Pika Parser: A parallelizable bottom-up packrat parser employing dynamic programming

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

A recursive descent or top-down parser is built from a set of mutually-recursive functions where each function directly implements one of the nonterminals of a grammar. The code structure of a recursive descent parser exactly parallels the grammar. Recursive descent parsers are extremely simple to write, but suffer from three significant problems: (1) the worst case time complexity of a recursive descent parser can be exponential in the length of the input; (2) left-recursive grammars cause the parser to get stuck in infinite recursion; (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 past the error. Problem (1) can be solved with memoization, resulting in what is known as a *packrat parser*; however, problems (2) and (3) have remained difficult to solve. We present a new type of parser, the *pika parser*, that solves these problems by using dynamic programming to implement bottom-up parsing, starting with individual character terminals and building upwards towards the root node of the Abstract Syntax Tree (AST). Bottom-up parsing (1) employs memoization, by definition as a dynamic programming algorithm, which (like packrat parsing) solves the exponential worst-case behavior of non-packrat recursive descent parsing, resulting in performance linear in the size of the input and the depth of the parse tree; (2) directly enables the use of some left-recursive grammars with some inputs, simplifying grammar design (a simple new non-left-recursive grammar-writing pattern is provided to address the use of left-recursive grammars in the general case); (3) conveys near-perfect error recovery capabilities on the parser, since all parsing depends only on the immediate surrounding context in the input; and as a bonus (4) enables parsing to be parallelized across the input, so that multiple parts of the input can be parsed simultaneously, utilizing available cores.

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, left-recursive grammars, parsing error recovery, memoization, dynamic programming, parallel computing

ACM Reference Format:
Luke A. D. Hutchison. 2020. Pika Parser: A parallelizable bottom-up packrat parser employing dynamic programming. *ACM Trans. Program. Lang. Syst.* TBD, TBD, Article TBD (TBD 2020), 29 pages. https://doi.org/TBD

1 INTRODUCTION

1.1 Parsing generative grammars

Since the earliest beginnings of computer science, computer scientists have sought to identify robust algorithms for parsing formal languages [1, 2]. Careful work to establish the theory of computation [24] 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.

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

Manuscript submitted to ACM

Generative systems of grammars, particularly Chomsky's context-free grammars (CFGs) [4] and Kleene's regular expressions [16] became the workhorses of parsing of languages and data formats, and have been used nearly ubiquitously over several decades. However, the theory of these languages and the algorithms used to parse them are relatively complex, and additionally, inherent ambiguities and nondeterminism increase the implementational complexity of parsers for these classes of language [20, 25], sometimes resulting in parsing conflicts, or exponential blowup in time or space [16, 21].

1.2 Bottom-up parsing

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, and these trees are joined together into larger trees as higher-level grammar rule *productions* or *reductions* are able to be 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, 17, 23], the later development of LR [18] and LALR (lookahead) [10] parsers, and the more complex GLR parser [20] 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. Error recovery in all shift-reduce parsers is complex and difficult to implement optimally.

A particularly interesting bottom-up parser is the CYK algorithm [6, 7, 15] 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 parsing

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) The worst case time complexity of a recursive descent parser can be exponential in the length of the input.
(2) Left-recursive grammars cause naive implementations of recursive descent parsing to get stuck in infinite recursion, which eventually causes a 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 past 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 between and after any syntax errors that are encountered during parsing.

These problems have gradually been addressed over the years:

(1) The discovery of *packrat parsing*, employing memoization to avoid duplication of work [22], reduced parsing time 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) Left recursive rules were found to be rewritable as *indirectly* left recursive [11], although the rewriting rules can be extremely complex, especially when semantic actions are involved – or as right-recursive, with only a weak equivalence to the left recursive form. With extension, packrat parsing can be made to handle left recursion directly [11, 13, 14, 26], although usually with loss of linear time parsing, which is one of the most important reasons packrat parsing is chosen over other parsing algorithms [26].
(3) Progress has been made on 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). This is far from a solved problem.

1.4 PEG grammars

A significant refinement of recursive descent parsing known as Parsing Expression Grammars (PEG) was proposed by Ford in his 2002 PhD thesis [11]. PEG rules replace ambiguity with deterministic choice, i.e. all PEG grammars are unambiguous. There are only a handful of PEG rule types (see Section 1.4), and they can be implemented in a straightforward way. All non-left-recursive PEG grammars can be parsed directly by recursive descent, and they can be parsed efficiently with a packrat parser, i.e. using memoization. PEG grammars are more powerful than regular expressions, but it is conjectured (and as yet unproved) that there exist context-free languages that cannot be recognized by a PEG parser [12]). As PEG grammars are a subset of all grammars that can be parsed with recursive descent, PEG packrat 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.

In contrast with most bottom-up parsers, which represent grammar rules as generative productions, PEG parsers represent grammar rules as *greedily-recognized patterns*. A PEG grammar consists of:

- A finite set N of *rule names*.
- A finite set Σ of *terminal symbols* that is disjoint from N. These could match individual characters of the input (`'+'`), words (`"import"`), or even a more complex terminal pattern recognized by a regular expression or similar, creating a hybrid parser.
- A *starting expression* e_s. This corresponds to the root node of the Abstract Syntax Tree (AST) after a successful parse.
- A finite set P of *parsing rules*. Each rule in P has the form $A \leftarrow e$, where A is a unique rule name and e is a *parsing expression* or *clause*.

A parsing expression or clause is a hierarchical expression consisting of:

- Terminal symbols.
- Rule names (as a reference to the rule of that name) – this corresponds to a recursive call in top-down parsing.
- The *empty string* ϵ (with ASCII notation `()`). This always matches, even at the end of input, and consumes zero characters.
- a hierarchy of *PEG operators*, with parameters selected from the three items listed above. Each PEG operator and its parameters forms a clause. The parameters of the PEG operator are termed *subclauses*. Parentheses may be used to control precedence of nested operators.

Given parsing expressions e and e_i, a new parsing expression can be constructed using the following PEG operators:

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

`Optional` and `ZeroOrMore` can either be implemented directly, or automatically transformed into the equivalent forms shown above.

The match logic for these PEG operators is as follows:

- A `Seq` clause matches if all of its subclauses match the input, starting at the current parsing position. Each matching subclause consumes zero or more characters, which determines the starting parse position for the match attempt of the subsequent subclause. Matching stops once a single clause fails to match the input at its start position.
- A `First` clause matches if any of its subclauses match, 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.
- 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 the match consumes zero characters.
- A `OneOrMore` clause matches if its subclause matches at least once, greedily consuming as many matches as possible.
- A `ZeroOrMore` clause matches if its subclause matches at least once, greedily consuming as many matches as possible, but if its subclause does not match, the `ZeroOrMore` still matches the input (i.e. this operator always matches), but consumes zero characters.
- A `FollowedBy` clause matches if its subclause matches at the current parsing position, but it does not consume any characters if it matches.
- 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.
- `Longest` is a new PEG operator we propose as a way to simplify grammar-writing. 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 are not in the correct order. 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. 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 inputs, and then the 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 these sequences. If it is not easy or possible to determine this partial ordering of prefix matches of all subclauses across all inputs, then a `Longest` operator may be able to be used as a shortcut to obtain the desired parsing behavior. `Longest` will tend to match the grammar writer's intent when using a `First` operator, since the PEG philosophy is one of greedily matching as many characters in the input as possible with each grammar rule.

2 THE PIKA PARSER: BOTTOM-UP PACKRAT PARSING

In this work we introduce the *pika parser*[1], a new type of packrat parser that works bottom-up using dynamic programming, rather than top-down using basic memoization as with standard packrat parsing.

2.1 Advantages of pika parsing

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

(1) As a near-semantically-equivalent "inversion" of top-down packrat parsing, pika parsing has roughly the same performance as packrat parsing, i.e. is roughly linear in the size of the input and the depth of the parse tree, usually within a moderate constant factor.
(2) Bottom-up parsing directly handles left-recursive grammars and grammars containing arbitrary numbers of cycles, which can greatly simplify grammar design, and eliminates the need for grammar rewriting steps.
(3) Bottom-up parsing conveys near-perfect error recovery capabilities on the parser, since all parsing depends only on the immediate surrounding context in the input. In fact it is straightforward to recover parsing at any level of the grammar, by skipping ahead from a syntax error to the next valid match of a grammar rule of interest. This will be particularly helpful in providing code assist features for IDEs that work robustly in spite of syntax errors.
(4) We define the pika parser to work with PEG grammars, which are unambiguous and deterministic, eliminating ambiguity and nondeterminism problems that affect CFG parsers.
(5) The dynamic programming algorithm employed by the pika parser enables parsing to be parallelized across the input, or more accurately, to be parallelized across a dynamic programming "wavefront" which initially consists of all terminals in the input. This allows multiple parts of the input to be parsed simultaneously, splitting the work across all available cores for most of the parsing process. With the right data structures, this parallelization is very straightforward.

Note however that in the general case, advantages (1) and (2) may be be mutually exclusive due to the greediness of PEG grammar rules (advantage (4)) – see Section 3. There is a simple way to rewrite grammar rules by hand to avoid this tradeoff while keeping linear performance – see Section 3.3.

2.2 High-level overview of pika parsing

The premise of pika parsing is very simple: take the recursive structure of a packrat parser's parse of a given input with a given grammar, and invert the top-down recursive structure so that parsing happens bottom-up from terminals towards the root. Instead of shallower frames of recursion recursing down into deeper frames of recursion, as with recursive descent parsing, bottom-up parsing has deeper frames of recursion trigger the execution of shallower frames

[1] A pika (/ˈpaɪkə/) 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.

of recursion. This seems like a strange concept, until one has the realization that this is exactly the definition of dynamic programming.

In most dynamic programming (DP) algorithms, there is an underlying recurrence, which is represented as lookups into earlier, already-computed parts of the DP table. These are directly analogous to recursive calls in recursive descent parsing. As the DP algorithm proceeds, it follows these dependency arrows `backwards` from leaves (usually the top and left edge of the table, if the recurrence is two-dimensional) towards the root (represented by the bottom right cell of the table). Because the recurrence is the same in every cell, figuring out how to reverse these dependency arrows is very simple – but the problem is further simplified since all DP table cells are always evaluated, so you only have to find an evaluation order so that all the previous values needed for an evaluation of the recurrence relation are always available. For a DP algorithm like the string edit distance algorithm [19], the recurrence relation for cell $\langle i, j \rangle$ depends upon cells $\langle i-1, j \rangle$, $\langle i, j-1 \rangle$ and $\langle i-1, j-1 \rangle$, so it is possible to populate the DP table in row major order, column major order, or using a trailing diagonal "wavefront" that sweeps from upper left to lower right through the table. Any of these orders ensures that a "lower" cell in the recurrence is always evaluated before a "higher" cell in the recurrence.

Inversion of packrat parsing applies exactly the same idea as dynamic programming, except that the structure of the recurrence relations (i.e. the recursive calls) between the cells are now dynamic rather than static. For a `Seq` clause like `P <- A B C`, the starting position at which subclause `B` will be matched depends upon the number of characters consumed by the match of subclause `A`, and the starting position of subclause `C` depends upon how many characters are consumed by both `A` and `B`. These dynamic relationships are recorded by storing `backRefs` or back-references in the memo table entries, which record when a parent clause tried to match a subclause at a given position.

In other words, to produce a bottom-up "inverted recursion" when there is a dynamic dependency relationship between parent clauses and subclauses, each step in the bottom-up parse actually performs a single top-down step from recursion depth i to recursion depth $i + 1$, in order to check the contents of the memo table for subclause matches at a deeper level of recursion. Then back-references are recorded from that deeper level of recursion, so that the parent clause can be found in the future if the subclause match is later improved.

Bottom-up parsing is probably best explained with a concrete example.

2.3 Example of pika parsing

Listing 1 shows a simple "sum of products" grammar. The most complex expression this could recognize is a sum of two products. The lowest precedence rule `S` (i.e. Sum) is listed as the first (root) rule, so the root of the parse tree will be an `S` node if the parse is successful. The `S` rule is comprised of two `P` rules (i.e. Product), which are higher precedence. The `P` rule contains two clauses: an outer `First` clause and (in the first position) a nested `Seq` clause, `(N X N)`. The `N` rule recognizes one or more digits. The `A` and `X` rules recognize arithmetic operators.

```
1  Grammar:
2    S <- P A P;
3    A <- '+' / '-';
4    P <- (N X N) / N;
5    N <- ([0-9])+;
6    X <- '*' / '/';
7  Input:
8    1+2*3
```

Listing 1. Example grammar and input string

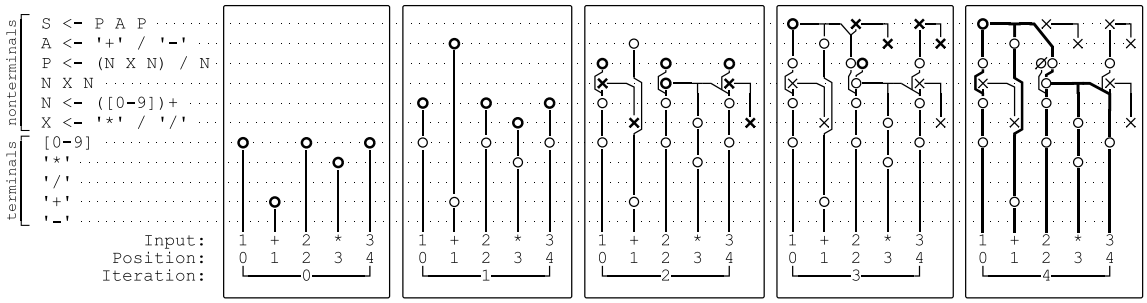

Fig. 1. Example parse of Listing 1

Iteration	`activeSet` entry	Match(○)/Mismatch(×)	Match len	Parent memo keys
0	⟨`[0-9]`, 0⟩	○	1	⟨N, 0⟩
0	⟨`'+'`, 1⟩	○	1	⟨A, 1⟩
0	⟨`[0-9]`, 2⟩	○	1	⟨N, 2⟩
0	⟨`'*'`, 3⟩	○	1	⟨X, 3⟩
0	⟨`[0-9]`, 4⟩	○	1	⟨N, 4⟩
1	⟨N, 0⟩	○	1	⟨`N X N`, 0⟩, ⟨P, 0⟩
1	⟨A, 1⟩	○	1	
1	⟨N, 2⟩	○	1	⟨`N X N`, 2⟩, ⟨P, 2⟩
1	⟨X, 3⟩	○	1	
1	⟨N, 4⟩	○	1	⟨`N X N`, 4⟩, ⟨P, 4⟩
2	⟨`N X N`, 0⟩	×	–	
2	⟨P, 0⟩	○	1	⟨S, 0⟩
2	⟨`N X N`, 2⟩	○	3	⟨P, 2⟩
2	⟨P, 2⟩	○	1	⟨S, 2⟩
2	⟨`N X N`, 4⟩	×	–	
2	⟨P, 4⟩	○	1	⟨S, 4⟩
3	⟨S, 0⟩	○	3	
3	⟨P, 2⟩	○	3	⟨S, 0⟩
3	⟨S, 2⟩	×	–	
3	⟨S, 4⟩	×	–	
4	⟨S, 0⟩	○	5	

Table 1. Memo table entries created in an example pika parse of Listing 1.

Fig. 1 and Table 1 show how pika parsing proceeds with this example. Each of the five boxes in Fig. 1 depicts the content of the memo table at the end of one step of parsing. The grammar rules are shown on the left hand side. Note that the rule `P` contains the nested clause `(N X N)`, the clause `N` contains the nested terminal `[0-9]`, and the clauses `A` and `X` contain nested terminals for the arithmetic operators, and each of these nested clauses are each given their own row in the memo table (this is so that duplicate work is avoided in more complex grammars, where sub-clauses may be shared between multiple rules). In Fig. 1, matches in the memo table are shown by the symbol ○ and mismatches are shown by the symbol × (note that the use of × to indicate mismatch is unrelated to the rule with the name `X`). These symbols are bolded if they are new matches added to the memo table at the end of the iteration.

A *memo key* or index into the memo table has the form $\langle$`clause`, $p\rangle$, where `clause` designates the row of the memo table and `p` designates the column (or the start position within the input of a match of the given clause).

Between iteration i and iteration $i+1$, any new matches or mismatches added to the memo table (marked in bold) must trigger the evaluation of all applicable *parent memo keys* to be evaluated in the next iteration. This is the mechanism by which the top-down PEG parsing algorithm is inverted and turned into a bottom-up algorithm. These parent memo keys are added to a set, `activeSet`, which is the upwards-expanding wavefront of the dynamic programming algorithm (also known sometimes as the "dirty set" when describing dynamic programming algorithms). Since `activeSet` elements are independent of each other (they all depend only on the state of the memo table at the end of the previous iteration, not upon each other), `activeSet` elements can all be processed in parallel.

The first mechanism for identifying parent memo keys is termed `seedParentClauses`. Given the rules `S <- P A P` and `P <- (N X N) / N`, rule `S` is a seed parent clause of rule `P`, since in top-down evaluation, an attempt to match rule `S` at position 0 would first try to match subclause `P` at position 0. Therefore, in bottom-up parsing, if the memo table entry with the memo key $\langle$`P`, $p\rangle$ is updated in iteration i for some position p, then $\langle$`S`, $p\rangle$ must be scheduled for matching in iteration $i+1$. The entries in `seedParentClauses` are determined for each clause and subclause at initialization, since the grammar does not change during parsing (this `seedParentClauses` is static). The parent memo key consists of the seed parent clause of the subclause, paired with the start position p of the subclause match's memo key.

The second mechanism for identifying parent memo keys is termed `backRefs`, and is dynamic, in the sense that the start position of the parent memo key (stored in `backRefs` within a memo entry) depends upon how many characters the parent clause consumed before it tried to match the subclause in question. When any parent clause attempts to match the input at a given start position, for each subclause that the parent clause tries to match by looking up the subclause at the current parsing position in the memo table, a back reference memo key is stored in the subclause's memo entry at that position, linking back to the parent clause and the parent clause's start position (whether or not the subclause actually matched at that start position). If the subclause match changes in a subsequent iteration (due to identifying a better match for the subclause at this start position), then any back-referenced parent clauses are triggered for re-evaluation.

The update steps for the memo table are described below.

2.3.1 Iteration 0. The memo table is initially seeded with all matches of each terminal at some position of the input, requiring $O(N|T|)$ time for input length N, and given $|T|$ terminals. (Optimizations of this preprocessing step will be discussed later.) This yields the following memo table entries. Note that parent memo keys produced from each `activeSet` entry in iteration i become the `activeSet` entries for iteration $i+1$.

2.3.2 Iteration 1. Each parent memo key in the `activeSet` again matches a single character at a higher clause in the grammar, triggering their own parent clauses for evaluation in Iteration 2. However, $\langle$`A`, 1$\rangle$ does not have any `seedParentClauses`, because it is the second clause of `S <- P A P`, and the first clause consumes at least one character (similarly $\langle$`X`, 3$\rangle$ does not have any `seedParentClauses`).

2.3.3 Iteration 2. Each of the new active set entries $\langle$`P`, 0$\rangle$, $\langle$`P`, 2$\rangle$ and $\langle$`P`, 4$\rangle$ generated by iteration 1 results in an attempt to match `P` at the corresponding position. Since `P` is a `First` clause, this results in first testing whether the first subclause of `P <- (N X N) / N`, i.e. `N X N`, matches at position $p \in 0, 2, 4$. As always, this is achieved by looking at *the state of the memo table at the end of the previous iteration.* The memo table has no entries for `N X N` at the end of iteration 1, so all three of these subclause match attempts fail, and then the the second subclause `N` of rule `P` is tested. In all three

cases, there is a positive match entry in the memo table at the end of iteration 1; therefore, ⟨P, p⟩ is recorded as a match in the memo table for all three values of p at the end of iteration 2, and the parent memo entries ⟨S, p⟩ are triggered by adding them to `activeSet` for the next iteration.

However, in spite of the fact that ⟨`N X N`, p⟩ does not match in the memo table for any of $p \in 0, 2, 4$ at the end of iteration 1, when the clause ⟨P, p⟩ looks up those subclause memo entries, a `backRef` is added to each subclause memo entry, pointing back to ⟨P, p⟩. If the match state of any of the ⟨`N X N`, p⟩ entries subsequently changes from a mismatch to a match, or from a shorter match to a longer match, the corresponding parent clause ⟨P, p⟩ will be triggered in the following iteration.

For `activeSet` entries ⟨`N X N`, 0⟩ and ⟨`N X N`, 4⟩, the first subclause `N` matches (as determined by looking up ⟨N, 0⟩ and ⟨N, 4⟩ respectively in the memo table in the state it was in at the end of iteration 1). In both cases, the subclause match consumes one character, so when the second subclause `X` is matched, it starts matching at the following character position (1 and 5 respectively). However, ⟨X, 1⟩ does not match, and ⟨N, 5⟩ is past the end of the input. Entries are added to `backRefs` for these mismatching entries, pointing back to their respective parent clauses (in this case, the subclause match status will never change, so these back references will never be followed).

A complete match is found for `activeSet` entry ⟨`N X N`, 2⟩, based on three entries in the memo table at the end of iteration 1: ⟨N, 2⟩, ⟨X, 3⟩, and ⟨N, 4⟩, consuming a total of three characters, matching the last three characters of the input string, `"2*3"`. This match causes the parent memo key ⟨P, 2⟩ to be added to `activeSet` for the next iteration.

2.3.4 Iteration 3. The `activeSet` entries ⟨S, 2⟩ and ⟨S, 4⟩ fail to match, because they fail to find a match of rule `A` in their second subclause (input position 3 and 5 respectively). However, ⟨S, `0`⟩ succeeds in matching the first three characters, i.e. `1 + 2`, via the entries in the memo table at the end of iteration 2: ⟨P, `0`⟩, ⟨A, 1⟩, and ⟨P, 2⟩. Note that this match does not respect precedence order, and consequently it will be discarded in the next iteration and replaced with a better match. Rule `S` does not have any parent clauses, so this match does not cause any new memo keys to be written to `activeSet`.

The `activeSet` entry ⟨P, 2⟩, i.e. the rule `P <- (N X N) / N`, checks the memo table for a match of its first subclause, ⟨`N X N`, 2⟩, and finds a match in that position, consuming three characters. Therefore, ⟨P, 2⟩ is a new match in iteration 3, also consuming 3 characters.

Note that ⟨P, 2⟩ matched only one character at the end of iteration 2 (the previous best match), but now matches 3 characters at the end of iteration 3 (the new best match). This is shown as two adjacent ∘ characters in the memo table at this position – the older match is shown to the left, non-bolded, and the newer match is shown to the right, bolded. Because memo table values are always looked up in the state the memo table was in at the end of the previous iteration, the match for ⟨S, `0`⟩ actually depends upon the older match obtained in iteration 2, not upon the new best match found in the iteration 3. The previous best match will be overwritten by the new best batch at the end of iteration 3 (shown in iteration 4 by putting a slash through the older match). Replacing the best match in this memo entry will trigger the parent clause ⟨S, `0`⟩ to be re-evaluated in iteration 4, via the `backRefs` mechanism – hence this is parent memo key generated by ⟨P, 2⟩ in this iteration.

2.3.5 Iteration 4. The single ⟨S, `0`⟩ is re-evaluated, and now for its third subclause, picks up the updated match for ⟨P, 2⟩ from the memo table in the state it was in at the end of iteration 3. The final subclause matches are `P = "1"`, `A = "+"`, and `P = "2*3"`, corresponding to the parse `"1+(2*3)"`. The final parse tree is shown with bolded lines.

At this point `activeSet` is empty, because the clause of the only active memo entry ⟨S, `0`⟩ has no parent clauses, so the parse is complete.

2.3.6 Efficiency Analysis. The final parse tree, constructed bottom-up, has the same structure as the parse tree that would have been obtained by top-down parsing of the same input with the same grammar[2].

There are a number of entries in the memo table at the end of the parse that are unused (meaning they are not part of the final parse tree rooted at ⟨S, 0⟩), such as mismatching entries and entries that were eagerly triggered by bottom-up parsing that did not end up in the final parse tree. Two entries in the memo table, ⟨S, 0⟩ and ⟨P, 2⟩, started with a worse match, and the match was replaced in a later iteration with a better match. (The logic used to determine whether a match is better or worse is given later.) In each of these cases, unnecessary work was done by the bottom-up pika parser, compared to a top-down packrat parser.

The overhead of wasted work performed by bottom-up parsing is not enormous – it should amount to only a moderately small constant factor. This is the price paid for the advantages listed in Section 2.1, and the performance overhead of bottom-up parsing can me mitigated by parallelization of the parser.

Not all the unused nodes left over in the memo table at the end of parsing are useless: in particular, in the case of an incomplete parse due to a syntax error, all matches for any desired grammar rule can be found by simply reading off the entries in the row of the memo table corresponding to the rule. Thus an IDE or compiler can very easily recover after a syntax error at any desired level of grammar, for example by finding the start of the next valid expression, statement, or function.

Additionally, the replacing of an earlier shorter match with a later longer match in the same memo entry is not always wasteful of work, because the earlier shorter match may be incorporated into the parse tree as a descendant of the later longer match, as in the case of a grammar that contains cycles, such as an expression rule that can match of a self-recursive tree of sub-expressions of mutually nested types. The matches stored in the memo table are actually connected to their subclause matches, i.e. they form nodes in the cumulatively built structure of the parse tree.

2.3.7 Preprocessing optimization: lexing. The preprocessing step of initializing the memo table with all matches of all terminals against the input can be optimized by running a lex (or lexing) preprocessing step, which is a shallow acyclic top-down preprocessing parse of the input stream to break it into complete tokens. This can reduce the number of memo entries that the memo table is seeded with.

Lexing serves another important purpose: the PEG operator `OneOrMore` greedily matches from left to right, consuming as many matches of its single subclause as possible. Given the rules `Word <- [A-Z]+` and `Num <- [0-9]+` that matches a sequence of one or more letters or digits respectively, and given the input `CAT123`, the bottom-up parsing algorithm would start by seeding the memo table with the matches for the terminal clauses `[A-Z]` and `[]0-9]`, i.e. {⟨`'C'`, 0⟩, ⟨`'A'`, 1⟩, ⟨`'T'`, 2⟩, ⟨`'1'`, 3⟩, ⟨`'2'`, 4⟩, ⟨`'3'`, 5⟩}. Each of the letter matches would seed the `[0-9]+` parent clause and each of the digit matches would seed the `[0-9]+` parent clause. Therefore, the first step of bottom-up parsing would produce the memo table matches {⟨`"CAT"`, 0⟩, ⟨`"AT"`, 1⟩, ⟨`'T'`, 2⟩, ⟨`"123"`, 3⟩, ⟨`"23"`, 4⟩, ⟨`'3'`, 5⟩}, in other words all suffix strings of every `OneOrMore` token is added to the memo table. This can cascade up the memo table until all the suffixes are finally discarded by a larger rule combining matches from multiple adjacent rules, e.g. `WordNum <- Whitespace? Word Num Whitespace?`, which selects only the first of each series of suffix matches.

An even more important use of lexing is to deal with quoted strings and comments. Given that bottom-up parsing starts with no context, a language consisting of tokens, quoted strings and comments may incorrectly assume that a word inside a quoted string or comment is a token. In general, the contents of a quoted string or comment can produce

[2]This is not quite true, especially considering that left-recursive grammars can be parsed bottom-up without any problem, but top-down recursion of the same grammar can get stuck in an infinite loop. However, assuming a top-down parse succeeds, the parse tree obtained by bottom-up parse should be structurally the same as the parse tree obtained by the top-down parse.

numerous unnecessary memo table entries in the lowel levels of parsing. These unnecessary entries will eventually be ignored as the structure of the document starts to emerge at higher and higher levels of parsing, as more and more adjacent context is joined. However, it is wasteful to store these unnecessary matches in the memo table. If the lex rule extracts tokens, quoted strings and comments in a preprocessing parse, and if these lexed tokens are used to seed the bottom-up parse rather than character-level terminals, this problem is solved.

In the reference parser (Section 5), lexing is enabled by adding a rule called `Lex` that matches all terminals (or tokens, quoted strings, comments, etc.) using a single `First` clause. This rule is repeatedly applied until all input is consumed. 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 should end up in the final parse tree, otherwise the appropriate parent clauses will not be triggered for subsequent bottom-up parsing.

Note that top-down parsing of the `Lex` rule in the reference parser implements non-memoized recursive descent parsing (to avoid creating unnecessary memo entries, in order to keep the memo table small), so the tokens matched by the `Lex` rule should be short, and each clause of the `Lex` rule should be simple, in order to keep lex preprocessing running in close to linear time.

2.4 Implementation details

The key classes of the reference pika parser are as follows[3]. The code shown is simplified for to illustrate the most important aspects of the parsing algorithm and data structures. The full code for this class can be viewed in the reference parser implementation (see Section 5).

2.4.1 The `MemoKey` *class.* This class is a simple record type holding the two-dimensional coordinate of a memo table entry, i.e. ⟨`clause`, `startPos`⟩.

```
class MemoKey {
    Clause clause;
    int startPos;
}
```

Listing 2. The `MemoKey` class

2.4.2 The `Parser` *class.* This class accepts a `Grammar` in its constructor (not shown). A `Grammar` is a collection of `Rule` objects, and also has an `allClauses` field, which is a set of all unique clauses and sub-clauses across all rules. The main `parse` method of the `Parser` class first creates two sets, `activeSet` (which holds the memo keys constituting the current DP wavefront), and `updatedEntries`, which holds the set of memo entries for which new matches were found in the current iteration, then initializes the memo table, by populating the memo table with the locations that all terminals match the input. (Note that the optional lex preprocessing step is not shown here, since it is only an optimization.)

(Note the use of the Java construction `collection.parallelStream().forEach(lambda)` here, which processes all elements of `collection` in parallel by calling the function `lambda` on each element. Both the initialization step and

[3]Listings are given in Java-like notation, however some Java-specific details and quirks have been simplified or omitted, for example all data structures that may be written from multiple threads are prefixed by `Concurrent` for clarity, even if a class of this exact name does not exist in the Java standard libraries. For example, `ConcurrentPriorityQueue` represents the Java standard library class `java.lang.concurrent.PriorityBlockingQueue`, and `ConcurrentSet` represents a concurrent hashset resulting from calling `Collections.newSetFromMap(new ConcurrentHashMap<T, Boolean>())`. Some field initializers (`new` statements) and modifiers such as `public` and `final` have been omitted, as well as trivial assignment constructors.

each stage of the main parsing loop can be item-wise parallelized using this construction, since none of the items in the collections being iterated over have any interdependencies.)

After initialization, the `parse` method enters the main parsing loop. This loop consists of two alternately-run inner loops (each implemented as a parallel `forEach`) that are mutually interdependent upon each other's output. The first loop iterates through and consumes all items in `activeSet`, the current DP wavefront, checking all subclauses of each clause to see if they match in the memo table, and applying the PEG operator for each clause to determine if the clause itself matches based on the subclause match(es). This first loop produces `updatedEntries`, the set of all memo table entries for which a new match was found.

The second inner loop iterates through and consumes all `updatedEntries`, overwriting the previous best match for each updated memo table entry with the new best match, generating the next `activeSet` from the parent clauses of these new best matches. Overwriting the previous best match with the new best match is performed in this second separate step to ensure the memo table does not change while memo entries are still being read for the `activeSet` entries in the first inner loop.

Each element of `activeSet` generates elements in `updatedEntries`, and vice versa, until both sets are empty, at which point the parse is complete, because there are no more parent clauses to trigger, and no more memo table entries to update.

```
public class Parser {
    final Grammar grammar;
    MemoTable memoTable = new MemoTable();

    public void parse(String input) {
        var activeSet = new ConcurrentSet<MemoKey>();
        var updatedEntries = new ConcurrentSet<MemoEntry>();

        // Initialize memo table by matching all terminal clauses against the input at each position.
        // Adds the memo entries for the initial terminal matches to updatedEntries.
        grammar.allClauses.parallelStream().forEach(clause -> {
            if (clause Instanceof Terminal) {
                for (int startPos = 0; startPos < input.length(); startPos++) {
                    var memoKey = new MemoKey(clause, startPos);
                    // Add a match to the memo table if the terminal matches
                    clause.match(MatchDirection.TOP_DOWN, memoTable, memoKey, input, updatedEntries);
                }
            }
        });

        // Main parsing loop
        while (!activeSet.isEmpty() || !updatedEntries.isEmpty()) {
            // Test match status of each memo key in the active set, producing updatedEntries
            activeSet.parallelStream().forEach(memoKey -> {
                memoKey.clause.match(MatchDirection.BOTTOM_UP, memoTable, memoKey, input,
                        updatedEntries);
            });
            activeSet.clear();

            // Update the best match for all updatedEntries, populating activeSet with parent clauses
            updatedEntries.parallelStream().forEach(memoEntry -> {
                memoEntry.updateBestMatch(input, activeSet);
            });
            updatedEntries.clear();
        }
    }
}
```

Listing 3. The `Parser` class

2.4.3 The `Clause` *class.* This class represents a PEG operator or clause and its `subClauses`. This class is abstract, and is subclassed for each PEG operator type (the subclasses implement the `match` method).

The `Clause.seedParentClauses` field records which parent clauses try to match this subclause at the parent clause's start position (i.e. before the parent clause has consumed any previous characters). Like `MemoEntry.backRefs`, `Clause.seedParentClauses` triggers parent clauses to be checked for a match when the match for one of their sub-clauses is updated. These two back-reference mechanisms are collectively used to turn top-down recursion into bottom-up recursion. Both are needed, since `MemoEntry.backRefs` stores back-references to parent clauses no matter how many previous characters the parent clause previously consumed, whereas `Clause.seedParentClauses` is needed to trigger the evaluation of parent clauses that have not yet been evaluated for a match (and therefore have not stored back references in their subclauses' memo entries).

The `canMatchZeroChars` field is used to record whether a clause can match while consuming zero characters, such as ϵ, `X?`, `X*`, or `X / Y / Z?`. (Clauses that can match zero characters always match at every input position, consuming zero or more characters, which slightly changes how they should be handled, as will be described later.)

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 lexing if `matchDirection == TOP_DOWN`.

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

    abstract Match match(MemoTable memoTable, MemoKey memoKey, String input,
            Set<MemoEntry> updatedEntries);
}
```

Listing 4. The `Clause` class

2.4.4 The `Char` class. This class is the simplest example of a PEG terminal clause. `Char` extends `Terminal`, and `Terminal` extends `Clause`. The `Terminal` class is used to detect which clauses are terminals using the `instanceof` operator.

The `Char` class simply attempts to match the character it is initialized with at the location provided by `memoKey.startPos`. If the character matches, it stores a one-character match in the memo table at the entry indexed by `memoKey`, by calling `memoTable.addTerminalMatch`.

```
class Char extends Terminal {
    char c;

    Char(char c) {
        this.c = c;
    }

    @Override
    Match match(MatchDirection matchDirection, MemoTable memoTable, MemoKey memoKey, String input,
            Set<MemoEntry> updatedEntries) {
        return memoKey.startPos < input.length() && input.charAt(memoKey.startPos) == c
            ? memoTable.addTerminalMatch(memoKey, /* terminalLen = */ 1, updatedEntries)
            : null;
    }
}
```

Listing 5. The implementation of the `Char` terminal

2.4.5 Non-terminal PEG operator classes. The `First` subclass of `Clause` implements the `First` PEG operator. The `match` method iterates through each subclause of the `First` clause, and returns a new `Match` object for the first subclause that matches, or `null` if no match was found. In contrast, the `Seq` subclass' `match` method requires all subclauses to match in order for the `Seq` clause to match, and the match start position must be updated for each subclause as characters are consumed.

If matching top-down (i.e. during lexing), the `match` method recursively calls the `match` method of the subclause. When matching bottom-up, the `match` method looks for subclause matches in the memo table by calling `memoTable.lookUpMemo` method, but does not recurse to the subclauses' `match` methods.

The other subclasses of `Clause` (implementing the other PEG operator types) can be found in the reference implementation (see Section 5).

```
class First extends Clause {
    @Override
    Match match(MatchDirection matchDirection, MemoTable memoTable, MemoKey memoKey,
            String input, Set<MemoEntry> updatedEntries) {
        for (int subClauseIdx = 0; subClauseIdx < subClauses.length; subClauseIdx++) {
            var subClause = subClauses[subClauseIdx];
            var subClauseMemoKey = new MemoKey(subClause, memoKey.startPos);
            var subClauseMatch = matchDirection == MatchDirection.TOP_DOWN
                    ? subClause.match(MatchDirection.TOP_DOWN, memoTable, subClauseMemoKey,
                              input, updatedEntries)
                    : memoTable.lookUpMemo(subClauseMemoKey, input, memoKey, updatedEntries);
            if (subClauseMatch != null) {
                return memoTable.addNonTerminalMatch(memoKey,
                        /* firstMatchingSubClauseIdx = */ subClauseIdx,
                        new Match[] { subClauseMatch }, updatedEntries);
            }
        }
        // No match
        return null;
    }
}
```

Listing 6. The implementation of the `First` PEG operator

```
class Seq extends Clause {
    @Override
    Match match(MatchDirection matchDirection, MemoTable memoTable, MemoKey memoKey, String input,
            Set<MemoEntry> updatedEntries) {
        var subClauseMatches = new Match[subClauses.length];
        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
                    ? subClause.match(MatchDirection.TOP_DOWN, memoTable, subClauseMemoKey,
                              input, updatedEntries)
                    : memoTable.lookUpMemo(subClauseMemoKey, input, memoKey, updatedEntries);
            if (subClauseMatch == null) {
                // If a subclause fails to match, the Seq clause does not match
                return null;
            }
            subClauseMatches[subClauseIdx] = subClauseMatch;
            currStartPos += subClauseMatch.len;
        }
        return memoTable.addNonTerminalMatch(memoKey, /* firstMatchingSubClauseIdx = */ 0,
                subClauseMatches, updatedEntries);
    }
}
```

Listing 7. The implementation of the `Seq` PEG operator

2.4.6 The `MemoEntry` *class.* This class stores the current best match, and a priority queue of new best matches that were found in the current iteration. The `addNewBestMatch` method adds new matches during the first inner loop of the main parse loop, when `activeSet` is being iterated over. Importantly, if a new match is not better than the current best match (i.e. if `newMatch.compareTo(bestMatch) >= 0`, see the explanation of the `Match.compareTo` method below), then the new match is never written to the `newMatches` priority queue. This ensures that every new best match *monotonically* improves the match, so that the parse is guaranteed to terminate.

The `updateBestMatch` method is called by the second inner loop of the main parse loop, when `updatedEntries` is being iterated over. The `updateBestMatch` method finds the best new match, by removing the head of the `newMatches` priority queue, then setting `bestMatch` to this new best match. Then all the seed parent clauses of the clause of the new best match and all the backrefs from the memo entry are used to generate parent memo keys that are added to the `activeSet` for the next iteration.

```
class MemoEntry {
    MemoKey memoKey;
    Match bestMatch;
    ConcurrentPriorityQueue<Match> newMatches = new ConcurrentPriorityQueue<>();
    ConcurrentSet<MemoKey> backRefs = new ConcurrentSet<>();

    MemoEntry(MemoKey memoKey) {
        this.memoKey = memoKey;
    }

    void addNewBestMatch(Match newMatch, Set<MemoEntry> updatedEntries) {
        // If the new match is better than the current best match from the previous iteration
        if (bestMatch == null || newMatch.compareTo(bestMatch) < 0) {
            // Add the new match to the priority queue, and mark the entry as updated
            newMatches.add(newMatch);
            updatedEntries.add(this);
        }
    }

    void updateBestMatch(String input, Set<MemoKey> activeSetOut) {
        // Get the best new updated match for this MemoEntry, if there is one
        var newBestMatch = newMatches.peek();
        if (newBestMatch != null) {
            newMatches.clear();
            // Replace previous best match with new best match
            bestMatch = newBestMatch;
            // Generate new activeSet entries from seed parent clauses and backrefs
            for (var seedParentClause : memoKey.clause.seedParentClauses) {
                MemoKey parentMemoKey = new MemoKey(seedParentClause, bestMatch.memoKey.startPos);
                activeSetOut.add(parentMemoKey);
            }
            for (var backref : backRefs) {
                activeSetOut.add(backref);
            }
            // Clear backrefs
            backRefs.clear();
        }
    }
}
```

Listing 8. The `MemoEntry` class

2.4.7 The `Match` *class.* This class is used to store matches and their lengths in memo entries. `Match` instances are nodes in the parse tree, connected to their child nodes by their `subClauseMatches` field. The most important method in `Match` is `compareTo`, which is used to determine when one match is better than another. This comparator method is used by the `MemoEntry.newMatches` priority queue so that the best new match is always at the head of the queue, and also to ensure the monotonic improvements of memo entry updates, as described above.

According to the `Comparable` contract, the comparison `match1.compareTo(match2)` should return a negative number if `match1` should sort earlier than `match2` (in this context, if `match1` is a better match than `match2`, zero if the two matches are equal, and a positive number if `match1` should sort later than `match2` (in this context, if `match1` is a worse match than `match2`. This `compareTo` method is designed to as closely as possible replicate the greedy left-to-right semantics of the PEG operators as they are defined for top-down parsing.

```
class Match implements Comparable<Match> {
    MemoKey memoKey;
    int len;
    Match[] subClauseMatches;
    int firstMatchingSubClauseIdx;

    @Override
    public int compareTo(Match o) {
        if (o == this) {
            return 0;
        }
        // An earlier subclause match in a First clause wins over a later match
        if (this.memoKey.clause instanceof First) {
            var diff0 = this.firstMatchingSubClauseIdx - o.firstMatchingSubClauseIdx;
            if (diff0 != 0) {
                return diff0;
            }
        }
        // A longer match wins over a shorter match
        var diff1 = o.len - this.len;
        if (diff1 != 0) {
            return diff1;
        }
        var minSubclauses = Math.min(this.subClauseMatches.length, o.subClauseMatches.length);
        for (int i = 0; i < minSubclauses; i++) {
            var diff2 = o.subClauseMatches[i].len - this.subClauseMatches[i].len;
            if (diff2 != 0) {
                return diff2;
            }
        }
        // A longer list of OneOrMore subclause matches wins over a shorter list
        return o.subClauseMatches.length - this.subClauseMatches.length;
    }
}
```

Listing 9. The `Match` class

2.4.8 The `MemoTable` *class.* This class contains a double-level map, `memoTable`, mapping a clause to a start position to a `MemoEntry`. This nesting of maps, and the use of `ConcurrentSkipListMap` for the inner map implementation, allows all matches anywhere in the input to be quickly enumerated for any clause, in order of start position. This is useful for error recovery and other purposes.

The `getOrCreateMemoEntry` method returns the memo entry for a given memo key, or if a memo entry doesn't yet exist for that memo key, creates a new memo entry for that memo key and returns the new memo entry. (The implementation shown below uses `computeIfAbsent`, which is the Java-specific mechanism for creating `Map` entries in a lock-free way while avoiding calling `new` to create a new entry where possible. It is guaranteed to implement singleton allocation semantics in this case.)

The `lookupMemo` entry simply looks up a memo key in the memo table, and returns the current best match (for the state the memo table was in at the end of the previous iteration). However, it also adds the memo key of the parent clause (the clause that called this method to check its subclause matches) to the `backRefs` field of the subclause's memo

entry. This dynamically provides for the parent clause to be triggered for re-evaluation if the memo entry's best match is ever updated.

The `addMatch` method (with variants for nonterminals and terminals) creates a new `Match` object, linking a match to its subclause matches, and then calls `MemoEntry.addNewBestMatch` to add the new match to the priority queue of new matches generated in the current iteration. The best of these matches will overwrite the memo entry's current `bestMatch` value at the end of the iteration, if it is an improvement over the previous best match.

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

    // Get the memo entry for memoKey if one is already present; if not, create and return a new entry
    private MemoEntry getOrCreateMemoEntry(MemoKey memoKey) {
        var skipList = memoTable.computeIfAbsent(memoKey.clause, c -> new ConcurrentSkipListMap<>());
        var memoEntry = skipList.computeIfAbsent(memoKey.startPos, s -> new MemoEntry(memoKey));
        return memoEntry;
    }

    Match lookUpMemo(MemoKey memoKey, String input, MemoKey parentMemoKey,
                Set<MemoEntry> updatedEntries) {
        var memoEntry = getOrCreateMemoEntry(memoKey);
        memoEntry.backRefs.add(parentMemoKey);
        return memoEntry.bestMatch;
    }

    private Match addMatch(MemoKey memoKey, int firstMatchingSubClauseIdx, int len,
            Match[] subClauseMatches, Set<MemoEntry> updatedEntries) {
        var memoEntry = getOrCreateMemoEntry(memoKey);
        var newMatch = new Match(memoKey, firstMatchingSubClauseIdx, len, subClauseMatches);
        memoEntry.addNewBestMatch(newMatch, updatedEntries);
        return newMatch;
    }

    Match addNonTerminalMatch(MemoKey memoKey, int firstMatchingSubClauseIdx,
            Match[] subClauseMatches, Set<MemoEntry> updatedEntries) {
        var totSubClauseMatchLen = 0;
        for (Match subClauseMatch : subClauseMatches) {
            totSubClauseMatchLen += subClauseMatch.len;
        }
        return addMatch(memoKey, firstMatchingSubClauseIdx, totSubClauseMatchLen, subClauseMatches,
                updatedEntries);
    }

    Match addTerminalMatch(MemoKey memoKey, int terminalLen, Set<MemoEntry> updatedEntries) {
        return addMatch(memoKey, /* firstMatchingSubClauseIdx = */ 0, terminalLen, new Match[0],
                updatedEntries);
    }
}
```

Listing 10. The `MemoTable` class

2.4.9 Handling zero-length positive matches. Actually the code shown above for `MemoTable.lookupMemo` is incomplete, because of an interesting corner case, involving positive matches that consume zero characters. The terminal ϵ (or `()` in ASCII representation), which positively matches zero characters everywhere in the input (including beyond the end of

the input string) is the most basic example of this type of match. There are numerous other examples of PEG clauses that can positively match zero characters, such as `Optional`, since the clause `X?` is equivalent to `X / ()`, and if the clause `X` does not match, `()` will always match zero characters.

This can cascade up the grammar from terminals towards the root. For example, a `Seq` clause whose subclauses can all positively match zero characters can itself positively match zero characters. A `First` clause with even a single subclause that can positively match zero characters can itself positively match zero characters. (Furthermore, in the case of `First`, any subclauses after the first subclause that can positively match zero characters will be ignored, because that subclause will always match.)

The PEG operator that matches ϵ, `Nothing`, is a terminal clause, so the memo table could be seeded with positive zero-length matches for $\langle$`Nothing`, $p\rangle$ for all $0 \leq p \leq$ (`input.len`). This will lead to the grammar being parsed correctly bottom-up, since all parent clauses containing a `Nothing` terminal subclause will be properly triggered by seeding the memo table during initialization, just as with all other terminals. However, this adds a large number of unnecessary matches to the memo table, and these matches can cascade upwards through higher-level clauses that can also positively match zero characters, creating many unnecessary memo entries at higher levels of the grammar that will ultimately be ignored because most of them will not be connected to the final parse tree. Therefore, it is better to never memoize `Nothing` matches to avoid polluting the memo table.

Instead, we make the assumption that a rule will never have `Nothing` as its first subclause, which practically speaking is a reasonable assumption. This allows us to only schedule parent clauses for matching if `Nothing`, or any another clause that can positively match zero characters, is looked up in the memo table as the second or subsequent subclause of a clause. We do this by updating the code for `MemoTable.lookupMemo` as shown below. If a memo entry is looked up by a parent clause, and there is no known match for that entry, but the subclause can positively match zero characters, then a zero-length match is added to the memo table for the memo entry, by calling `memoEntry.addNewBestMatch`. This match will trigger the parent clause to be re-evaluated for matching in the next iteration.

Without this refinement, failing to memoize `Nothing` can cause parsing to fail, because the original definition of `lookUpMemo` will return `null` (implying there is no match) for subclauses that can positively match zero characters but have never been memoized before (because they were not triggered bottom-up from the `Nothing` terminal).

This revised method requires the `canMatchZeroChars` field to be set for each clause during initialization. The DAG of all grammar clauses must be topologically ordered, then the initialization code should iterate upwards through the DAG of clauses from terminals towards the root grammar rule, determining which clauses can positively match zero characters.

```
Match lookUpMemo(MemoKey memoKey, String input, MemoKey parentMemoKey,
                 Set<MemoEntry> updatedEntries) {
    var memoEntry = getOrCreateMemoEntry(memoKey);
    memoEntry.backRefs.add(parentMemoKey);
    if (memoEntry.bestMatch != null) {
        return memoEntry.bestMatch;
    } else if (memoKey.clause.canMatchZeroChars) {
        // Find index of first subclause that can match zero characters
        int firstMatchingSubClauseIdx = 0;
        for (int i = 0; i < memoKey.clause.subClauses.length; i++) {
            if (memoKey.clause.subClauses[i].canMatchZeroChars) {
                firstMatchingSubClauseIdx = i;
                break;
            }
        }
        // Create and memoize a new zero-width match. This will trigger the parent clause to be
        // re-evaluated in the next iteration.
        var newMatch = new Match(memoKey, firstMatchingSubClauseIdx, /* len = */ 0, new Match[0]);
        memoEntry.addNewBestMatch(newMatch, updatedEntries);
        return newMatch;
    } else {
        return null;
    }
}
```

Listing 11. Improvement of the `MemoTable.lookupMemo` method to handle zero-length matches without memoizing `Nothing`

3 PRECEDENCE PARSING WITH PEG GRAMMARS

3.1 Representing precedence levels with PEG rules

Because PEG parsers are naturally greedy, and operate in a top-down, left-to-right order in their standard recursive descent form, building a hierarchy of operator precedence into a grammar can be complicated. Because left recursive grammars result in infinite recursion, a grammar containing a left-recursive cycle must be rewritten in a non-left-recursive form, usually by using a `OneOrMore` operator, complicating the grammar, particularly if a parse tree contains more than one loop around the grammar cycle.

However, when PEG parsing is run bottom-up, cycles are handled naturally, so it becomes very natural to build precedence into the parser. Consider the following grammar, where `E[`i`]` defines grammar rule `E` for precedence level i, and `E` on the right hand side of the rule is a self-reference that respects operator precedence:

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

Listing 12. Example grammar containing cycles (self-references); rule names have precedence suffix

This grammar can be rewritten directly as standard PEG rules, where each reference to `E` is rewritten to refer to the next highest level of precedence of the grammar, as required by the semantics of precedence. If an entire underlying grammar rule fails, then the grammar simply defers to the next highest level of precedence, effectively elevating the match attempt to the next highest level of precedence.

(With top-down PEG parsing, this failover to higher precedence levels results in a depth-first postorder traversal of the grammatical structure of the input, so even though it is counterintuitive to start recursing at the lowest-precedence rule, this is the correct order to ensure proper precedence semantics. Elsewhere in the grammar, references to rule `E` should resolve to the lowest precedence clause, i.e. references to `E` from grammar rules other than E should rewrite `E` to `E0`, or should match the implicit rule `E <- E0`.)

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

Listing 13. Example grammar with precedence (does not handle associativity)

3.2 Representing both precedence and left-associativity with PEG rules

However, even though (with either top-down or bottom-up parsing) this grammar can handle nested levels of precedence, e.g. `"1*2+3*4"`, it cannot handle runs of equal precedence, or direct nesting of operators of equal precedence, for example `"1+2+3"` or `"((2+3))"` or `"--3"`.

It is tempting to handle this situation by ensuring that one of the operands defers first to the same level of precedence, and then to the next highest level of precedence, for example it would be easy to assume that the following grammar implements left precedence for `E0` and E1:

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

Listing 14. Example grammar, handling associativity using `First`

However, although this left-recursive grammar can match `"1+2+3"` (as `(1+2)+3` as expected), as well as `"((2+3))"` and `"--3"`, it cannot match the string `"1+2*(3+4)"`, because this requires multiple loops around the grammar cycle – the "looped precedence problem". Top-down parsing cannot even handle this grammar without rewriting the rules into non-left-recursive `OneOrMore` form, whereas bottom-up parsing gets stuck due to the greediness of PEG operators. (Note that a right-associative self-recursive precedence rule such as `E0 <- (E1 '+' (E0 / E1)) / E1` would not get stuck.)

(Note that to get the grammar to "wrap around" so that it is possible to parse nested parenthesized clauses of arbitrary depth, the rule `E4`, originally `E4 <- '(' E0 ')'`, needs to be rewritten in the form `E4 <- '(' (E4 / E0) ')'`. Actually, following the rewriting pattern of the other rules, the highest-precedence rule should probably be written `E4 <- ('(' (E4 / E0) ')') / E0`, but the final fallthrough to `E0` is not necessary, since the lowest-precedence rule `E0` is used as the root clause for a fully-parsed nested expression, using `E <- E0`.)

For the input `"1+2*(3+4)"`, the failure of bottom-up parsing due to left recursion occurs as follows:

- The initial terminal matches are added to the memo table, including ⟨`[0-9]`, 2⟩ matching `"2"`.
- ⟨`[0-9]+`, 2⟩ matches `"2"`.
- ⟨`E3`, 2⟩ matches `"2"`.
- ⟨`E2`, 2⟩ matches `"2"`, through the final `E3` subclause of rule `E2`.

- ⟨E0, 5⟩ matches "3+4".
- ⟨E1, 2⟩ fails to match "2*(...", because there is no ⟨E4, 4⟩ match yet. Therefore, the final "failover" subclause promotes the match to E2, and ⟨E1, 2⟩ is found to match "2".
- ⟨E4, 4⟩ matches "(3+4)".
- ⟨E0, 0⟩ matches "1+2", because its third subclause ⟨E1, 2⟩ currently matches "2". This fragment of the putative parse tree does not respect precedence, so in theory the rightmost sub-clause match ("2") should be replaced with a better match in a future iteration, specifically with "2*(3+4)". (That doesn't end up happening.)
- ⟨E1, 2⟩ matches "2*(3+4)". This is a new better match than the previous match for this memo entry, so it triggers ⟨E0, 0⟩ to be added to activeSet for re-matching in the following iteration, through the backRefs mechanism, since the third subclause match of ⟨E0, 0⟩ is ⟨E1, 2⟩.
- When attempting to match ⟨E0, 0⟩ again, E0's own directly left-recursive subclause (E0 / E1) greedily trips up the match, because this subclause will now greedily consume the entire previous best match for ⟨E0, 0⟩ as its first operand, and will try to parse ('+'/ '-') in the fourth character position, after "1+2"; therefore, the match fails.
- The E1 rule cannot itself fix the problem by expanding its own best batch from "2*(3+4)" to "1+2*(3+4)", because the "1+" part of the input only matches at lower precedence.
- The memo table ends up with two locally-best matches that overlap by one character, "1+2" and "2*(3+4)", and the parsing ends without producing a complete parse tree.

For similar reasons, this grammar cannot parse "1+2+3*4*5", due to an overlap of E0 and E1 matches at "3".

There is a straightforward way to fix this problem with the parser, but it would lead to a potentially large performance degradation. Instead of recording a single best match within each memo entry, all previous matches should be stored within each memo entry, and then when a rule is applied, it should try the cross product of all previous matches of all subclauses.

The length of the set of previous matches for a given memo entry is a linear function of the number of loops around a grammar cycle taken by a path from the memo entry to its descendant terminals. For the clause type Seq, the cross product of all previous matches for all subclauses would result in a performance degradation that grows as the product of the number of loops around a grammar cycle for each subclause, i.e. the performance degradation would be polynomial function of the number of loops in the parse tree (which can be roughly simplified to a polynomial function of the depth of the parse tree). However, for OneOrMore, the number of subclause matches can be arbitrarily large, resulting in a cross product of an arbitrary number of previous subclause matches, yielding a potentially exponential degradation in performance.

For many grammars, in practical terms the performance degradation of this parser modification would be sub-exponential and in fact minimal, since most clauses will only ever produce zero matches or or one matches at any given input position. The suggested parser modification would actually result in a fundamentally more powerful parser than a standard top-down PEG parser, because that there would be grammars that could be parsed by this modified bottom-up parser that could not be parsed by a standard bottom-up (or top-down) PEG parser. However, since the worst case is exponential, this change is not advisable.

We therefore need to add a caveat to the advantages listed in Section 2.1: an unmodified pika parser can handle arbitrary left-recursive grammars, without problems, for many inputs. However, there are inputs for which parsing will not complete. When this does happen, figuring out why the parse failed can be difficult and non-intuitive. The parser

can be modified to fix this, so that all left recursive grammars work for all inputs, but at the cost of potentially losing the linear-time performance of the parser. Therefore, *the ability to parse grammars containing arbitrary left-recursive cycles may be mutually exclusive with keeping the performance of the parser linear in the length of the input and the depth of the parse tree.*

This a direct consequence of the selection of PEG as the fundamental operator type, since PEG operators are inherently greedy in nature. It may be possible to come up with a different set of operators that does not exhibit this behavior. However, almost certainly this set of operators does not exist, at least for recursive descent parsers (or their bottom-up inversion), because there is no way to directly write a set of grammar rules of different precedence levels such that (1) the grammar is not left-recursive, but (2) parsing an input with the grammar produces a left-associative parse tree. This is because for recursive descent parsing, the structure of the grammar (and therefore the structure of recursive calls) is always exactly embedded in the structure of the resulting parse tree.

There is one other possible way of solving this problem in the parser that may preserve linear-time parsing (see Section 3.4). However, it is worth first considering a way to fix this issue in the grammar rather than the parser.

3.3 Rewriting grammar rules to handle "precedence ambiguity"

Instead, we can take a clue from the rewriting of left-recursive rules into non-left-recursive (`OneOrMore`) form to guess at how this conundrum could be solved with grammar rewriting.

Directly applying the heuristics used to rewrite left-recursive rules into non-left-recursive form would solve the left recursion problem not just for top-down parsing, but also for bottom-up parsing. However, these rewriting heurstics can be complex, *since they try to ensure that the produced parse tree maintains left-associative structure* – and for inputs that require multiple loops around grammar cycles, rewriting rules in this way can result in a blowup of the size and complexity of the grammar. There is a better alternative, if we relax the requirement that the parse tree should maintain left-associative structure.

Any rule at a given precedence level that has a left-recursive reference to its own precedence level will fail due to greediness, as demonstrated. The whole purpose of adding these self-references from a precedence level to itself is to add left associativity in the context of precedence. However, associativity is only a convention used to specify how to interpret an arithmetic expression. Any notation that requires interpretation based on conventions is fundamentally ambiguous. The right way to approach this problem is to not even try to resolve the interpretation of equal-precedence runs of operands and operators until *after* parsing has completed – in other words, precedence should not even be resolved in the parse tree, it should be resolved if necessary in a postprocessing step, perhaps when converting the parse tree into an Abstract Syntax Tree, or the Abstract Syntax Tree into a Concrete Syntax Tree.

In other words, precedence levels should only ever reference higher precedence levels, except in the case of a *precedence-breaking pattern* (e.g. a rule that matches a set of parentheses around a lower-precedence clause).

Runs of equal precedence should be collected into `OneOrMore` matches – this will be referred to as "precedence-agnostic form". For binary (two-arg) arithmetic expressions, these can be directly written as right-recursive rules. This avoids the complexity of the standard heuristics of rewriting left-recursive rules into non-left-recursive rules, since we no longer care about resolving associativity as the parse tree is built.

The rewritten grammar is as follows:

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

Listing 15. Example grammar, handling associativity using right-recursive `OneOrMore`

Using this grammar, all adjacent terms of equal precedence (i.e. all `E0` and `E1` matches) will match as sibling subclauses of a `OneOrMore` match in the final parse tree.

This grammar will correctly parse `"1+2*(3+4)"` as `1+(2*(3+4))`, and will correctly parse `"1+2+3*4*5"` as `(1+2+(3*4*5)` (although the parse tree in this latter case consists of three subclause match nodes per non-leaf match node, due to the `OneOrMore` clause in the revised grammar, rather than the normal two child nodes for arithmetic operators in left- or right-associative parse trees).

This grammar rewriting pattern is simple general enough to satisfy most needs for precedence parsing, without modifying the parser.

3.4 A possible parser modification for parsing left-recursive grammars in linear time

However there may be one possible way to solve the looped precedence issue, allowing the parser to directly parse left-recursive grammars while maintaining linear parsing performance: "manual looping". This entails:

- No precedence level should refer to itself, only to higher precedence levels. (In the case of the grammar examples used so far, `E4` is a `precedence-breaking pattern`, because the clause that potentially loops back to `E0` has to be surrounded by anambiguously-distinguishing parentheses, so this rule is allowed to loop back to `E0`.)
- This will cause all clauses to parse with higher-precedence operands, until further such matches are impossible. Beyond that point, the grammar rule will be promoted to the next highest level of precedence, using the final failover clause. The effect of this is that any clause that successfully matched `E2` using `E3` operands, but where better `E3` operand matches are not yet known, will result in both `E1` and `E0` matching at the same input position and with the same length.
- The parser is then allowed to run to completion. If there is a single parse tree generated spanning all input, parsing is complete.
- Otherwise, we try to "loop the grammar" by temporarily replacing references to `E[`$i+1$`]` with `E[`i`]` in the left subclause of rule `E[`i`]` (i.e. turning the rule into left-recursive form), then restarting the parser to see if anything else matches, then undoing the temporary replacement again.
- Probably this replacement needs to be performed for only one precedence level i at a time, starting from the highest precedence level and iterating down to the lowest precedence level. If after temporarily replacing the left subclause with a self-reference, the parser is able to find new matches, then i should be reset back to the highest precedence level again. (This will result in a degradation of performance that is quadratic in the number of precedence levels in the worst case, but since the number of precedence levels is a constant, this results in only a constant time-complexity multiplier, so the parser will still run in time linear in the length of the input and the depth of the parse tree.)
- Once i reaches the lowest precedence level and no new matches were found, the parse is complete.

The reason this parser modification was not yet attempted (beyond the fact that this would add some degree of complexity to the parser) is that although this mechanism should work for a single set of stacked precedence levels, if

there are two or more mutually recursive sets of grammar rules, each employing their own precedence levels, it is not clear in what order the replacements should be made to avoid the parser getting stuck in pathological cases, because there are now multiple separate and nested total orders of precedence. There is probably a nice graph-theoretic proof of a method to find the correct order to "manually loop" mutually-recursive sets of precedence levels to result in a correct parse tree, but this method is not yet known, and is left as future work.

3.5 Reducing the depth of parse trees with higher precedence selector clauses

The "precedence promotion" mechanism (using a final failover clause that promotes a rule to the next highest precedence level if it doesn't match) can result in deep parse trees, since intermediate parse tree nodes will be created for all intermediate precedence levels between the two precedence levels of an outer expression and an inner expression (so if the input skips over one or more precedence levels, the parse tree will still contain a node for the skipped precedence levels).

This can be mitigated by replacing the use of failover clauses with *higher precedence selector clauses*. This higher precedence selector form produces a grammar that is looks more complicated, but is likely to parse faster than the failover mechanism, and produces shallower parse trees.

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

Listing 16. Example grammar, handling associativity using higher precedence selector clauses

The reference parser (Section 5) can automatically transform a precedence grammar into higher precedence selector form, however the user must currently still write the grammar in precedence agnostic form (Section 3.3), using `OneOrMore` to collect terms of equal precedence.

3.6 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, using the syntax `(LabelName ':' 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. 2).

4 BROADER APPLICATION OF INVERSION OF RECURSION

The techniques presented in this paper for converting a top-down recursive algorithm into a bottom-up dynamic programming algorithm may have applications beyond parsing, and could bring the benefits of pika parsing relative to top-down packrat parsing (Section 2.1) to other non-parsing recursive algorithms.

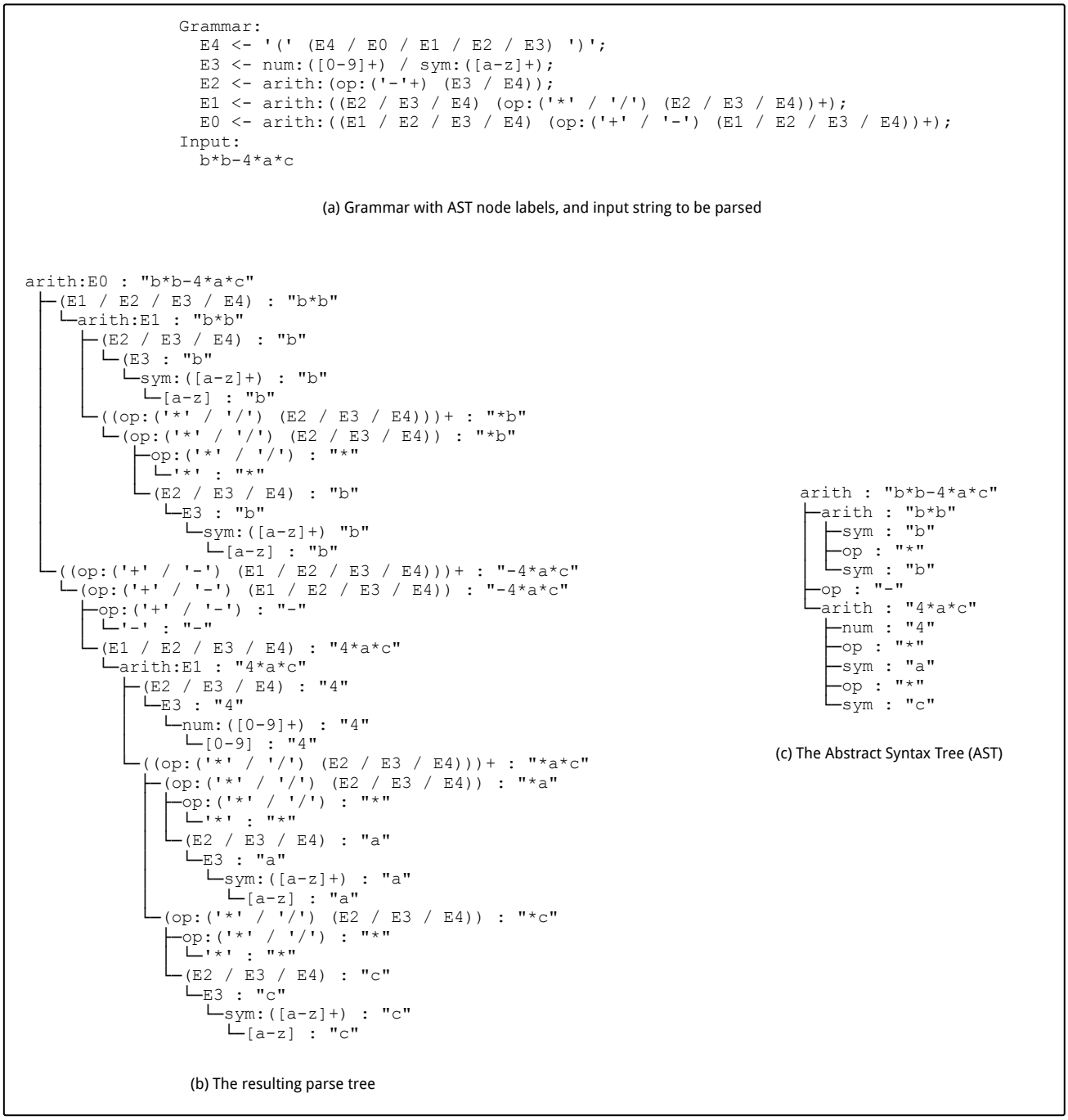

Fig. 2. The conversion of a parse tree into an AST, by annotating grammar clauses with AST node labels

5 REFERENCE IMPLEMENTATION

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

There are numerous details that can be tricky to get right when implementing this parser, so directly porting the reference parser to other languages will save time compared to writing a parser from scratch using the description in this paper.

The reference implementation includes a `meta-grammar` that is able to parse PEG rules in standard textual form, creating a grammar that can then be used to parse documents conforming to the grammar.

6 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 without any rewriting of grammar rules, 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). Additionally, a pika parser can be parallelized to accelerate parsing across available cores. A lexing step can be applied to reduce the size of the memo table and the amount of work wasted by bottom-up parsing compared to top-down parsing. A fundamental tension between left recursion (or left associativity) and linear time performance of recursive descent parsers was elucidated. Mechanisms for implementing precedence and associativity were demonstrated, and several new insights were provided into the problem of left-recursive grammars and the handling of precedence order by PEG parsers, or more generally by recursive descent parsers. The techniques presented 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] Bryan Ford. 2002. *Packrat parsing: a practical linear-time algorithm with backtracking*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[12] 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.

[13] 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.

[14] 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.

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

[16] Stephen Cole Kleene. 1951. Representation of events in nerve nets and finite automata. *RAND Research Memorandum* RM-704 (1951).
[17] Donald E Knuth. 1962. History of writing compilers. In *Proceedings of the 1962 ACM National Conference on Digest of Technical Papers*. 43.
[18] Donald E Knuth. 1965. On the translation of languages from left to right. *Information and control* 8, 6 (1965), 607–639.
[19] Joseph B Kruskal. 1983. An overview of sequence comparison: Time warps, string edits, and macromolecules. *SIAM review* 25, 2 (1983), 201–237.
[20] Bernard Lang. 1974. Deterministic techniques for efficient non-deterministic parsers. In *International Colloquium on Automata, Languages, and Programming*. Springer, 255–269.
[21] Robert McNaughton and Hisao Yamada. 1960. Regular expressions and state graphs for automata. *IRE transactions on Electronic Computers* 1 (1960), 39–47.
[22] Peter Norvig. 1991. Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics* 17, 1 (1991), 91–98.
[23] Klaus Samelson and Friedrich L Bauer. 1960. Sequential formula translation. *Commun. ACM* 3, 2 (1960), 76–83.
[24] Michael Sipser. 2012. *Introduction to the Theory of Computation*. Cengage Learning.
[25] Masaru Tomita. 2013. *Efficient parsing for natural language: a fast algorithm for practical systems*. Vol. 8. Springer Science & Business Media.
[26] 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.