom-up, looking in the memo table for subclause matches
                    : memoTable.lookUpBestMatch(subClauseMemoKey);
            if (subClauseMatch == null) {
                // Fail after first subclause fails to match
                return null;
            }
            if (subClauseMatches == null) {
                subClauseMatches = new Match[subClauses.length];
            }
            subClauseMatches[subClauseIdx] = subClauseMatch;
            currStartPos += subClauseMatch.len;
        }
        return new Match(memoKey, /* firstMatchingSubClauseIdx = */ 0,
                /* len = */ currStartPos - memoKey.startPos, subClauseMatches);
    }
}
```

Listing 5. The implementation of the Seq PEG operator

```
class Match {
    MemoKey memoKey;
    int firstMatchingSubClauseIdx;
    int len;
    Match[] subClauseMatches;

    boolean isBetterThan(Match other) {
        // An earlier subclause match in a First clause is better than a later subclause match;
        // a longer match is better than a shorter match
        return (memoKey.clause instanceof First
                && this.firstMatchingSubClauseIdx < other.firstMatchingSubClauseIdx)
                        || this.len > other.len;
    }
}
```

Listing 6. The Match class

```
class MemoEntry {
    Match bestMatch;

    void addMatch(Match newMatch, PriorityQueue<MemoKey> priorityQueue) {
        if ((bestMatch == null || newMatch.isBetterThan(bestMatch))) {
            bestMatch = newMatch;
            for (var seedParentClause : newMatch.memoKey.clause.seedParentClauses) {
                MemoKey parentMemoKey = new MemoKey(seedParentClause, newMatch.memoKey.startPos);
                priorityQueue.add(parentMemoKey);
            }
        }
    }
}
```

Listing 7. The MemoEntry class

```
class MemoTable {
    Map<Clause, NavigableMap<Integer, MemoEntry>> memoTable = new HashMap<>();

    Match lookUpBestMatch(MemoKey memoKey) {
        var clauseEntries = memoTable.get(memoKey.clause);
        var memoEntry = clauseEntries == null ? null : clauseEntries.get(memoKey.startPos);
        return memoEntry == null ? null : memoEntry.bestMatch;
    }

    void addMatch(Match match, PriorityQueue<MemoKey> priorityQueue) {
        // Get the memo entry for memoKey if already present; if not, create a new entry
        var clauseEntries = memoTable.computeIfAbsent(match.memoKey.clause, c -> new TreeMap<>());
        var memoEntry = clauseEntries.computeIfAbsent(match.memoKey.startPos, s -> new MemoEntry());
        // Record the new match in the memo entry; possibly schedule parent memo keys for matching
        memoEntry.addMatch(match, priorityQueue);
    }
}
```

Listing 8. The MemoTable class

```
Match lookUpBestMatch(MemoKey memoKey) {
    var clauseEntries = memoTable.get(memoKey.clause);
    var memoEntry = clauseEntries == null ? null : clauseEntries.get(memoKey.startPos);
    if (memoEntry != null && memoEntry.bestMatch != null) {
        return memoEntry.bestMatch;
    } else if (memoKey.clause.canMatchZeroChars) {
        int firstMatchingSubClauseIdx = 0;
        for (int i = 0; i < memoKey.clause.subClauses.length; i++) {
            // The first matching subclause is the first subclause that can match zero characters
            if (memoKey.clause.subClauses[i].canMatchZeroChars) {
                firstMatchingSubClauseIdx = i;
                break;
            }
        }
        return new Match(memoKey, firstMatchingSubClauseIdx, /* len = */ 0,
                /* subclauseMatches = */ new Match[0]);
    }
    return null;
}
```

Listing 9. Improvement of the `MemoTable.lookUpBestMatch` method to handle zero-length matches with minimal memoization

```
@Override
public void determineWhetherCanMatchZeroChars() {
    // For Seq, all subclauses must be able to match zero characters for the whole clause to
    // be able to match zero characters
    this.canMatchZeroChars = true;
    for (var subClause : this.subClauses) {
        if (!subClause.canMatchZeroChars) {
            this.canMatchZeroChars = false;
            break;
        }
    }
}

@Override
public void addAsSeedParentClause() {
    // All sub-clauses up to and including the first clause that matches one or more characters
    // needs to seed its parent clause if there is a subclause match
    for (var subClause : this.subClauses) {
        subClause.seedParentClauses.add(this);
        if (!subClause.canMatchZeroChars) {
            // Don't need to any subsequent subclauses to seed this parent clause
            break;
        }
    }
}
```

Listing 10. Methods of the Seq class used to determine whether a Seq clause can match zero characters, and to find which subclauses need to seed the Seq clause as a parent

```
E4 <- '(' E0 ')';
E3 <- ([0-9]+ / [a-z]+) / E4;
E2 <- ('-' E3) / E3;
E1 <- (E2 ('*' / '/') E2) / E2;
E0 <- (E1 ('+' / '-') E1) / E1;
```

Listing 11. A primitive precedence-climbing grammar

```
E4 <- '(' (E4 / E0) ')';
E3 <- ([0-9]+ / [a-z]+) / E4;
E2 <- ('-' (E2 / E3)) / E3;
E1 <- (E1 ('*' / '/') E2) / E2;
E0 <- (E0 ('+' / '-') E1) / E1;
```

Listing 12. Improved precedence-climbing grammar, supporting left associativity and self-nesting of parentheses and unary minus

```
E[4] <- '(' E ')';
E[3] <- [0-9]+ / [a-z]+;
E[2] <- '-' E;
E[1,L] <- E ('*' / '/') E;
E[0,L] <- E ('+' / '-') E;
```

Listing 13. Shorthard reference parser notation for the grammar of Listing 12