Pika Parser: A backwards packrat parser employing dynamic programming

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, and the structure of recursive calls directly parallels the structure of the grammar. A memoized recursive descent parser, or a *packrat parser*, takes 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 recursive descent parsers to get stuck in infinite recursion, and (ii) it is difficult or impossible to optimally recover from syntax errors in a recursive descent parser. The highly surprising solution to both of these problems is to parse the input string *backwards*, from right-to-left and bottom-to-top (i.e. from the end of the input towards the beginning, and from terminals towards the root of the parse tree), rather than the standard recursive descent parsing order of top-to-bottom and left-to-right. The *pika parser* is a new type of packrat parser that employs dynamic programming to parse the input backwards, enabling the parser to directly handle left-recursive grammars, and to easily and optimally recover from syntax errors, while maintaining the linear-time performance of packrat parsers to within a moderately small constant factor. Numerous 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 Parser: A backwards packrat parser employing dynamic programming. *ACM Trans. Program. Lang. Syst.* TBD, TBD, Article TBD (TBD 2020), 22 pages. https://doi.org/TBD

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

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

Manuscript submitted to ACM

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 in theory has near-perfect error recovery capability since all rules are matched against all substrings of the input. However, the CYK algorithm is inefficient for very large input sizes n, 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 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 nice 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, the worst case time complexity of a recursive descent parser is exponential in the length of the input and the complexity of the grammar, due to unlimited lookahead, and the lack of termination of already-visited nodes during depth-first recursion over the Directed Acyclic Graph (DAG) structure of the grammar.
(2) Without special handling [23], left-recursive grammars cause naïve recursive descent parsers to get stuck in infinite recursion, which leads directly to 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 limits a compiler's ability to provide 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). Before now, error recovery has remained far from solved.

1.4 The pika parser

The *pika parser*[1] is a new type of packrat parser that applies PEG operators from right to left, from the end of the input back towards the beginning, and bottom-up, from terminals towards the root of the parse tree (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 roughly the same big-Oh time complexity as packrat parsing, i.e. scales linearly in the size of the input and the depth of the parse tree.
(2) Bottom-up parsing directly handles left-recursive grammars, and grammars containing arbitrary numbers of cycles, without modification. This can greatly simplify grammar design.
(3) Bottom-up parsing conveys near-perfect error recovery capabilities on the parser, since when parsing right-to-left from the end of the document back towards the beginning, if an error is encountered, the input to the right of the syntax error was already 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 over the syntax error to the next valid match of any grammar rule of interest. This will be particularly helpful in providing code assist features for IDEs to work robustly in spite of syntax errors.

The pika parser is defined to work with PEG grammars, 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 parsers, which represent grammar rules as *generative productions*, PEG parsers operate 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

[1] A pika (/'paɪkə/) 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.

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.

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 herein 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* (where a rule is referred to by name). 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.

Nonterminals can be constructed out of the PEG operators shown in Table 1, 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 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, attempts to match further subclauses are abandoned. If no subclauses match, `First` clause does not match. The `First` clause gives a top-down PEG parser *limited backtracking behavior*.
- 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 it matches. This operator also 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 matches zero times), then the `ZeroOrMore` still matches the input, consuming zero characters (i.e. this operator always matches). The clause $e*$ can be automatically transformed into $(e+)?$, which reduces to $e+/\epsilon$, 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. `Longest` can also help with writing lex rules (Section 3.3).

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.

It is difficult to modify top-down parsers to handle left recursion and to have robust error recovery characteristics. However, both of these problems are solved (or nearly solved) "for free" in most bottom-up parsers. 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. This is implemented using lookups of earlier-computed entries in a *DP table* (which is an indexed memo table), and 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 recurrence $F_i = F_{i-1} + F_{i-2}$ can be implemented using the three methods as follows:

(1) Inefficiently, top-down, using recursion. This 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 size of the DAG.

(2) Efficiently, top-down, using memoized recursion. This avoids duplicated work by storing just the two previous entries F_{i-1} and F_{i-2} in a small "memo table", which terminates the recursion at the root node, i.e. this transforms the calculation of F_i into an iterative algorithm over F_0 to F_i. However, the iteration cannot start at F_i as with method (1), it must start at F_0 and iteratively populate the memo table, calculating each subsequent term until F_i is reached.[2]

(3) Efficiently, bottom-up, by inverting the recurrence relation. This algorithm starts by seeding the dynamic programming table (the memo table) with the base cases (also termed the terminals, or leaf nodes for recursion) $F_0 = 0$ and $F_1 = 1$. Both these recurrence terms are added to an initially-empty "active set" (also known as the "dirty set" or "DP wavefront"). Referring to F_i as the "parent clause" of the recurrence, and each of F_{i-1} and F_{i-2} to each be "subclauses" of the recurrence, each new subclause calculation that is added to the active set needs to trigger the evaluation of its corresponding parent clause (since in top-down evaluation, the parent clause recurses to its subclauses). For the Fibonacci example, each F_{i-1} in the active set should trigger the calculation of F_i, and each F_{i-2} in the active set should also trigger the calculation of F_i. Alternatively, these two seeding actions can be collected for a subclause recurrence term $F_{i'}$ by *inverting the indexing function* to find the parent clause index in terms of the subclause index. When $F_{i'}$ is computed and added to the active set, it should trigger the calculation of both the parent clauses $F_{i'+1}$ and $F_{i'+2}$, since $F_{i'}$ is a subclause of the recurrence $F_{i'+1} = F_{i'+1-1} + F_{i'+1-2} = F_{i'} + F_{i'-1}$, and also of the recurrence $F_{i'+2} = F_{i'+2-1} + F_{i'+2-2} = F_{i'+1} + F_{i'}$. In other words, triggering parent clauses when subclauses are evaluated requires both reversing the arrow direction for the edges in the call graph between a parent clause and its subclauses, and inverting the index function so that the recurrence is expressed in terms of the subclause index rather than the parent index. In the Fibonacci example, the relationship $F_i \longrightarrow F_{i-1}$ needs to be inverted to $F_{i'+1} \longleftarrow F_{i'}$, and the relationship $F_i \longrightarrow F_{i-2}$ needs to be inverted to $F_{i'+2} \longleftarrow F_{i'}$. Some care should also be taken to ensure that DP table values are computed before they are needed, or alternatively, that if a parent clause is triggered for evaluation before all its subclause recurrence results have been computed, the parent clause will be re-evaluated once all its subclauses have been computed. Once a parent clause $F_{i'+j}$ has been computed, it should be added to the active set, and its own parent clauses should then be triggered by the same mechanism, expanding the wavefront outwards until there are no more updates to the memo table and the active set is empty.

Almost all DP algorithms taught in an algorithms classes use method (2), which requires the specification of a careful evaluation order (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]). This is to ensure that the recurrence dependencies of each new table entry are always computed before they are needed. However, despite the recurrence table being populated bottom-up, the recurrence is always applied top-down for each new table entry, which does not require inverting the recurrence.

Method (3) 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 (2) can be exceptionally difficult. The simplest

[2]This apparently-reversed order of evaluation in method (2) may seem like an inversion of method (1), since method (1) starts with F_i rather than F_0, but actually method (1) is a *postorder* depth-first traversal, so the actual order in which terms are computed is the same for both methods.

efficient solution to many of these problems requires method (3), 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, this is the method used to implement the pika parser (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 (3)).

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 Incorrect DP parsing order: bottom-up, in parallel across the input. With bottom-up parsing formulated as a dynamic programming problem, it may seem possible to even 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. 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 4) 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 table entry for `3` was consumed as a subclause match of the first fragment before precedence-climbing replaced the match at `3` 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 matched and memoized. (All DP algorithms fail if the DP table is not populated in the right order.)

2.1.3 Correct DP parsing order. The match status for a PEG operator depends *only* upon (i) the match results for higher-precedence (lower-level) 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. Therefore, since all match operations depend upon entries either lower in the grammar (of higher precedence) or to the right of the current entry in the memo table, we can directly deduce the correct order to populate the memo table, so that all subclause matches of any parent clause are already in the memo table before the parent clause is matched. The correct matching order runs *from the end of the document 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).*.

This yields the only correct population order for the memo table: whereas a packrat parser operates top-to-bottom and then left-to-right, a pika parser operates from right-to-left and then bottom-to-top – a direct mirror-image of the top-down parsing order. 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. To implement any DP algorithm, the recurrence relation must be inverted (Section 1.7, method (3)). Inverting the recurrence relations represented by a set of PEG grammar rules would seem to require *dynamic back-references* from subclauses to their parent clauses. In particular, back-references from a subclause at input position $i + m$ to a parent `Seq` clause at input position i dynamically depend upon how many characters m were consumed by the previous subclauses. However, under the assumption that everything to the right of the current parse position has already been completely parsed and memoized because of the chosen DP evaluation order, no additional matches will ever be added to the memo table to the right of the current parse position. Therefore, there is never any need to update parent matches to the *left* of the current match position, by inverting the function mapping the start position of the parent clause match to the start position of the subclause match, $i \rightarrow (i + m)$, in order to express the start position of the parent clause match in terms of the start position of the subclause match, i.e. $(i' - m) \leftarrow i'$. In fact when a subclause matches at start position i in the input, the only memo entries (corresponding to parent clause(s) and start position(s)) that need to be scheduled for matching share the same start position i as the subclause, i.e. there is never any dynamic offset needed to convert from subclause start position i to parent clause start position i, since a parent always matches its first subclause at the parent's own start position. The recurrence indexing function $i \rightarrow i$ is the identity function, which needs no inversion. 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.)

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), stored within a memo key (the position of the entry in the memo table), the length of the match (the number of characters consumed by the match), and an array of subclause matches (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 and 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 table entry is only updated with a new match if there is no previous match for that memo table 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 table entry containing a previous match is not updated with a new match if the two matches are of the same length (and have the same matching subclause index, in the case of `First` clauses). The parent clauses of a memo table entry's clause are only triggered for evaluation (expanding the DP wavefront upwards) when the match for 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 grammar cycle must increase the length of match by at least one character.
- Left recursion will terminate early (before consuming the entire span of the input string to the right of the match position) if no more input can be consumed by the rule. This is indicated by two loops around the grammar cycle having equal match length. Since the length of the newer match is the same as the length of the older match, the memo table entry is not updated with the newer match, therefore the parent clauses of the memo table entry's clause are not triggered for evaluation by adding them to the DP wavefront. This terminates the upwards expansion of the grammar rule.

Lower-level matches in the parse tree match monotonically fewer characters than their ancestral matches, and lower-level matches also correspond to higher precedence matches than their ancestral matches (due to precedence climbing). This yields the following interesting corollary, where *absolute precedence* takes into account the number of loops around a precedence hierarchy through a *precedence-breaking pattern* in the highest precedence clause (e.g. parentheses – Section 4):

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 a precedence hierarchy, where the matches have the same start position, are monotonically reverse-ordered with respect to the absolute precedence level of the matches. Longer matches correspond to lower absolute precedence; shorter matches correspond to higher absolute precedence. This allows us to compute *a monotonic function of the difference in absolute precedence levels of two matches that share a start position* simply by comparing the match lengths.

2.1.6 Proof of optimality of error recovery characteristics in a pika parser. Because the input is parsed from right to left, and the memo table is fully populated to the right of the current parse position, the span of input after a syntax error is always parsed optimally, because the syntax error has not even been encountered yet in the input, in right-to-left order. After the syntax error is encountered, i.e. to the left of the syntax error, clause matching will recover as soon as a

complete match of a given clause can be found starting at the current parse position and ending before the syntax error. (PEG clause matches only depend upon the span of input from their start position to their end position, and not upon any characters after the end position.)

Therefore, parsing of input after a syntax error is optimal for bottom-up parsing, and parsing of input before a syntax error has the same characteristics for bottom-up parsing as it does for top-down parsing (which is that rules that match completely before the syntax error will still match, and rules whose matches overlap with the syntax error will fail to match).

Combining these two observations, parsing between any closest pair of syntax errors in the input is also optimal.

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, etc.). Listings are given in Java notation (for increased clarity compared to an invented pseudocode); however, simple assignment constructors have been omitted, and some Java-specific details and quirks have been simplified or removed, as have parsing optimizations. See the reference parser for the full implementation (Section 8).

2.2.1 The `MemoKey` *class.* (See Listing 1.) This class with 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 input character).

We define the *natural ordering* of this class[3] such that `MemoKey` instances are naturally sorted into decreasing order of `startPos` and then into bottom-up topological order of `clause` (i.e. based on a reverse topological sort of the DAG structure of the grammar, which produces a clause index, `clause.clauseIdx`, ordered from lowest to highest clause). This natural ordering is used to ensure that `MemoKey` instances are removed from a priority queue of outstanding work items in the correct DP table population order, right-to-left and then bottom-to-top.

2.2.2 The `Grammar` *class.* (See Listing 2.) This class has a single method, `parse`, which parses an input string. The `Grammar` class contains a collection of `Rule` objects for the rules of the grammar, and also has an `allClauses` field, which is a set 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 the locations that all terminals match the input. The empty string match, ϵ (written `"()"` in ASCII notation, and implemented using the `Nothing` class), is not memoized in order to keep the memo table small, since this clause matches everywhere (Section 3.1). 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 3.3).

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 table entry designated by `memoKey` (also stored at `match.memoKey`). This `addMatch` method call also adds the *seed parent clauses* of each terminal

[3]This is accomplished by implementing the `Comparable<MemoKey>` 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)`

match to the priority queue (these are generated from the parent clauses of each terminal, by looking upwards in the grammar, and the start position of the subclause match is used as the start position of the parent clause).

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`, in bottom-up order (i.e. memoized). If the clause matches, the match is added to the memo table, again possibly adding new entries to the priority queue for seed parent clauses of the matching clause.

This is the entire parsing algorithm, at the highest level, although `clause.match` and `memoTable.addMatch` still need to be detailed.

2.2.3 The `Clause` *class.* (See Listing 3.) This class represents a PEG operator or clause and its subclauses. This class has a subclass `Terminal`, which is the superclass of all terminal clauses. One of these two classes is subclassed for each PEG operator type, in particular to provide 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 parent clause's start position (i.e. before the parent clause has consumed any previous characters). This triggers parent clauses to be matched when a memo table match for a subclause is updated.

The `canMatchZeroChars` field is used to record whether a clause can match while consuming 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 these clauses are handled differently in order to keep the memo table small (Section 3.1).

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 matching terminals or for the optional lex preprocessing step (Section 3.3) if `matchDirection == TOP_DOWN`.

The `Char` class (Listing 4) is a simple subclass of `Terminal` that can match a single character in the input. Terminals match directly against the input, rather than looking in the memo table.

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 `Seq` match node in 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.* (See Listing 6.) This class is used to store matches (parse tree nodes) and their lengths in memo entries, and to link matches to their subclause matches (child parse tree nodes). The subclause index of the matching clause is also stored for `First` clauses, 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 to determine whether the first match is better than the second match, as defined in Section 2.1.5.

2.2.5 The `MemoEntry` *class.* (See Listing 7.) This class stores the current best match for a specific `MemoKey` (clause and start position). The `addMatch` method checks if `newMatch` is better than the current `bestMatch` for the memo entry, and if so, sets `bestMatch` to `newMatch`. If `bestMatch` is updated, parent clauses are triggered to be matched by pairing the memo entry's start position with each of the seed parent clauses, creating parent `MemoKey` instances, which are added the priority queue, expanding the DP wavefront.

2.2.6 The `MemoTable` *class.* (See Listing 8.) This class contains a double-level map, `memoTable`, mapping a clause to a second-level map, which maps a start position within the input string to a `MemoEntry`. This nesting of maps allows all matches anywhere in the input to be found in $O(1)$ amortized time for any clause and start position, whereas the use of the sorted map `NavigableMap` (and its concrete implementation, `TreeMap`) for the inner map implementation allows the next successful match of any clause after any input position to be found in $\log(N)$ time (e.g. for recovery after a syntax error).

The `lookUpBestMatch` method looks up a memo key in the memo table, and returns the current best match of the clause given in the memo key at the start position given in the memo key, or returns `null` if there was no match.

The `addMatch` method looks up a memo entry in the two-level memo table given the memo key in 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 OPTIMIZATIONS

3.1 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 an interesting corner case involving zero-length matches.

The empty-string terminal ϵ, which always matches zero characters everywhere in the input (including beyond the end of the input string), is the simplest example of this type of match. There are numerous other PEG clauses that can match zero characters, for example any instance of `Optional` or `ZeroOrMore`; 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?)`. Zero-length matches can cascade up the grammar from terminals towards the root, filling up the memo table with many zero-length matches. Therefore, several mitigations are put in place for these clauses in order to avoid adding zero-length matches to the memo table.

It is safe to assume that a parent clause will never have ϵ as its first subclause, since this would be useless for any type of parent clause. (However, any other clause type that can match zero characters can be validly used as the first subclause of a clause.) Since ϵ cannot be in the first subclause position, 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 (Listing 2), since any other subclause in the first subclause position can do that. This allows us to eliminate $O(N)$ memo table entries for ϵ, given input of length N, as well as a potential chain of spurious zero-length matches upwards through the parse tree at each input position. However, it comes at a cost of slightly higher code complexity, as shown in the revision to `MemoTable.lookUpBestMatch` as shown below. If a memo entry is looked up by a parent clause, and

[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.

there is no known match for that entry, but the subclause can positively match zero characters, then a zero-length "placeholder" match must be returned, since the subclause can always match zero characters even if no longer match is currently known. This new match does not need to be memoized, which again prevents the memo table from filling up with zero-length matches.

This revised method (Listing 9) requires the `canMatchZeroChars` field to be set for each clause during initialization. The DAG of all grammar clauses is topologically sorted, 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 any of their subclauses can match zero characters. The implementation of this initialization step can be seen in the reference parser (Section 8).

3.2 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 forms 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 given the maximum number of `OneOrMore` subclause matches, M. 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. given the 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 is to turn runs of the `Y` subclause into a linked list. Each match of `X` consists of one or two subclause matches: a match of `Y`, and, optionally, the `X` match representing the tail of the list. With this modification, the number of match nodes created and added to the memo table is $O(M)$ in the maximum number of `OneOrMore` subclause matches, 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 the right-recursive linked list of `OneOrMore` subclause matches into an array of subclause matches of a single `OneOrMore` node.

3.3 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 preceding context when encountering tokens inside the quoted string or comment, so the parser may add entries to the memo table for patterns encountered inside the quoted string or comment that match some random rule. These spurious matches will not be linked into the final parse tree, since at some point before the entire document has been parsed, no higher clause in the grammar will be looking for the spurious match as a subclause match.

Because of this potential for wasted work (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, 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 by adding them to the priority queue. 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, so bottom-up parsing will not be able to parse the whole input.

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 initial 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 it is worth considering this overhead to be the price paid for the benefits of direct support for left recursive grammars, and optimal error recovery.

4 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), for all but the highest precedence clause (which uses a *precedence-breaking pattern* such as parentheses, if multiple loops through the precedence hierarchy need to be supported).

The rule for each level of precedence in the grammar in Listing 10 has subclauses (corresponding to operands) that are references to rules at the next highest level of precedence. In top-down parsing, this leads to lower levels of precedence matching at higher levels of the parse tree, and vice versa. If the operands and/or operator don't match, then the 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 11, assuming `E1` and `E0` are intended to be left-associative.

Similarly a right-associative rule `Y1` with binary operator X would use the opposite order of operands, yielding the right-recursive form `Y1 <- (Y2 X Y1) / Y2`.

The reference pika parser (Section 8) can automatically rewrite grammars into the correct precedence-climbing form, with an optional `L` or `R` associativity-specifying parameter, using the syntax shown in Listing 12.

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 are not needed (for example nodes that match semantically-irrelevant whitespace or comments in the input), and could be suppressed in the output tree. The reference parser allows the user to mark any clause in the grammar with a label,

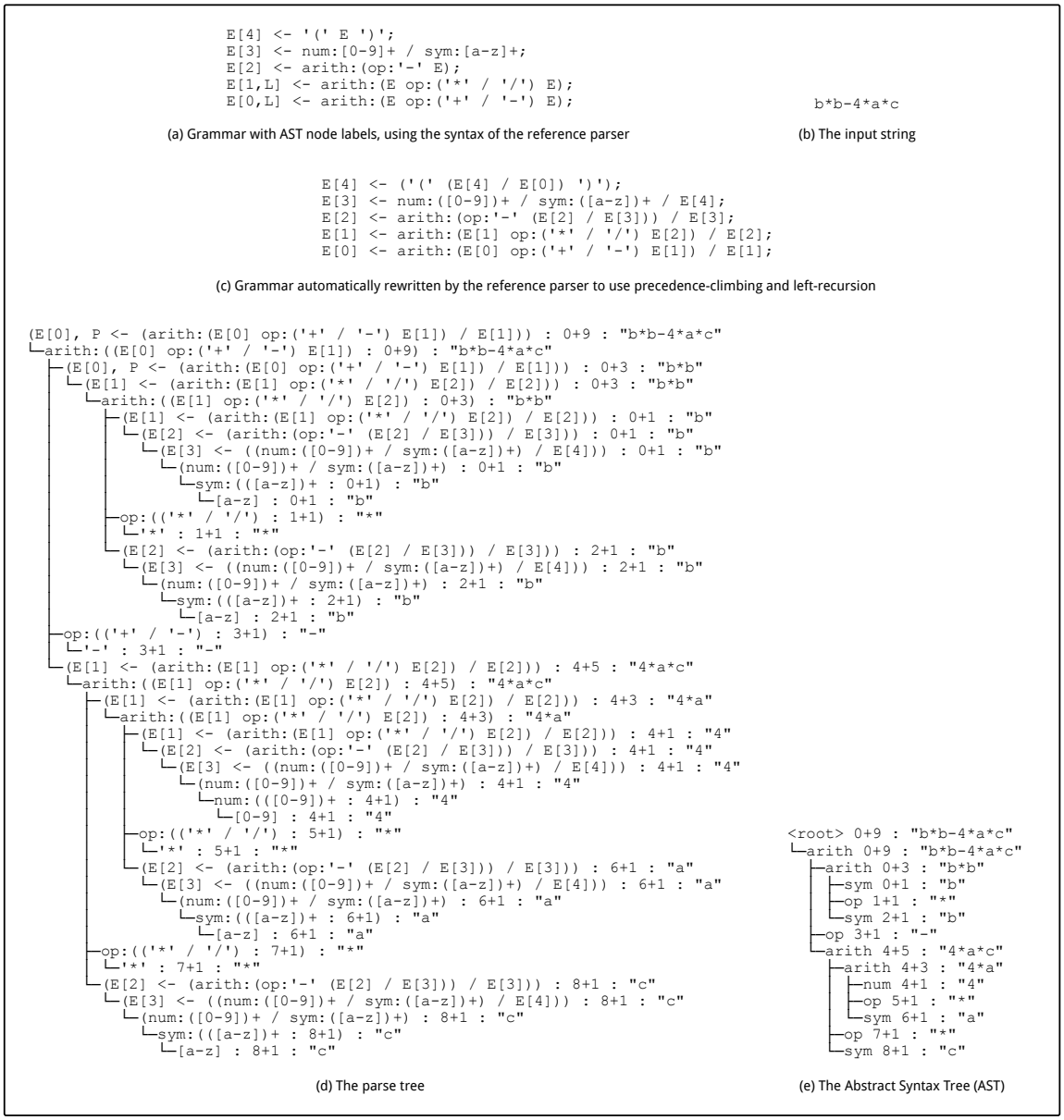

Fig. 1. Automatic conversion of a parse tree into an Abstract Syntax Tree (AST) via AST node labels on grammar clauses

using the syntax `(ASTNodeLabel ':' Clause)`. After parsing is complete, all matched clauses in the parse tree that do not have an AST node label are simply dropped. The resulting tree is the Abstract Syntax Tree (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 `Set<MemoKey> backRefs`, 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, and a new memo table entry is added if there was not was already a match in the memo table for that memo key, 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 for entries that need to be re-matched.
   (d) Find the transitive closure T consisting of all memo keys in E, and the memo keys of all memo table entries 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`.
(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 liked list of input characters, and this list should be updated as the input is edited. Then the `next` pointer connecting the nodes in the input linked list to the following character would itself be annotated with back-refs to memo entries that tried to access the following character. These back-refs would be used to trigger re-evaluation in case of an insertion or deletion, just as with `backRefs` as described above.

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 extremely fast (scaling linearly in the length of the input and the depth of the parse tree), therefore unless it is possible that a user will want to edit enormous documents in realtime, with the parse tree updated after every keystroke, it is probably not unreasonable to simply re-parse the whole document after every edit operation, rather than implementing full incremental parsing.

7 BROADER APPLICATION 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 may have applications beyond parsing, and could bring similar benefits to other non-parsing recursive 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. These optimizations make the 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 bottom-up, from characters to the root of the parse tree. Bottom-up parsing enables left-recursive grammars to be parsed directly, making grammar writing simpler, and also enables almost perfect error recovery after syntax errors are encountered, making pika parsers useful for implementing IDEs. The pika parser operates within a moderately small constant performance factor of packrat parsing – both are 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 due to spurious matches. 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 '07)*. Association for Computational Linguistics, USA, 109–120.
[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.

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;

    MemoTable parse(String input) {
        // Initialize memo table
        var memoTable = new MemoTable();
        var priorityQueue = new PriorityQueue<MemoKey>();
        for (var clause : allClauses) {
            if (clause instanceof Terminal && !(clause instanceof Nothing)) {
                for (int startPos = 0; startPos < input.length(); startPos++) {
                    var memoKey = new MemoKey(clause, startPos);
                    var match = clause.match(MatchDirection.TOP_DOWN, memoTable, memoKey, input);
                    if (match != null) {
                        memoTable.addMatch(match, priorityQueue);
                    }
                }
            }
        }
        // Main parsing loop
        while (!priorityQueue.isEmpty()) {
            var memoKey = priorityQueue.remove();
            var match = memoKey.clause.match(MatchDirection.BOTTOM_UP, memoTable, memoKey, input);
            if (match != null) {
                memoTable.addMatch(match, priorityQueue);
            }
        }
        return memoTable;
    }
}
```

Listing 2. The Grammar class

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

    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

```
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 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

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

Listing 10. Incomplete 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 11. Fixed precedence-climbing grammar with two left-recursive rules, `E1` and `E0`

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

Listing 12. Fixed precedence-climbing grammar, using reference parser syntax