PEG Parsing in Less Space using Progressive Tabling and Dynamic Analysis

Fritz Henglein and Ulrik Terp Rasmussen

Department of Computer Science, University of Copenhagen (DIKU), Denmark

{henglein, dolle}@diku.dk

Abstract

Tabular top-down parsing and its lazy variant, Packrat, are linear-time execution models for the TDPL family of recursive descent parsers with limited backtracking. Exponential work due to backtracking is avoided by tabulating the result of each (nonterminal, offset)-pair at the expense of always using space proportional to the product of the input length and grammar size. Current methods for limiting the space usage rely either on manual annotations or on static analyses that are sensitive to the syntactic structure of the grammar.

We present *progressive tabular parsing* (PTP), a new execution model which progressively computes parse tables for longer prefixes of the input and simultaneously generates a leftmost expansion of the parts of the parse tree that can be resolved. Table columns can be discarded on-the-fly as the expansion progresses through the input string, providing best-case constant and worst-case linear memory use. Furthermore, semantic actions are scheduled before the parser has seen the end of the input. The scheduling is conservative in the sense that no action has to be "undone" in the case of backtracking.

The time complexity is $O(dmn)$ where m is the size of the parser specification, n is the size of the input string, and d is either a configured constant or the maximum parser stack depth.

For common data exchange formats such as JSON, we demonstrate practically constant space usage.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—*Parsing*; F.4.2 [*Mathematical Logic and Formal Languages*]: Grammars and Other Rewriting Systems—*Parsing*

Keywords Parsing expression grammars, context-free grammars, regular expressions, Packrat, streaming parsing

1. Introduction

Parsing of computer languages has been a topic of research for several decades, leading to a large family of different parsing methods and formalisms. Still, with each solution offering varying degrees of expressivity, flexibility, speed and memory usage, and often at a trade-off, none of them can be regarded as an ideal general approach to solving all parsing problems. For example, compiler writers often specify their languages in a declarative formalism such as context-free grammars (CFG), relying on LL(k) or LR(k) parser generators to turn their specifications into executable parsers. The resulting parsers are often fast, but with the downsides that a separate lexical preprocessing is needed, and that the programmer is required to mold the grammar into a form that is deterministic for the chosen parser technology. Such solutions require a large investment in time, as identifying the sources of nondeterminism in a grammar can be quite difficult. A user who needs an ad-hoc parser will thus not find the amount of time invested to make up for the apparent benefits.

Aho and Ullman's TDPL/GTDPL languages [1], which were later popularized as Parsing Expression Grammars (PEG) [8], provide a formal foundation for the specification of recursive-descent parsers with limited backtracking. They do away with the problem of nondeterminism by always having, by definition, a single unique parse for every accepted input. The syntax of PEGs resembles that of CFGs, but where a CFG is a set of generative rules specifying its language, a PEG is a deterministic backtracking program, whose language is the set of strings on which it terminates without failure. This ensures unique parses, but with the downside that it can sometimes be quite hard to determine what language a given PEG recognizes. Recognition can be performed in linear time and space by an algorithm which computes a table of results for every (nonterminal, input offset)-pair [1], although it seems to never have been used in practice, probably due to its large constant factors in complexity. Ford's Packrat parsing [7] reduces these constants by only computing the table entries that are needed to construct the actual parse. However, the memory usage of Packrat is $\Theta(mn)$ for PEGs of size m and inputs of size n, which can be prohibitively expensive for large m and n, and completely precludes applying it in a streaming context where input is potentially infinite. Heuristics for reducing memory usage [13; 21] still store the complete input string, and even risk triggering exponential time behavior. One method [18] can remove both table regions and input prefixes from memory during runtime, but relies on manual annotations and/or a static analysis which does not seem to perform well beyond LL languages [22].

In this paper, we present *progressive tabular parsing* (PTP), a new execution model for the TDPL family of languages. The method is based on the tabular parsing of Aho and Ullman, but avoids computing the full parse table at once. We instead start by computing a table with a single column based on the first symbol in the input. For each consecutive symbol, we append a corresponding column to the table and update all other entries based on the newly added information. We continue doing this until the end of the input has been reached and the full parse table has been computed. During this process, we have access to partial parse tables which we use to guide a leftmost expansion of the parse tree for the overall parse. Whenever a prefix of the input leads to a unique parse tree expansion, the prefix and its corresponding table columns can be removed from memory. The result is a linear-time parsing algorithm which still uses $O(mn)$ memory in the worst case, but

$O(m)$ in the best case. Since we have access to the partial results of every nonterminal during parsing, a simple dynamic analysis can use the table to rule out alternative branches and speculatively expand the parse tree before the corresponding production has been fully resolved. The speculation is conservative and never has to undo an expansion unless the whole parse turns out to fail. The analysis changes the time complexity to $O(dmn)$ for a configurable constant d bounded by the maximum stack depth of the parser, but preliminary experiments suggests that it pays for itself in practice by avoiding the computation of unused table entries.

The method can be formulated elegantly using least fixed points of monotone table operators in the partial order of tables with entrywise comparison, and where unresolved entries are considered a bottom element in the partial order. The computation of parse tables is then an instance of *chaotic iteration* [5] for computing least fixed points using a work set instead of evolving all entries in parallel. The work set is maintained such that we obtain meaningful partial parse tables as intermediate results which can be used by the dynamic analysis. Linear time is obtained by using an auxiliary data structure to ensure that each table entry is added to the work set at most once.

Our evaluation demonstrates that PTP dynamically adapts its memory usage based on the amount of lookahead required to resolve productions. The complexity constant due to indiscriminately computing all entries of the parse table can be quite large, but can be reduced by applying an optimization inspired by Ford's Packrat algorithm, which reduces the amount of work by an order of magnitude on some inputs. We believe that our general formulation of PTP offers a solid foundation for further development of both static and dynamic analyses for improving performance.

To summarize, we make the following contributions:

- *Progressive tabular parsing* (PTP), a new execution model for the TDPL family of parsing formalisms. The execution of a program proceeds by progressively computing parse tables, one for each prefix of the input, using the method of *chaotic iteration* for computing least fixed points. Meanwhile, a leftmost expansion of the parse tree is generated in a streaming fashion using the parse table as an oracle. Table columns are discarded on-the-fly as soon as the method detects that a backtracking parser would never have to return to the corresponding part of the input.
- An algorithm for computing progressive parse tables in an incremental fashion. It operates in amortized time $O(mn)$ for grammars of size m and inputs of size n, and produces n progressive approximations of the parse table. We show that for certain grammars and inputs, as little as $O(m)$ space is consumed.
- A configurable dynamic analysis which can dramatically improve the streaming behavior of parsers by allowing a longer trace to be generated earlier in the parse. The dynamic analysis changes the time complexity to $O(dmn)$ where d is either a configured constant or the maximum parser stack depth.
- An optimized version of the above algorithm which uses a technique inspired by Packrat parsing to reduce the amount of work by an order of magnitude on certain inputs.
- An evaluation of a prototype of the algorithm which demonstrates that a) for an unannotated JSON parser written in the PEG formalism, memory usage is practically constant, b) for parsers of non-LL languages, the algorithm adjusts memory usage according to the amount of lookahead required, c) however, ambiguous tail-recursive programs trigger worst-case behavior.

The paper is organized as follows. The GTDPL and PEG parsing formalisms are introduced in Section 2 together with bit-serialized parse trees and *streaming parsing*. In Section 3 we recall the linear-time tabular parsing method, but defined using least fixed points. We extend this in Section 4 to obtain an approximation of the full parse table based on a prefix of the full input string. In the same section, we define the streaming generation of execution traces based on dynamic analysis of approxmation tables, which we then use to present the *progressive tabular parsing* method. In Section 5 we exhibit—and prove correct—an amortized linear-time algorithm for computing all progressive table approximations for all consecutive prefixes of an input string, and we also present an optimization inspired by Packrat parsing. A prototype implementation is evaluated on three different parsing programs in Section 6. We conclude with a discussion of related and future work in Section 7.

2. Parsing Formalism

The *generalized top-down parsing language* (GTDPL) is a language for specifying top-down parsing algorithms with limited backtracking [1; 3]. It has the same recognition power as the *top-down parsing language* (TDPL), from which it was generalized, and *parsing expression grammars* (PEG) [8], albeit using a smaller set of operators.

The top-down parsing formalism can be seen as a recognition-based alternative to declarative formalisms used to describe machine languages, such as context-free grammars (CFGs). A CFG constitutes a set of generative rules that characterize a language, and the presence of ambiguity and nondeterminism poses severe challenges when such a specification must be turned into a deterministic parsing algorithm. In contrast, every GTDPL/PEG by definition denotes a deterministic *program* which operates on an input string and returns with an outcome indicating failure or success.

Definition 1 (Program). A GTDPL program (henceforth just *program*) is a tuple $P = (\Sigma, V, S, R)$ where

1. Σ is a finite input alphabet; and
2. V is a finite set of *nonterminal* symbols; and
3. $S \in V$ is the starting nonterminal; and
4. $R = \{A_0 \leftarrow g_0, ..., A_{m-1} \leftarrow g_{m-1}\}$ is a non-empty finite set of numbered *rules*, where each A_i is in V and each $g_i \in \mathsf{GExpr}$ is an *expression* generated by the grammar

$$\mathsf{GExpr} \ni g ::= \epsilon \mid \mathsf{f} \mid a \mid A[B, C]$$

where $A, B, C \in V$, $a \in \Sigma$, and ϵ, f are distinct symbols not in Σ. Rules are unique: $i \neq j$ implies $A_i \neq A_j$.

Define the *size* $|P|$ of a program to be the cardinality of its rule set $|R| = m$. When P is understood, we will write $A \leftarrow g$ for the assertion $A \leftarrow g \in R$. By uniqueness of rule definitions, we can write i_A for the unique index of a rule $A_i \leftarrow g_i$ in R. If g_i is of the form $B[C, D]$ it is called a *complex expression*, otherwise it is *simple*.

The intuitive semantics of a production $A \leftarrow a$ is to read an a off the input string and return the remainder. If the first symbol is not an a, it fails and returns f. A production $A \leftarrow \epsilon$ always succeeds reading no symbols, and $A \leftarrow \mathsf{f}$ always fails. A production $A \leftarrow B[C, D]$ first tries parsing the input with B, and if this succeeds, parses the remainder with C. Only if B fails does it backtrack and parse from the beginning of the input with D. For this reason we call B the *condition* and C and D the *continuation branch* and *failure branch*, respectively. The semantics are formally defined in the following.

Definition 2 (Operational semantics). Let $P = (\Sigma, V, S, R)$ be a program. The *matching relation* $\Rightarrow_P$ is the smallest binary relation $\Rightarrow \subseteq (V \times \Sigma^*) \times (\Sigma^* \uplus \{\mathsf{f}\})$ closed under the following rules:

$$(1)\ \frac{}{(A, u) \Rightarrow u}\ (A \leftarrow \epsilon) \qquad (2)\ \frac{}{(A, u) \Rightarrow \mathsf{f}}\ (A \leftarrow \mathsf{f})$$

$$(3\mathrm{i})\ \frac{}{(A, au) \Rightarrow u}\ (A \leftarrow a)$$

$$\text{(3ii)}\quad \frac{}{(A,u) \Rightarrow \mathsf{f}} \quad (A{\leftarrow}a \text{ and } a \text{ not prefix of } u)$$

$$\text{(4i)}\quad \frac{(B,u) \Rightarrow v \qquad (C,v) \Rightarrow r}{(A,u) \Rightarrow r} \quad (A{\leftarrow}B[C,D])$$

$$\text{(4ii)}\quad \frac{(B,u) \Rightarrow \mathsf{f} \qquad (D,u) \Rightarrow r}{(A,u) \Rightarrow r} \quad (A{\leftarrow}B[C,D])$$

The proof derivations generated by the rules will be denoted by subscripted variations of the letter $\mathcal{D}$.

We write $(A,u) \not\Rightarrow_P$ if there is no r such that $(A,u) \Rightarrow_P r$ and say that A *matches* u if $(A,u) \Rightarrow_P v$ for some $v \in \Sigma^*$. (Note that A does not have to consume all of the input.) The *language* $L_P(A)$ *recognized* by A is the set of $u \in \Sigma^*$ it matches. The language *rejected* by A is $\overline{L}_P(A) = \{u \in \Sigma^* \mid (A,u) \Rightarrow_P \mathsf{f}\}$. A program P is *complete* if the start symbol S accepts or rejects each string.

The following two properties are easily shown by induction.

Proposition 2.1 (Suffix output). *If $(A,u) \Rightarrow_P w$, then w is a suffix of u: $\exists v.\, u = vw$.*

Proposition 2.2 (Determinacy). *If $(A,u) \Rightarrow_P r_1$ and $(A,u) \Rightarrow_P r_2$, then $r_1 = r_2$.*

We recall the following negative decidability results proved by Ford for the PEG formalism [8]. Since any GTDPL can be effectively converted to an equivalent PEG and vice-versa, they hold for GTDPL as well.

Proposition 2.3. *It is undecidable whether $L_P(A) = \emptyset$ and whether $L_P(A) = \Sigma^*$.*

Proposition 2.4. *It is undecidable whether a program is complete.*

2.1 Parsing Expression Grammars

Having only a single complex operator, GTDPL offers a minimal foundation which simplifies the developments in later sections. The drawback is that it is very hard for a human to determine the language denoted by a given GTDPL program. In order to make examples more readable, we will admit programs to be presented with expressions from the extended set PExpr defined as follows:

$$\mathsf{PExpr} \ni e ::= g \in \mathsf{GExpr} \mid e_1 e_2 \mid e_1/e_2 \mid e^* \mid !e_1$$

This corresponds to the subset of *predicate-free parsing expressions* extended with the ternary GTDPL operator. A program P with productions in PExpr is called a *PEG program*, and desugars to a pure GTDPL program by adding productions $E{\leftarrow}\epsilon$ and $F{\leftarrow}\mathsf{f}$ and replacing every non-conforming production as follows:

$A{\leftarrow}e_1e_2$	$\longmapsto$	$A{\leftarrow}B[C,F]$
		$B{\leftarrow}e_1$
		$C{\leftarrow}e_2$
$A{\leftarrow}e_1/e_2$	$\longmapsto$	$A{\leftarrow}B[E,C]$
		$B{\leftarrow}e_1$
		$C{\leftarrow}e_2$
$A{\leftarrow}e_1^*$	$\longmapsto$	$A{\leftarrow}B[A,E]$
		$B{\leftarrow}e_1$
$A{\leftarrow}!e_1$	$\longmapsto$	$A{\leftarrow}B[F,E]$
		$B{\leftarrow}e_1$

The desugaring embeds the semantics of PEG in GTDPL [8], so there is no need to introduce semantic rules for parsing expressions. Note that although parsing expressions resemble regular expressions, the recognizers that they denote may not recognize the same languages as their usual set-theoretic interpretation. For example, the expression a^*a recognizes the empty language! For every string of the form a^n, $n \geq 0$, subexpression a^* successfully consumes all n symbols, leaving the empty string to be matched by subexpression a, which subsequently fails.

2.2 Parse Trees

We are usually interested in providing a parse tree instead of just doing recognition, e.g. for the purpose of executing semantic actions associated with parsing decisions. Unlike generative frameworks, any program uniquely matches an input via a unique derivation $\mathcal{D}$, which we therefore could take as our notion of parse tree. However, for space complexity reasons, we will employ a more compact notion for which we also define a bit coding for the purpose of providing a definition of streaming parsing.

A *parse tree* $\mathcal{T}$ is an ordered tree where each leaf node is labeled by the empty string or a symbol in Σ, and each internal node is labeled by a nonterminal subscripted by a symbol from $\mathbf{2} \cup \{\varepsilon\}$ where $\mathbf{2} = \{0,1\}$ and ε denotes the empty string (to distinguish it from the GTDPL *expression* ϵ).

Definition 3 (Parse trees and codes). For any $A \in V$, $u, v \in \Sigma^*$, and derivation $\mathcal{D} :: (A,u) \Rightarrow_P v$, define simultaneously a *parse tree* $\mathcal{T}_\mathcal{D}$ and a *parse code* $\mathcal{C}_\mathcal{D} \in \mathbf{2}^*$ by recursion on $\mathcal{D}$:

1. If $A{\leftarrow}\epsilon$, respectively $A{\leftarrow}a$, then $\mathcal{T}_\mathcal{D}$ is a node labeled by A_ε with a single child node labeled by ε, respectively a. Let $\mathcal{C}_\mathcal{D} = \varepsilon$.
2. If $A{\leftarrow}B[C,D]$ and $\mathcal{D}_1 :: (B,u) \Rightarrow_P u'$ we must have $\mathcal{D}_2 :: (C,u') \Rightarrow_P v$. Let $\mathcal{T}_\mathcal{D}$ be a node A_0 with subtrees $\mathcal{T}_{\mathcal{D}_1}$ and $\mathcal{T}_{\mathcal{D}_2}$. Let $\mathcal{C}_\mathcal{D} = 0\,\mathcal{C}_{\mathcal{D}_1}\mathcal{C}_{\mathcal{D}_2}$.
3. If $A{\leftarrow}B[C,D]$ and $\mathcal{D}_1 :: (B,u) \Rightarrow_P \mathsf{f}$, then we must have $\mathcal{D}_2 :: (D,u') \Rightarrow_P v$. Create a node labeled by A_1 with a single subtree $\mathcal{T}_{\mathcal{D}_2}$. Let $\mathcal{C}_\mathcal{D} = 1\,\mathcal{C}_{\mathcal{D}_2}$.

The code $\mathcal{C}_\mathcal{D}$ intuitively encodes for each encountered $A[B,C]$ whether the condition A succeeds or fails. This compactly encodes $\mathcal{T}_\mathcal{D}$ which can be reconstructed without dependence on the derivation $\mathcal{D}$ or even the input u.

The size of a parse tree $|\mathcal{T}|$ is the number of nodes in it. Note that only the parts of a derivation counting towards the successful match contribute to its parse tree, while failing subderivations are omitted. This ensures that, for fixed grammars, parse trees have size proportional to the input. This is in contrast to *derivations*, which be of exponential size in the length of the input in the worst case.

Proposition 2.5 (Linear tree complexity). *Fix a program P. For all $A \in V$ and $u, v \in \Sigma^*$ and derivations $\mathcal{D} :: (A,u) \Rightarrow_P v$ we have $|\mathcal{T}(\mathcal{D})| = O(|u|)$.*

Parse trees and parse codes both provide injective codings of the subset of derivations with non-failing results.

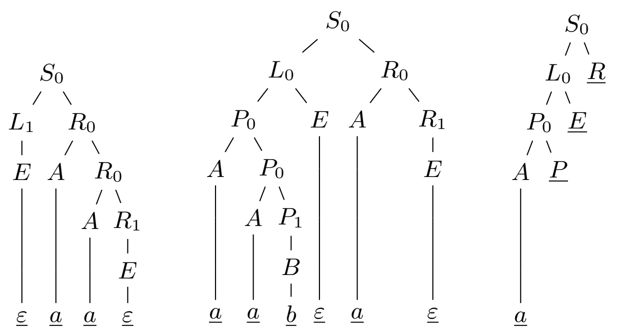
Proposition 2.6 (Injectivity). *Fix a program P and symbol $A \in V$. For all $u_1, u_2, v_1, v_2 \in \Sigma^*$ and derivations $\mathcal{D}_1 :: (A,u_1) \Rightarrow_P v_1$ and $\mathcal{D}_2 :: (A,u_2) \Rightarrow_P v_2$, if $\mathcal{D}_1 \neq \mathcal{D}_2$, then $\mathcal{T}_{\mathcal{D}_1} \neq \mathcal{T}_{\mathcal{D}_2}$ and $\mathcal{C}_{\mathcal{D}_1} \neq \mathcal{C}_{\mathcal{D}_2}$.*

It is easy to check that a code can be used to construct the corresponding parse tree in linear time, regardless of the size of the underlying derivation. In general, a code can be viewed as an oracle that guides a leftmost derivation of the start symbol S to match the input string. Any prefix of a code can thus be seen as a partially expanded parse tree. During expansion, we maintain a stack of nodes that are not yet expanded. If the top node is simple it can be expanded deterministically, and if it is complex the next code symbol determines its expansion; its child nodes are pushed on the stack.

Example 1. Consider the PEG program $S{\leftarrow}(a^*b/\epsilon)a^*$, which desugars into:

$S{\leftarrow}L[R,F]$	$L{\leftarrow}P[E,E]$	$P{\leftarrow}A[P,B]$	$R{\leftarrow}A[R,E]$
$A{\leftarrow}a$	$B{\leftarrow}b$	$E{\leftarrow}\epsilon$	$F{\leftarrow}\mathsf{f}$

We have derivations $\mathcal{D} :: (S, aa) \Rightarrow \varepsilon$ and $\mathcal{D}' :: (S, aaba) \Rightarrow \varepsilon$. Visualized below is, from left to right: the trees $\mathcal{T}_{\mathcal{D}}$, $\mathcal{T}_{\mathcal{D}'}$, and the partial tree expanded from the prefix 000 of the code $\mathcal{C}_{\mathcal{D}'}$. The leftmost nonterminal leaf is the next to be expanded.


S_0
L_1
R_0
E
A
R_0
A
R_1
E
$\underline{\varepsilon}$ $\underline{a}$ $\underline{a}$ $\underline{\varepsilon}$
S_0
L_0
R_0
P_0
E
A
R_1
A
P_0
E
A
P_1
B
$\underline{a}$ $\underline{a}$ $\underline{b}$ $\underline{\varepsilon}$ $\underline{a}$ $\underline{\varepsilon}$
S_0
L_0
$\underline{R}$
P_0
$\underline{E}$
A
$\underline{P}$
$\underline{a}$

The parse codes are $\mathcal{C}_{\mathcal{D}} = 01001$ and $\mathcal{C}_{\mathcal{D}'} = 0000101$, respectively. Observe that codes correspond to the subscripts of the internal nodes in the order they would be visited by an in-order traversal, reflecting the leftmost expansion order.

2.3 Streaming Parsing

Using parse codes, we can define *streaming parsing*.

Definition 4 (Streaming parsing function)**.** Let $\# \notin \Sigma$ be a special end-of-input marker. A *streaming parsing function* for a program P is a function $f : \Sigma^*(\# \cup \varepsilon) \to \mathbf{2}^*$ which for every input prefix $u \in \Sigma^*$ satisfies the following:

1. it is monotone: For all $v \in \Sigma^*$, $f(uv) = f(u)c'$ for some $c' \in \mathbf{2}^*$.
2. it computes code prefixes: For all $v \in \Sigma^*$ and matching derivations $\mathcal{D} :: (A, uv) \Rightarrow_P w$ ($w \in \Sigma^*$), we have $\mathcal{C}_{\mathcal{D}} = f(u)c'$ for some $c' \in \mathbf{2}^*$.
3. it completes the code: if there exists a matching derivation $\mathcal{D} :: (A, u) \Rightarrow_P w$, then $\mathcal{C}_{\mathcal{D}} = f(u\#)$.

In this paper, we develop an algorithm which implements a streaming parsing function as defined above. The code prefix produced allows consumers to perform parsing actions (e.g. construction of syntax trees, evaluation of expressions, printing, etc.) before all of the input string has been consumed. Monotonicity ensures that no actions will have to be "un-done", with the caveat that further input might cause the whole parse to be rejected.

3. Tabulation of Operational Semantics

In the following we fix a program $P = (\Sigma, V, S, R)$.

We will be working with various constructions defined as least fixed points of monotone operators on partially ordered sets. A partial order is a pair $(X, \sqsubseteq)$ where X is a set and $\sqsubseteq$ is a reflexive, transitive and antisymmetric relation on X. Given two elements $x, y \in X$, we will write $x \sqsubset y$ when $x \sqsubseteq y$ and $x \neq y$.

For any set X, let $X_\perp$ be $X \uplus \{\perp\}$ with the partial order $x \sqsubset y$ if and only if $x = \perp$. A *table* with X-entries is a $|P| \times \mathbb{N}_0$ matrix T where each entry T_{ij} is in $X_\perp$, and indices (i, j) are in the set $\mathsf{Index} = \{(i, j) \mid 0 \leq i < |P| \wedge 0 \leq j\}$. The set of all tables on X is denoted $\mathsf{Table}(X)$, and forms a partial order $(\mathsf{Table}(X), \sqsubseteq)$ by comparing entries pointwise: for $T, T' \in \mathsf{Table}(X)$, we write $T \sqsubseteq T'$ iff for all $(i, j) \in \mathsf{Index}$, we have $T_{ij} \sqsubseteq T'_{ij}$. Write $\perp \in \mathsf{Table}(X)$ for the table with all entries equal to $\perp \in X_\perp$. It is easy to verify that the partial order on $\mathsf{Table}(X)$ has the following structure:

complete partial order: For all chains $T_0 \sqsubseteq T_1 \sqsubseteq \ldots$ where $T_i \in \mathsf{Table}(X)$, $i \in \{0, 1, \ldots\}$, the least upper bound $\bigsqcup_i T_i$ exists.

meet-semilattice: For all non-empty subsets $S \subseteq \mathsf{Table}(X)$, the greatest lower bound $\bigsqcap S$ exists.

A function $F : \mathsf{Table}(X) \to \mathsf{Table}(X)$ is said to be *continuous* if it preserves least upper bounds of chains: For all $S \subseteq \mathsf{Table}(X)$, we have $F(\bigsqcup S) = \bigsqcup_{T \in S} F(T)$. A continuous function is *monotonicc*: $T \sqsubseteq T'$ implies $F(T) \sqsubseteq F(T')$. A *least fixed point* of F is an element T such that $F(T) = T$ (T is a fixed point) and also $T \sqsubseteq T'$ for all fixed points T'. A general property of complete partial orders is that if F is a continuous function then its least fixed point $\operatorname{lfp} F$ exists and is given by

$$\operatorname{lfp} F = \bigsqcup_n F^n(\perp)$$

where F^n is the n-fold composition of F with itself. We will also rely on the following generalization:

Lemma 3.1 (Lower bound iteration)**.** *If $T \sqsubseteq \operatorname{lfp} F$, then $\operatorname{lfp} F = \bigsqcup_n F^n(T)$.*

3.1 Parse Tables

We now recall the parse table used in the dynamic programming algorithm for linear time recognition [1], but presented here as a least fixed point. The table has entries from $\mathsf{Res} = \mathbb{N}_0 + \{\mathsf{f}\}$, that is, either a natural number or f indicating failure. Given a finite (respectively, infinite) string $w = a_0 a_1 \ldots a_{n-1}$ ($w = a_0 a_1 \ldots$), and an offset $0 \leq j < n$ ($0 \leq j$), write u_j for the suffix $a_j a_{j+1} \ldots a_{n-1}$ ($a_j a_{j+1} \ldots$) obtained by skipping the first j symbols.

Definition 5 (Parse table)**.** Let $u \in \Sigma^*$. Define a table operator F^u on $\mathsf{Table}(\mathsf{Res})$ as follows. Let $w = u\#^\omega$, the infinite string starting with u followed by an infinite number of repetitions of the end marker $\# \notin \Sigma$. For any table $T \in \mathsf{Table}(\mathsf{Res})$ define $F^u(T) = T'$ such that for all $(i, j) \in \mathsf{Index}$:

$$T'_{ij} = \begin{cases} \mathsf{f} & A_i \leftarrow \mathsf{f} \text{ or } A_i \leftarrow a \text{ and } a \text{ not a prefix of } w_j \\ 1 & A_i \leftarrow a \text{ and } a \text{ is a prefix of } w_j \\ 0 & A_i \leftarrow \epsilon \\ m + m' & A_i \leftarrow A_x[A_y, A_z] \wedge T_{xj} = m \wedge T_{y(j+m)} = m' \\ \mathsf{f} & A_i \leftarrow A_x[A_y, A_z] \wedge T_{xj} = m \wedge T_{y(j+m)} = \mathsf{f} \\ T_{zj} & A_i \leftarrow A_x[A_y, A_z] \wedge T_{xj} = \mathsf{f} \\ \perp & \text{otherwise} \end{cases}$$

The operator F^u is easily seen to be continuous, and we define the *parse table for u* by $T(u) = \operatorname{lfp} F^u$.

For any $u \in \Sigma^*$, the table $T(u)$ is a tabulation of all parsing results on all suffixes of u:

Theorem 3.2 (Fundamental theorem)**.** *Let $u \in \Sigma^*$ and consider $T(u)$ as defined above. For all $(i, j) \in \mathsf{Index}$:*

1. *$j \leq |u|$ and $T(u)_{ij} = \mathsf{f}$ iff $(A_i, u_j) \Rightarrow_P \mathsf{f}$; and*
2. *$j \leq |u|$ and $T(u)_{ij} = m \in \mathbb{N}_0$ iff $(A_i, u_j) \Rightarrow_P u_{j+m}$; and*
3. *$j \leq |u|$ and $T(u)_{ij} = \perp$ iff $(A_i, u_j) \not\Rightarrow_P$;*
4. *if $j > |u|$ then $T_{ij} = T_{i|u|}$*

The converse also holds: for any T satisfying the above, we have $T = T(u)$.

Property 4 is sufficient to ensure that all parse tables have a finitary representation of size $|P| \times |u|$. It is straightforward to extract a parse code from $T(u)$ by applying Definition 3 and the theorem.

Example 2. Consider the program P from Example 1. The tables $T = T(aa)$ and $T' = T(aaba)$ are shown below:

	0	1	2	···
	a	a	#	···
A	1	1	f	
B	f	f	f	
E	0	0	0	
F	f	f	f	
L	0	0	0	···
P	f	f	f	
R	2	1	0	
S	2	1	0	

	0	1	2	3	4	···
	a	a	b	a	#	···
A	1	1	f	1	f	
B	f	f	1	f	f	
E	0	0	0	0	0	
F	f	f	f	f	f	
L	3	2	1	0	0	···
P	3	2	1	f	f	
R	2	1	0	1	0	
S	4	3	2	1	0	

Note that columns 1,2 in the left table equals columns 3,4 in the right table. In general, columns depend on the corresponding input suffix but are independent of the previous columns. This is a simple consequence of Theorem 3.2.

For a table T and $m \in \mathbb{N}_0$, let $T[m]$ be the table obtained by removing the first m columns from T, i.e. $T[m]_{ij} = T_{i(j+m)}$.

Corollary 3.3 (Independence)**.** *Let $u \in \Sigma^*$. For all $0 \leq m$, we have $T(u)[m] = T(u_m)$.*

Proof. By Theorem 3.2. For example, if $T(u)_{i(j+m)} = m'$ for some m' then $(A_i, u_{j+m}) \Rightarrow u_{j+m+m'}$. Have $(u_m)_j = u_{m+j}$, so $(A_i, (u_m)_j) \Rightarrow (u_m)_{j+m'}$, and therefore $T(u_m)_{ij} = m'$. □

Independence leads to the linear-time $O(|P|^2 n)$ parsing algorithm of Aho and Ullman. For input u with $|u| = n$, compute $T(u)$ column by column, starting from the right. In each step $j \leq n$, compute column j by fixed point iteration of F^{u_j} on the current table state. Since $T(u)[j+1] = T(u_{j+1})$ is given by induction, only the $|P|$ entries in column j need to be processed. Naive fixed-point iteration will reach a fixed point in $O(|P|)$ substeps, each computing $|P|$ entries, leading to the constant factor $O(|P|^2)$.

4. Streaming Parsing with Tables

The linear-time parsing algorithm has asymptotically optimal time complexity. However, it always uses space linear in the length of the input string, since all columns of the parse table have to be computed before the final result can be obtained. For large grammars and inputs, this can be prohibitively expensive. In the following we describe a method for computing only an initial part of the table. The initial columns will in some cases provide enough information to construct a prefix of the parse code and allow us to continue parsing with a smaller table, saving space.

Let us illustrate the idea by an example. Let $w = uv$ be an input string, and let $A_i \leftarrow A_x[A_y, A_z]$ be a rule in the program. Suppose that by analyzing only the prefix u, we can conclude that there is a constant m such that $T(uv')_{x0} = m$ for all v'. In particular, this holds for $v' = v$, so $T(w)_{i0} \in \mathbb{N}_0$ if and only if $T(w)_{i0} = m + m'$ where $m' = T(w)_{ym} = T(w)[m]_{y0} = T(w_m)_{y0}$ (the last equation follows by independence). By examining only the prefix u, we have thus determined that the result only depends on $T(w_m)$, freeing up m columns of table space. The process can be repeated for the remaining input w_m.

We will need an analysis that can predict results as described. The theoretically optimal analysis is defined as follows:

Definition 6 (Optimal prefix table)**.** Let $u \in \Sigma^*$, and define the *optimal prefix table* $T^{\sqcap}(u) \in \mathsf{Table}(\mathsf{Res})$ as the largest approximation of all the complete tables for all extensions of u:

$$T^{\sqcap}(u) = \bigsqcap_{v \in \Sigma^*} T(uv)$$

Theorem 4.1. *For all u, i, j:*

1. *if $T^{\sqcap}(u)_{ij} \neq \bot$ then $\forall v.\, T(uv)_{ij} = T^{\sqcap}(u)_{ij}$;*
2. *if $(\forall v.\, T(uv)_{ij} = r \neq \bot)$, then $T^{\sqcap}(u)_{ij} = r$.*

Unfortunately, we cannot use this for parsing, as the optimal prefix table is too precise to be computable:

Theorem 4.2. *There is no procedure which computes $T^{\sqcap}(u)$ for all GTDPLs P and input prefixes u.*

Proof. Assume otherwise that $T^{\sqcap}(u)$ is computable for any u and GTDPL P. Then $L(P) = \emptyset$ iff $T^{\sqcap}(\varepsilon)_{i_S,0} = \mathsf{f}$. Hence emptiness is decidable, a contradiction by Proposition 2.3. □

A conservative and computable approximation of $T^{\sqcap}$ can easily be defined as a least fixed point. Given a table operator F and a subset $J \subseteq \mathsf{Index}$ define a restricted operator F_J by

$$F_J(T)_{ij} = \begin{cases} F(T)_{ij} & \text{if } (i,j) \in J \\ T_{ij} & \text{otherwise} \end{cases}$$

If $J = \{(p,q)\}$ is a singleton, write F_{pq} for F_J. Clearly, if F is continuous then so is F_J.

For any $u \in \Sigma^*$, define an operator $F^{(u)}$ by $F^{(u)} = F^u_{J_u}$ where $J_u = \{(i,j) \in \mathsf{Index} \mid j < |u|\}$. The *prefix table* for u is the least fixed point of this operator:

$$T^{<}(u) = \operatorname{lfp} F^{(u)}$$

Intuitively, a prefix table contains as much information as can be determined without depending on column $|u|$. Prefix tables are clearly computable by virtue of being least fixed points, and properly approximate the optimal analysis:

Theorem 4.3 (Approximation)**.** *For all $u \in \Sigma^*$, we have $T^{<}(u) \sqsubseteq T^{\sqcap}(u)$. In particular, if $T^{<}(u)_{ij} = m$ or $T^{<}(u)_{ij} = \mathsf{f}$, then $\forall v.\, T(uv)_{ij} = m$ or $\forall v.\, T(uv)_{ij} = \mathsf{f}$, respectively.*

Prefix tables become better approximations as the input prefix is extended:

Proposition 4.4 (Prefix monotonicity)**.** *For all $u, v \in \Sigma^*$, we have $T^{<}(u) \sqsubseteq T^{<}(uv)$.*

We will make use of this property and Lemma 3.1 to efficiently compute prefix tables in an incremental fashion.

The full parse table can be recovered as a prefix table if we just append an explicit end marker to the input string:

Proposition 4.5 (End marker)**.** *For all $u \in \Sigma^*$ and $(i,j) \in \mathsf{Index}$, if $j \leq |u|$ then $T^{<}(u\#)_{ij} = T(u)_{ij}$.*

Independence carries over to prefix tables. For all $u \in \Sigma^*$ and $m \geq 0$, we thus have $T^{<}(u)[m] = T^{<}(u_m)$.

4.1 Streaming Code Construction

The resolved entries of a prefix table can be used to guide a partial leftmost expansion of a parse tree. We model this expansion process by a labeled transition system which generates the corresponding parse code. By constructing the expansion such that it is a prefix of all viable expansions, the parse code can be computed in a streaming fashion. In order to determine as much of the parse code as possible, we can commit to the continuation branch when a dynamic analysis determines that the failure branch must fail.

Definition 7 (Leftmost parse tree expansion)**.** Let $T \in \mathsf{Table}(\mathsf{Res})$ be a table and $d \in \mathbb{N}_0$ a *speculation constant*. Define a labeled transition system $\mathcal{E}_T = (Q, E)$ with states $Q = V^* \times \mathbb{N}_0$ and transitions $E \subseteq \{q \xrightarrow{c} q' \mid c \in \mathbf{2}^*; q, q' \in Q\}$. Let E be the smallest set such that for all $A_i \in V$, $\vec{K} \in V^*$ and $j \in \mathbb{N}_0$:

1. If $A_i \leftarrow A_x[A_y, A_z]$; and either $T_{xj} \in \mathbb{N}_0$ or $(A_z\vec{K}, j)$ **fails**$_d$, then:
$$(A_i\vec{K}, j) \xrightarrow{0} (A_xA_y\vec{K}, j) \in E$$
2. If $A_i \leftarrow A_x[A_y, A_z]$; and $T_{xj} = \mathsf{f}$, then:
$$(A_i\vec{K}, j) \xrightarrow{1} (A_z\vec{K}, j) \in E$$
3. If $A_i \leftarrow \epsilon$ or $A_i \leftarrow a$; and $T_{ij} = m$, then:
$$(A_i\vec{K}, j) \xrightarrow{\varepsilon} (\vec{K}, j + m) \in E$$
4. If $q \xrightarrow{c} q' \in E$ and $q' \xrightarrow{c'} q''$, then: $q \xrightarrow{cc'} q'' \in E$.

where for all $\vec{K}, j, n$, write $(\vec{K}, j)$ **fails**$_n$ if $\vec{K} = A_i\vec{K}'$ and either

1. $T_{ij} = \mathsf{f}$; or
2. $T_{ij} = m, n = n' + 1$ and $(\vec{K}', j + m)$ **fails**$_{n'}$.

A state encodes the input offset and the stack of leaves that remain unexpanded. The node on the top of the stack is expanded upon a transition to the next state, with the expansion choice indicated in the label of the transition. The system is deterministic in the sense that every state can step to at most one other state in a single step (the label is determined by the source state).

The highlighted disjunct allows us to speculatively resolve a choice as succeeding when the failure branch is guaranteed to fail. This is determined by examining the table entries for at most d nonterminals on the current stack $\vec{K}$.

Example 3. The partial parse tree of Example 1 corresponds to the following steps in $\mathcal{E}_{T'}$ where T' is the table from Example 2:
$$(S, 0) \xrightarrow{0} (LR, 0) \xrightarrow{0} (PER, 0) \xrightarrow{0} (APER, 0) \xrightarrow{\varepsilon} (PER, 1)$$

A state q is *quiescent* if there is no transition from it. Say that q is *convergent* and write $q \xrightarrow{c} q' \downarrow$ if either there is a path $q \xrightarrow{c} q'$ such that q' quiescent; or, q is already quiescent and $q' = q$ and $c = \varepsilon$. Clearly, if such c and q' exists, then they are unique and can be effectively determined. Otherwise, we say that q is *divergent*.

Expansions compute coded (matching) derivations in full parse tables:

Proposition 4.6. *Let $u \in \Sigma^*$ and consider the system $\mathcal{E}_{T(u)}$.*

1. *There is a derivation $\mathcal{D} :: (A, u) \Rightarrow_P u_m$ with $c = \mathcal{C}_\mathcal{D}$ if and only if $(A, 0) \xrightarrow{c} (\varepsilon, m) \downarrow$.*
2. *We have $(A, u) \Rightarrow_P \mathsf{f}$ if and only if $(A, 0)$ **fails**$_n$ for some n.*

It follows that a state $(A, 0)$ is only divergent if the input is unhandled:

Proposition 4.7. *Let $u \in \Sigma^*$ and consider the system $\mathcal{E}_{T(u)}$. Then $(A, u) \not\Rightarrow_P$ if and only if $(A, 0)$ is divergent.*

Hence, if P is complete, then every state is convergent in $\mathcal{E}_{T(u)}$, and the relation $q \xrightarrow{c} q' \downarrow$ becomes a total function $q \mapsto (c, q')$.

The function associating every input prefix u with the code c given by $(S, 0) \xrightarrow{c} q' \downarrow$ in the system $\mathcal{E}_{T^<(u)}$ is a streaming parse function as per Definition 4. This is ensured by the following sufficient condition, which states that expansions never "change direction" as the underlying table is refined:

Proposition 4.8. *If $T \sqsubseteq T'$ and $(\vec{K}, j) \xrightarrow{c} (\vec{K}', j')$ is in $\mathcal{E}_T$, then either $(\vec{K}, j) \xrightarrow{c} (\vec{K}', j')$ is in $\mathcal{E}_{T'}$ or $(\vec{K}, j)$ fails in $\mathcal{E}_{T'}$.*

Expansions also never backtrack in the input, that is, if $(\vec{K}, j) \xrightarrow{c} (\vec{K}', j')$ then $j \leq j'$. This allows us to discard the initial columns of a table as we derive a leftmost expansion:

Proposition 4.9. *Let T be a table. Then $(\vec{K}, m) \xrightarrow{c} (\vec{K}', n)$ in $\mathcal{E}_T$ if and only if $(\vec{K}, 0) \xrightarrow{c} (\vec{K}', n - m)$ in $\mathcal{E}_{T[m]}$.*

4.2 Progressive Tabular Parsing

Assume that P is a complete program. We use the constructions of this section to define our *progressive tabular parsing* procedure. The algorithmic issues of space and time complexity will not be of our concern yet, but will we be adressed in the following section.

Given an input string with end marker $w\# = a_0a_1...a_{n-1}$ ($a_{n-1} = \#$), the procedure decides whether there exists a matching derivation $\mathcal{D} :: (S, w) \Rightarrow_P w_k$, and in that case produces $\mathcal{C}_\mathcal{D}$ in a streaming fashion. In each step $0 \leq k \leq n$, we compute a table $T^k \in \mathsf{Table}(\mathsf{Res})$, a stack $\vec{K}^k$, an offset $m^k \leq k$ and a code chunk $c^k \in \mathbf{2}^*$. Upon termination, we will have $\mathcal{C}_\mathcal{D} = c^0c^1...c^n$.

Initially $T^0 = T^<(\varepsilon)$, $\vec{K}^0 = S$, $m^0 = 0$ and $c^0 = \varepsilon$. For each $1 \leq k \leq n$, the values $T^k, \vec{K}^k, m^k$ and c^k are obtained by
$$T^k = T^<(a_{m^{k-1}}...a_{k-1})$$
$$m^k = m^{k-1} + m' \text{ where } (\vec{K}^{k-1}, 0) \xrightarrow{c^k} (\vec{K}^k, m') \downarrow$$
Since P is complete, we have by Proposition 4.7 that the last line above can be resolved. If $\vec{K}^n = \varepsilon$, then accept; otherwise reject.

Theorem 4.10. *The procedure computes $\mathcal{C}_\mathcal{D}$ iff there is a derivation $\mathcal{D} :: (S, w) \Rightarrow_P w_k$.*

Proof. We claim that after each step k, we have $(S, 0) \xrightarrow{c^0...c^k} (\vec{K}^k, m^k) \downarrow$ in $\mathcal{E}_{T(w)}$. This holds for $k = 0$, as $(S, 0)$ is quiescent. For $k > 0$, we assume that it holds for $k - 1$ and must show $(\vec{K}^{k-1}, m^{k-1}) \xrightarrow{c^k} (\vec{K}^k, m^k) \downarrow$ in $\mathcal{E}_{T(w)}$. By construction, we have a path $(\vec{K}^{k-1}, 0) \xrightarrow{c^k} (\vec{K}^k, m^k - m^{k-1})$ in $\mathcal{E}_{T^k}$. By Proposition 4.4 and Theorem 4.3, we have $T^k = T^<(a_{m^{k-1}}...a_{k-1}) \sqsubseteq T^<(w_{m^{k-1}}) \sqsubseteq T(w_{m^{k-1}}) = T(w)[m^k - 1]$, so by Proposition 4.8, the path is in $\mathcal{E}_{T(w)[m^{k-1}]}$, and by Proposition 4.9, we obtain our subgoal.

If the procedure accepts the input, then we are done by Proposition 4.6. If it rejects, it suffices to show that $(\vec{K}, m^k)$ is quiescent in $\mathcal{E}_{T(w)}$ which by Proposition 4.6 implies that there is no matching derivation. Since $T^m = T^<(w\#)$, we can apply Proposition 4.5 to easily show $\mathcal{E}_{T^m} = \mathcal{E}_{T(w)}$, and we are done. ☐

Figure 1 shows an example of a few iterations of the procedure applied to the program in Example 1.

In the next section we show that the above procedure can be performed using linear time and space. Linear space is obtained by observing that the table T^{k-1} is no longer needed once T^k has been computed. On the other hand, a linear time guarantee requires careful design: Computing each table T^k using the classical right-to-left algorithm would take linear time in each step, and hence quadratic time in total. In the following section, we show how to obtain the desired time complexity by computing each table incrementally from the previous one.

5. Algorithm

The streaming parsing procedure of Section 4.2 can be performed in amortized time $O(|w|)$ (treating the program size as a constant). We assume that the program P is complete.

Our algorithm computes each prefix table T^k using a work set algorithm for computing fixed points. We save work by starting the computation from $T^{k-1}[m^{k-1}]$ instead of the empty table $\perp$. In order to avoid unnecessary processing, an auxiliary data structure is used to determine exactly those entries which have enough information available to be resolved. This structure itself can be maintained in constant time per step. Since at most $O(|w|)$ unique entries need to be resolved over the course of parsing w, this is also the time complexity of the algorithm.

(1)	a
A	1
B	f
E	**0**
F	f
L	$\bot$
P	$\bot$
R	$\bot$
S	$\bot$

(2)	a	a
A	1	1
B	f	f
E	**0**	0
F	f	f
L	$\bot$	$\bot$
P	$\bot$	$\bot$
R	$\bot$	$\bot$
S	$\bot$	$\bot$

(3)	a	a	b	
A	**1**	**1**	f	$\bot$
B	f	f	**1**	$\bot$
E	**0**	0	0	$\bot$
F	f	f	f	$\bot$
L	$\bot$	$\bot$	$\bot$	$\bot$
P	**3**	2	1	$\bot$
R	2	1	0	$\bot$
S	$\bot$	$\bot$	$\bot$	$\bot$

(4)	a	a	b	a	
A	**1**	**1**	f	**1**	$\bot$
B	f	f	**1**	f	$\bot$
E	**0**	0	0	**0**	$\bot$
F	f	f	f	f	$\bot$
L	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
P	**3**	2	1	$\bot$	$\bot$
R	2	1	0	$\bot$	$\bot$
S	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

(5)	a	a	b	a	#
A	**1**	**1**	f	**1**	f
B	f	f	**1**	f	f
E	**0**	0	0	**0**	**0**
F	f	f	f	f	f
L	$\bot$	$\bot$	$\bot$	$\bot$	0
P	**3**	2	1	$\bot$	f
R	2	1	0	$\bot$	0
S	$\bot$	$\bot$	$\bot$	$\bot$	0

(1) : $(S, aaba\#) \xrightarrow{0} (LR, aaba\#)$
(2) : $(LR, aaba\#)$
(3) : $(LR, aaba\#) \xrightarrow{0} (PER, aaba\#) \xrightarrow{0} (APER, aaba\#) \rightarrow (PER, aba\#) \xrightarrow{0} (APER, aba\#) \rightarrow (PER, ba\#) \xrightarrow{1} (BER, ba\#) \rightarrow (ER, a\#)$
(4) : $(ER, a\#) \rightarrow (R, a\#) \xrightarrow{0} (AR, a\#) \rightarrow (R, \#)$
(5) : $(R, \#) \xrightarrow{1} (E, \#) \rightarrow (\varepsilon, \#)$

Figure 1. At the top are the five consecutive tables constructed during parsing of input $w = aaba$, using the program from Example 1. Only the columns to the right of the dashed line have to be stored for the next iteration. Newly computed entries are colored; entries considered by the expansion process are written in bold face. The progression of the leftmost expansion is shown below.

Algorithm 1 (PARSE).
In: $w = a_0 a_1 ... a_{|w|-1} \in \Sigma^* \#$.
Out: Code $c^0 c^1 ... c^{|w|-1}$, accept/reject.

```
1:  $u := \varepsilon$
2:  $T := \bot$
3:  $\vec{K} := S$
4:  $R := (i, j) \mapsto \emptyset$
5:  for $k \in \{1, ..., |w|\}$ do
6:      $u := u\, a_{k-1}$
7:      run FIX
8:      compute $(\vec{K}, 0) \xrightarrow{c} (\vec{K}', m') \downarrow$
9:      $c^n := c$
10:     $\vec{K} := \vec{K}'$
11:     $T := T[m']$
12:     $R := R[m']$
13: accept if $\vec{K} = \varepsilon$ else reject
```

Algorithm 2 (FIX).
Precondition:
$u = a_m ... a_{k-1}$
$T = T^{<}(a_m ... a_{k-2}) \wedge R = (D^T)^{-1}$
Postcondition:
$u = a_m ... a_{k-1}$
$T = T^{<}(u) \wedge R = (D^T)^{-1}$

```
1:  $W := \{(i, |u| - 1) \mid g_i \text{ simple}\}$
2:  while $W \neq \emptyset$ do
3:      let $(p, q) \in W$
4:      $T := F^{(u)}_{pq}(T)$
5:      $W := W \setminus \{(p, q)\} \cup R_{pq}$
6:      for $i' \in C^{-1}_p$ do
7:          let $(k, \ell) = D^T_{i'q}$
8:          $R_{k\ell} := R_{k\ell} \cup \{(i', q)\}$
9:          if $T_{k\ell} \neq \bot$ then
10:             $W := W \cup \{(i', q)\}$
```

Reverse condition map

$$C^{-1} : |P| \rightarrow \mathbf{2}^{|P|}$$
$$C^{-1}_x = \{i \in |P| \mid A_i \leftarrow A_x[A_y, A_z]\}$$

Dynamic (reverse) dependency map

$$D : \mathsf{Table} \times \mathsf{Index} \rightarrow \mathsf{Index}_\bot$$
$$D^T_{ij} = \begin{cases} (y, j+m) & \text{if } A_i \leftarrow A_x[A_y, A_z] \wedge T_{xj} = m \\ (z, j) & \text{if } A_i \leftarrow A_x[A_y, A_z] \wedge T_{xj} = \mathsf{f} \\ \bot & \text{otherwise} \end{cases}$$
$$(D^T)^{-1}_{k\ell} = \{(i, j) \mid D^T_{ij} = (k, \ell)\}$$

Restrictions

$$T[m]_{ij} = T_{i(m+j)}$$
$$R[m]_{k\ell} = \{(i, j-m) \mid (i, j) \in R_{k(m+\ell)} \wedge j \geq m\}$$

Figure 2. Parsing algorithm.

The algorithm is presented in two parts in Figure 2. Algorithm 1 (PARSE) takes as input a #-terminated input stream and maintains two structures: A table structure T which incrementally gets updated to represent $T^{<}(u)$ for a varying substring $u = a_{m^{k-1}} ... a_{k-1}$; and a map R which keeps track of reverse data dependencies between the entries in T. In each iteration, any resolved code prefix is returned and the corresponding table columns freed. The main work is done in Algorithm 2 (FIX) which updates T and R to represent the next prefix table and its reverse dependencies, respectively.

5.1 Work Sets

Let T be a table such that $T \sqsubseteq T^{<}(u)$ for some prefix u. The *work set* $\Delta_u(T) \subseteq \mathsf{Index}$ consists of all indices of entries that can be updated to bring T closer to $T^{<}(u)$ by applying $F^{(u)}$:

$$\Delta_u(T) = \{(i, j) \mid T_{ij} \sqsubset F^{(u)}(T)_{ij}\}.$$

It should be clear that $T = T^{<}(u)$ iff $\Delta_u(T) = \emptyset$, and that for all $(p, q) \in \Delta_u(T)$, we still have $F^{(u)}_{pq}(T) \sqsubseteq T^{<}(u)$ for the updated table. In the following we show how $\Delta_u(F^{(u)}_{pq}(T))$ can be obtained from $\Delta_u(T)$ instead of recomputing it from scratch.

5.2 Dependencies

In order to determine the effect of table updates on the work set, we need to make some observations about the dependencies between table entries.

Consider an index (i, j) such that $A_i \leftarrow A_x[A_y, A_z]$ and $T_{ij} = \bot$. The index (i, j) cannot be in the work set for T unless either $T_{xj} = m$ and $T_{y(j+m)} \neq \bot$; or $T_{xj} = \mathsf{f}$ and $T_{zj} \neq \bot$. We say that (i, j) *conditions* on (x, j). The reverse condition map C^{-1} in Figure 2 associates every row index x with the set of row indices $i \in C^{-1}_x$ such that (i, j) conditions on (x, j) for all j.

If $T_{xj} = m$ or $T_{xj} = \mathsf{f}$ then (i, j) is in the work set iff $T_{y(j+m)} \neq \bot$ or $T_{zj} \neq \bot$, respectively. In either case, we say that (i, j) has a *dynamic dependency* on $(y, j+m)$ or (z, j), respectively. The dependency is *dynamic* since it varies based on the value of T_{xj}. The partial map $D : \mathsf{Table(Res)} \times \mathsf{Index} \rightarrow \mathsf{Index}_\bot$ defined in Figure 2 associates every index (i, j) with its unique dynamic dependency D^T_{ij} in table T. The dynamic dependency is undefined ($\bot$) if the condition is unresolved or if the corresponding expression g_i is simple (recall that *complex expressions* are of the form $A[B, C]$, and any other expression is *simple*).

By the observations above, we can reformulate the work set using dependencies:

Lemma 5.1 (Work set characterization)**.** *For all T we have*

$$\Delta_u(T) = \{(i,j) \in J_u \mid T_{ij} = \bot \\ \wedge (g_i \text{ complex} \Rightarrow D^T_{ij} \neq \bot \neq T_{D^T_{ij}})\}$$

5.3 Incremental Work Set Computation

When a table S is updated by computing $T = F^{(u)}_{pq}(S)$ for $(p,q) \in \Delta_u(S)$, Lemma 5.1 tells us that the changes to the work set can be characterized by considering the entries (i,j) for which one or more of the values D^T_{ij} and $T_{D^T_{ij}}$ differ from D^S_{ij} and $S_{D^S_{ij}}$, respectively.

The dependency map gets more defined as we go from S to T:

Lemma 5.2 (Dependency monotonicity)**.** *If $T \sqsubseteq T'$, then for all $(i,j) \in \mathsf{Index}$, we have $D^T_{ij} \sqsubseteq D^{T'}_{ij}$.*

Using this and the fact that $S \sqsubseteq T$, it is easy to show that we must have $\Delta_u(T) \supseteq \Delta_u(S) \setminus \{(p,q)\}$. Furthermore, we observe that $(i,j) \in \Delta_u(T) \setminus (\Delta_u(S) \setminus \{(p,q)\})$ iff

1. $D^S_{ij} \sqsubset D^T_{ij}$ and $T_{D^T_{ij}} \neq \bot$; or
2. $D^S_{ij} = D^T_{ij} \neq \bot$ and $S_{D^S_{ij}} \sqsubset T_{D^T_{ij}}$.

Since the second case can only be satisfied when $D^T_{ij} = (p,q)$, it is completely characterized by the reverse dependency set $(D^T)^{-1}_{pq}$, defined in Figure 2. The first case is when (i,j) conditions on (p,q) (equivalent to $D^S_{ij} \sqsubset D^T_{ij}$) and $T_{D^T_{ij}} \neq \bot$. The entries satisfying the former are completely characterized by the reverse condition map:

Lemma 5.3 (Dependency difference)**.** *Let $S \in \mathsf{Table}(\mathsf{Res})$ such that $S \sqsubseteq T^<(u)$ and $(p,q) \in \Delta_u(S)$, and define $T = F^{(u)}_{pq}(S)$. Then $\{(i,j) \mid D^S_{ij} \sqsubset D^T_{ij}\} = C^{-1}_p \times \{q\}$.*

By Lemmas 5.1, 5.2 and 5.3, we obtain the following incremental characterization of the work set:

Lemma 5.4 (Work set update)**.** *Let $S \sqsubseteq F^{(u)}(S) \sqsubseteq T^<(u)$, $(p,q) \in \Delta_u(S)$ and $T = F^{(u)}_{pq}(S)$. Then*

$$\begin{aligned}\Delta_u(T) = \Delta_u(S) &\setminus \{(p,q)\} \\ &\cup (D^S)^{-1}_{pq} \\ &\cup \{(i',q) \mid i' \in C^{-1}_p \wedge \bot \neq T_{D^T_{i'q}}\}\end{aligned}$$

The extra premise $S \sqsubseteq F^{(u)}(S)$ says that every entry in S must be a consequence of the rules encoded by $F^{(u)}$, and can easily be shown to be an invariant of our algorithm.

Reverse dependency map lookups $(D^T)^{-1}_{pq}$ cannot easily be computed efficiently. To accommodate efficient evaluation of these lookups, the algorithm maintains a data structure R to represent $(D^T)^{-1}$. The following lemma shows that the loop 6-10 in FIX will reestablish the invariant that $R = (D^T)^{-1}$:

Lemma 5.5 (Dependency update)**.** *Let $S \sqsubseteq T^<(u)$, $(p,q) \in \Delta_u(S)$ and $T = F^{(u)}_{pq}(S)$. Then for all $(k,\ell) \in \mathsf{Index}$, we have $(D^T)^{-1}_{k\ell} = (D^S)^{-1}_{k\ell} \cup \{(i',q) \mid i' \in C^{-1}_p \wedge (k,\ell) = D^T_{i'q}\}$.*

5.4 Correctness

Theorem 5.6 (Correctness of FIX)**.** *Let* FIX *be the algorithm in Figure 2. If the precondition of* FIX *holds, then the postcondition holds upon termination.*

Proof sketch. We first remark that the algorithm never attempts to perform an undefined action. It suffices to check that line 3 is always well-defined, and that Lemma 5.3 implies that the right of the equation in line 7 is always resolved.

The outer loop maintains that $R = (D^T)^{-1}$ and $W = \Delta_u(T)$. Initially, only the entries in the last column which are associated with simple expressions can be updated. If S is the state of T at the beginning of an iteration of loop 2-10, then at the end of the iteration T will have the form of the right hand side of Lemma 5.4. When the loop terminates we have $W = \Delta_u(T) = \emptyset$, so $T = T^<(u)$. □

Theorem 5.7 (Correctness of PARSE)**.** *The algorithm* PARSE *performs the streaming parsing procedure of Section 4.2.*

Proof sketch. After executing lines 1-4, we verify that $R = (D^T)^{-1}$, and that for $k = 0$:

$$T = T^<(a_{m^k}...a_{k-1}), \qquad \vec{K} = \vec{K}^k, \qquad u = a_{m^k}...a_{k-1}$$

The loop maintains the invariant: When entering the loop, we increment k and thus have $R = (D^T)^{-1}$ and

$$T = T^<(a_{m^{k-1}}...a_{k-2}), \quad \vec{K} = \vec{K}^{k-1}, \quad u = a_{m^{k-1}...a_{k-2}}$$

After the assignment to u, we have $u = a_{m^{k-1}}...a_{k-1}$. By running FIX, we then obtain $T = T^<(a_{m^{k-1}}...a_{k-1}) = T^k$. By assumption that P is complete, line 8 is computable, and we obtain

$$\vec{K}' = \vec{K}^k \qquad c = c^k \qquad m' = m^k - m^{k-1}$$

The last updates in the loop thus reestablishes the invariant. □

5.5 Complexity

We give an informal argument for the linear time complexity. Let $d \in \mathbb{N}_0$ be the constant from Definition 7 limiting the number of stack symbols considered when resolving choices.

It can be shown that the three sets on the right hand side of the equation in Lemma 5.4 are pairwise disjoint; likewise for Lemma 5.5. We thus never add the same element twice to W and R, meaning that they can be represented using list data structures, ensuring that all single-element operations are constant time.

The complexity argument is a simple aggregate analysis. To see that PARSE runs in linear time, we observe that the work set invariant ensures that we execute at most $O(|u|)$ iterations of the loop 2-10 in FIX. We add only unprocessed elements to the work list, and no element is added twice, so the total number of append operations performed in lines 5 and 10 is also $O(|u|)$. The same reasoning applies for the total number of append operations in line 8. The remaining operations in FIX are constant time.

Line 8 in PARSE computes an expansion of aggregate length $O(mn)$. For each expansion transition, we use at most d steps to resolve choices, and we thus obtain a bound of $O(dmn)$.

The restriction operator $T[m]$ can be performed in constant time by moving a pointer. The restriction of the reverse dependency map $R[m]$ can be implemented in constant time by storing the offset and lazily performing the offset calculation $j - m$ and filtering by $j \leq m$ on lookup.

5.6 Optimization

The algorithm presented above is simple, but as our evaluation will show, it may compute a large amount of table entries which are never actually needed. We can avoid this extra work by integrating the table computation with the expansion process, such that only those entries that are "forced" by the expansion process are added to the work set.

Figure 3 shows an optimized version of the algorithm operating in this fashion. The reverse condition and dependency maps C^{-1} and R are replaced by the maps R_1 and R_2. If we run the original and optimized algorithms in parallel, then after each iteration of the main loop, we have for all $(p,q) \in \mathsf{Index}$ that $R^1_{pq} \subseteq C^{-1}_p$ and $R^2_{pq} \subseteq R_{pq}$. A subset $Forced \subseteq \mathsf{Index}$ maintains the set of forced entries. An entry is forced by calling FORCE, which in turn forces its dependencies and updates the maps R^1 and R^2 accordingly. If an entry which can be resolved immediately is forced, then it is added to the work set. The algorithm implicitly assumes that the leftmost expansion calls $\textsc{Force}(i, j)$ followed by $\textsc{Fix}$ whenever it needs the value of an entry T_{ij}.

Correctness follows by verifying that the algorithm computes correct values for all entries and that all forced entries are resolved when the original unoptimized algorithm would also resolve them.

6. Evaluation

We have developed a simple prototype implementation for the purpose of measuring how the number of columns grow and shrink as the parser proceeds, which gives an indication of both its memory usage and its ability to resolve choices. The evaluation also reveals parts of the design which will require further engineering in order to obtain an efficient implementation. We have not yet developed an implementation optimized for speed, so a performance comparison with other tools is reserved for future work.

We consider three programs: a) a simplified JSON parser, b) a simplified parser for the fragment of statements and arithmetic expressions of a toy programming language, c) a tail-recursive program demonstrating a pathological worst-case.

All programs are presented as PEGs for readability. Nonterminals are underlined, terminals are written in `typewriter` and a character class [a...z] is short for a/b/.../z.

JSON Parser We wrote a JSON parser based on a simplification of the ECMA 404 specification (see `http://json.org`) and taking advantage of the repetition operator of PEG. To keep the presentation uncluttered, we have left out handling of whitespace:

$$\begin{array}{rcl}
\underline{object} & \leftarrow & \texttt{\{}\,\underline{members}\,\texttt{\}} \\
\underline{members} & \leftarrow & \underline{pair}(\texttt{,}\,\underline{pair})^*/\epsilon \\
\underline{pair} & \leftarrow & \underline{string}\,\texttt{:}\,\underline{value} \\
\underline{array} & \leftarrow & \texttt{[}\,\underline{elements}\,\texttt{]} \\
\underline{elements} & \leftarrow & \underline{value}(\texttt{,}\,\underline{value})^*/\epsilon \\
\underline{value} & \leftarrow & \underline{string}\,/\,\underline{object}\,/\,\underline{number}\,/\,\underline{array} \\
 & & /\texttt{true}/\texttt{false}/\texttt{null} \\
\underline{string} & \leftarrow & \texttt{"}[\texttt{a}...\texttt{z}]^*\texttt{"} \\
\underline{number} & \leftarrow & \underline{int}(\underline{frac}\,/\epsilon)(\underline{exp}\,/\epsilon) \\
\underline{int} & \leftarrow & [\texttt{1}...\texttt{9}]\,\underline{digits}\,/\texttt{-}[\texttt{1}...\texttt{9}]\,\underline{digits}\,/\texttt{-}[\texttt{0}...\texttt{9}]/[\texttt{0}...\texttt{9}] \\
\underline{frac} & \leftarrow & \texttt{.}\,\underline{digits} \\
\underline{exp} & \leftarrow & \underline{e}\,\underline{digits} \\
\underline{digits} & \leftarrow & [\texttt{0}...\texttt{9}][\texttt{0}...\texttt{9}]^* \\
\underline{e} & \leftarrow & \texttt{e+}/\texttt{e-}/\texttt{e}/\texttt{E+}/\texttt{E-}/\texttt{E}
\end{array}$$

The desugared program contains 158 rules.

We ran the program on a 364 byte JSON input with several nesting levels and syntactic constructs exercising all rules. The resulting parse code is computed in 3530 expansion steps. We would like to get an idea of how varying values of the speculation constant d affects the amount of memory consumed and the amount of work performed. Recall that d specifies the number of stack symbols considered when determining whether a branch must succeed on all viable expansions. The results are summarized in the following table (results stabilize after $d = 12$):

d	max cols	complex entries	spec. steps
	(max 365)	(max 39785)	(rel. to 3530)
0	362	39773 (99.97%)	0 (0.00%)
2	229	39673 (99.72%)	9 (0.25%)
4	10	35746 (89.85%)	283 (8.02%)
6	10	35625 (89.54%)	312 (8.84%)
8	2	35361 (88.88%)	419 (11.87%)
10	2	35346 (88.84%)	442 (12.52%)
12	2	35214 (88.51%)	453 (12.83%)

The second column shows the maximum number of columns stored at any point. The worst case is 365, which corresponds to storing every column for each of the 364 input symbols, and the column for the end of input marker. We observe that $d = 8$ results in at most two columns needing to be stored in memory.

The third column measures the potential work saved as d is increased by counting the total number of computed entries for complex expressions. We exclude entries for simple expressions since they are resolved immediately upon reading the next input symbol, and can hence be precomputed in an actual implementation.

The fourth column is the number of steps spent evaluating the $(\vec{K}, j)$ $\textbf{fails}_n$ predicate, and the relative number compared to the number of expansion steps. For this particular program, the overhead is seen to be very small compared to the reduction in computed entries and practically constant memory usage.

Memory usage goes from linear to constant when the speculation constant is increased. On the other hand, the amount of work that is saved on that account is modest, and the algorithm on average computes $35214/365 = 96.5$ complex entries per input symbol, which is 88.5% of the full table. However, most entries are never actually needed by the expansion process. The following table shows the same program executed with the optimized algorithm:

d	max cols	complex entries	spec. steps
	(max 365)	(max 39785)	(rel. to 3530)
0	362	4173 (10.49%)	0 (0.00%)
2	229	4183 (10.51%)	9 (0.25%)
4	10	3016 (7.58%)	283 (8.02%)
6	10	3008 (7.56%)	312 (8.84%)
8	2	3129 (7.86%)	419 (11.87%)
10	2	3127 (7.86%)	442 (12.52%)
12	2	3077 (7.73%)	453 (12.83%)

The amount of work is now reduced by an order of magnitude, and we compute on average $3077/365 = 8.4$ complex entries per input symbol.

We also scaled up the input to around 10KiB using repeated nesting, and ran the optimized algorithm with bound $d = 12$. The average number of computed complex entries per input symbol stayed the same at around 8.5, and so did the number of speculation steps relative to the number of expansion steps (12.75%).

Statement/Expression Parser The following is inspired by an example from a paper on ALL(*) [19]. The program parses a sequence of statements, each terminated by semicolon, with the whole sequence terminated by a single dot representing an end-of-program token. Each statement is either a single arithmetic expression or an assignment.

$$\begin{array}{rcl}
\underline{prog} & \leftarrow & \underline{stat}\,\underline{stat}^*\ \texttt{.} \\
\underline{stat} & \leftarrow & \underline{sum}\,\texttt{=}\,\underline{sum}\,\texttt{;}\,/\,\underline{sum}\,\texttt{;} \\
\underline{sum} & \leftarrow & \underline{product}\,\texttt{+}\,\underline{sum}\,/\,\underline{product} \\
\underline{product} & \leftarrow & \underline{factor}\,\texttt{*}\,\underline{product}\,/\,\underline{factor} \\
\underline{factor} & \leftarrow & \underline{id}\,\texttt{(}\,\underline{sum}\,\texttt{)}/\texttt{(}\,\underline{sum}\,\texttt{)}/\,\underline{id} \\
\underline{id} & \leftarrow & [\texttt{a}...\texttt{z}][\texttt{a}...\texttt{z}]^*
\end{array}$$

Top-down parsing of infix expressions may require unbounded buffering of the left operand, as the operator itself arrives later in the input stream. The following shows an input string, and below each symbol is the size of the parse table right after its consumption:

a_j	z	=	f	(	z	)	;	x	=	x	+	y	*	y	*	y	;	g	(	x	)	;	.	#
size	1	0	1	2	3	4	0	1	0	1	0	1	2	3	4	5	0	1	2	3	4	0	0	1

The speculation constant is unbounded. The example demonstrates how the method adapts the table size as input is consumed. Note that `;` and `=` resolves the sum expression currently being parsed, truncating the table, and also that the left operand of the `+` symbol is correctly resolved, while the `*` expression must be buffered.

Algorithm 3 (PARSE).
In: $w = a_0a_1...a_{|w|-1} \in \Sigma^*\#$.
Out: Code $c^0c^1...c^{|w|-1}$, accept/reject.

```
1: $u := \varepsilon$; $T := \bot$; $\vec{K} := S$
2: $R^1, R^2 := (i,j) \mapsto \emptyset$
3: $Forced := \emptyset$
4: FORCE$(S, 0)$
5: for $k \in \{1, ..., |w|\}$ do
6:     $u := u\, a_{k-1}$
7:     $W := \{(i, |u|-1) \mid g_i$ simple$\} \cap Forced$
8:     run FIX
9:     compute$(\vec{K}, 0) \xrightarrow{c} (\vec{K}', m') \downarrow$
       where for all $i, j$, before evaluating $T_{ij}$:
           run FORCE$(i, j)$, then run FIX.
10:    $c^n := c$; $\vec{K} := \vec{K}'$
11:    $T := T[m']$; $R^1 := R^1[m']$; $R^2 := R^2[m']$
12: accept if $\vec{K} = \varepsilon$ else reject
```

Algorithm 4 (FIX).

```
1: while $W \neq \emptyset$ do
2:     let $(p, q) \in W$
3:     $T := F^{(u)}_{pq}(T)$
4:     $W := W \setminus \{(p, q)\} \cup R^2_{pq}$
5:     for $i' \in R^1_p$ do
6:         let $(k, \ell) = D^T_{i'q}$
7:         $R^2_{k\ell} := R^2_{k\ell} \cup \{(i', q)\}$
8:         if $T_{k\ell} \neq \bot$ then
9:             $W := W \cup \{(i', q)\}$
10:        else
11:            run FORCE$(k, \ell)$
```

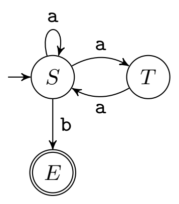
Algorithm 5 (FORCE).
In: $(i, j) \in \mathsf{Index}$

```
1: if $(i, j) \in Forced$ then
2:     return
3: $Forced := Forced \cup \{(i, j)\}$
4: if $A_i \leftarrow A_x[A_y, A_z]$ then
5:     if $T_{xj} = \bot$ then
6:         $R^1_{xj} := R^1_{xj} \cup \{i\}$
7:         run FORCE$(x, j)$
8:     else
9:         let $(k, \ell) = D^T_{ij}$
10:        if $T_{k\ell} = \bot$ then
11:            $R^2_{k\ell} := R^2_{k\ell} \cup \{(i, j)\}$
12:            run FORCE$(k, \ell)$
13:        else
14:            $W := W \cup \{(i, j)\}$
15: else if $j \leq |u| - 1$ then
16:     $W := W \cup \{(i, j)\}$
```

Figure 3. Optimized algorithm which only computes entries needed by the expansion.

Ambiguous Tail-Recursive Programs Any nondeterministic finite automaton (NFA) can be interpreted as a PEG program by assigning a nonterminal to each state, and for each state q with transitions $q \xrightarrow{a_1} q_1, ..., q \xrightarrow{a_n} q_n$ creating a rule $\underline{q} \leftarrow a_1\, \underline{q_1} / ... / a_n\, \underline{q_n}$. The ordering of transitions is significant and defines a disambiguation priority. The final state q^f is assumed to have no transitions, and is given the rule $\underline{q^f} \leftarrow \epsilon$.

If the NFA contains no ε-loops then its language will coincide with that of its PEG encoding, which is a complete program implementing a backtracking depth-first search for an accepting path. The following shows a simple example of an NFA and its prioritized interpretation as a PEG:



$$\underline{S} \leftarrow \mathtt{a}\,\underline{S} \,/\, \mathtt{a}\,\underline{T} \,/\, \mathtt{b}\,\underline{E} \qquad \underline{T} \leftarrow \mathtt{a}\,\underline{S} \qquad \underline{E} \leftarrow \epsilon$$

$$\begin{array}{ll} S \leftarrow P[E, Q] & P \leftarrow A[S, F] \\ T \leftarrow A[S, F] & Q \leftarrow V[E, W] \\ E \leftarrow \epsilon & V \leftarrow A[T, F] \\ F \leftarrow \mathtt{f} & W \leftarrow B[E, F] \\ A \leftarrow \mathtt{a} & B \leftarrow \mathtt{b} \end{array}$$

The NFA is ambiguous, as any string of the form $a^{n+2}b$, $n \geq 0$, can be matched by more than one path from S to E. Nonterminal T is never invoked, as the production $\mathtt{a}S$ *covers* the production $\mathtt{a}T$, meaning that every string accepted by the latter is also accepted by the former, which has higher priority in the choice.

The example triggers worst-case behavior for our method, which fails to detect coverage regardless of the speculation bound, resulting in a table size proportional to the input length. This is obviously suboptimal, as any regular language can be recognized in constant space.

In general, our method is unable to resolve a choice for an unbounded length of input whenever the program contains a production of the form $A_i \leftarrow A_x[A_y, A_z]$ such that (i) there is a string u where $(S, 0) \xrightarrow{c} (A_i\vec{K}, j)$ is in $\mathcal{E}_{T^<(u)}$; and (ii) there is an infinite number of strings $v_1, v_2, ...$ where each v_k is a strict prefix of v_{k+1}, and for all v_k, we have $T^<(uv_k)_{xj} = \bot$ and $(A_z\vec{K}, j)$ **fails**$_d$ is *not* provable for any d. Tail-recursive programs have this property iff either the underlying NFA recognizes an infinite number of strings ambiguously; or it contains a state from which an infinite language can be accepted while a common prefix of the language is accepted via a lower priority transition. Determinizing the NFA and making sure that no accepting state has outgoing transitions yields a fully streaming program, albeit with very different parse trees.

7. Discussion

The workset algorithm is an instance of *chaotic iteration* [5] for computing limits of finite iterations of monotone functions. Our parsing formalism goes back to the TS/TDPL formalism introduced by Birman and Ullman [3] and later generalized to GTDPL by Aho and Ullman [1]. They also present the linear-time tabular parsing technique and show that GTDPL can express recognizers for all deterministic context-free languages, including all deterministic LR-class languages. On the other hand, there are context-free languages that cannot be recognised by GTDPL if matrix multiplication requires super-linear time [14]. Ford's Parsing Expression Grammars (PEG) [8] have the same recognition power as GTDPL, albeit using a larger set of operators which arguably are better for practical use.

Packrat [7] implements the PEG operational semantics with memoization, and can be viewed as "sparse" tabular parsing where only the entries encountered on a depth-first search for an expansion are computed. Our optimized algorithm is a generalization of Packrat which operates in lockstep and explores alternative paths in parallel.

Heuristic approaches include Kuramitsu's Elastic Packrat algorithm [13] and Redziejowski's parser generator *Mouse* [21], both of which are Packrat parsers using memory bounded by a configurable constant. Both approaches risk triggering exponential behavior when backtracking exceeds the bounds of their configured constants, which however seems rare in practice. A disadvantage of heuristic memory reductions is that they have to store the full input string until the full parse is resolved, because they cannot guarantee that the parser will not backtrack.

Packrat With Static Cut Annotations Mizushima, Maeda and Yamaguchi observe that when Packrat has no failure continuations on the stack, all table columns whose indices are less than the index of the current symbol can be removed from memory. To increase the likelihood of this, they extend PEG with cut operators à la Prolog to "cut away" failure continuations, and also devise a technique for sound automatic cut insertion, i.e. without changing the recognized language [18]. Manually inserted cuts yield significant reductions in heap usage and increases in throughput, but automatic cut insertion seems to miss several opportunities for optimization. Redziejowski further develops the theory of cut insertion and identifies sufficient conditions for soundness, but notes that automation is difficult: *"It* *appears that finding cut points in non-LL(1) grammars must to a large extent be done manually"* [22].

The method of Mizushima et al. is subsumed by PTP. An empty stack of failure continuations corresponds to the case where the condition A in a top-level choice $A[B, C]$ is resolved. Insertion of cuts is the same as refactoring the grammar using the GTDPL operator $A[B, C]$, which is the cut operator $A \uparrow B/C$ in disguise. Increasing the speculation bound can achieve constant memory use without requiring any refactoring of the program.

To illustrate, consider the program $\underline{S} \leftarrow (\mathtt{a}/\mathtt{b})^*/(\mathtt{a}^*\mathtt{b}^*\mathtt{c}^*)$. PTP detects that the right alternative cannot succeed as soon as it has seen an input prefix of the form $\mathtt{a}^m\mathtt{b}^n\mathtt{a}$ for some $m, n \geq 1$. In order for the method by Mizushima et al. to obtain the same, cuts have to be inserted as follows: $\underline{S} \leftarrow (!!(\mathtt{aa}^*\mathtt{bb}^*\mathtt{a}))\uparrow(\mathtt{a}/\mathtt{b})^*/(\mathtt{a}^*\mathtt{b}^*\mathtt{c}^*)$. The double negation ($!!e$) functions as a positive lookahead operator which succeeds consuming the empty string only if e matches a prefix of the input. Manual cut insertion is non-trivial and decreases the readability of the grammar. Furthermore, as noted above, it is unlikely that an automatic analysis will be able to reliably infer all cuts needed in practice. Postponing the analysis to parse time, avoids the need for static analysis altogether.

Cost vs Benefit of Memoization Several authors argue that the cost of saving parse results outweighs its benefits in practice [2; 11]. The PEG implementation for the Lua language [11] uses a backtracking parsing machine instead of Packrat to avoid the memory cost [15]. Becket and Somogyi compare the performances of Packrat parsers with and without memoization using a parser for Java as a benchmark [2]. Their results show that full memoization is always much slower than plain recursive descent parsing, which never triggered the exponential worst case, and that performance only increased when just a few selected nonterminals were memoized. Memoization did thus not serve as an optimization, but rather as a safeguard against rare pathological worst-case scenarios. However, another experiment by Redziejowki for the C language shows a significant overhead due to backtracking which could not be eliminated by memoizing a limited number of nonterminals, but required manual rewriting of the grammar [20].

Our technique uses full tabulation rather than memoization, but the above results still suggests that a direct implementation will likely be slower than plain recursive descent parsers on common inputs and carefully constructed grammars. However, ad-hoc parsers cannot be expected to always be constructed in such an optimal way. Furthermore, our best-case memory usage—which is bounded—outperforms recursive descent parsers which must store the complete input string in case of backtracking. This is crucial in the case of huge or infinite input strings which cannot fit in memory, e.g. logging data, streaming protocols or very large data files.

Parsing Using Regular Expressions Medeiros, Mascarenhas and Ierusalimschy embed backtracking regular expression matching in PEG [16]. As we also demonstrate, backtracking regular expression matching can be simulated by tail-recursive PEGs, but may result in worst-case behavior since our method being unable to detect *coverage*. Coverage is undecidable for PEG in general, but is decidable for regular grammars [10].

Grathwohl, Henglein and Rasmussen give a streaming regular expression parsing technique with optimal coverage analysis [10]. With Søholm and Tørholm they develop *Kleenex* which compiles grammars for regular languages into high-performance streaming parsers with backtracking semantics [9]. Since PEGs combine lexical and syntactic analysis, they can be expected to contain many regular fragments. Perhaps the technique of Kleenex can be combined with PTP to obtain better performance.

General CFG parsing As noted, general CFG parsing is unlikely to be doable in worst-case (quasi-)linear time as is the case for PEGs. The classical technique by Cocke, Kasami and Younger [4; 12; 27] is a bottom-up dynamic programming algorithm requiring $O(n^3)$ time and, in practice often prohibitive, $\Theta(n^2)$ randomly accessed memory. Earley's parser [6] has the same time bound, but processes the input reactively in increasing index order, but also requiring $\Theta(n^2)$ randomly accessed data for the prefix read. A fascinatingly simple reactive parser for CFGs constructs derivatives, grammar representations of the remaining language to be accepted after processing a prefix of the input [17]. GLR [26] and GLL parsers [23] extend bottom-up LR-parsing, respectively top-down LL-parsing, to general CFGs by building compact data structures representing multiple stacks for multiple ambiguous parses. Their main advantage is that they support streaming input processing and adapt their run-time to the degree of local nondeterminism present in a grammar. They share with progressive tabling the strategy of eagerly building parse candidates with zero lookahead. As such they are highly reactive to the arrival of input symbols and execute embedded semantic actions prior to reading (much) additional input. Analyzing some lookahead to locally disambiguate and avoid building unnecessary parse candidates may yield substantial run-time benefits, however [19; 9].

PEGs are context-free grammars with a built-in disambiguation strategy that goes beyond mere disambiguation by not even accepting some strings that have a context-free parse. This corresponds to the difference between a combinatory parser over the list functor [25; 24] for CFGs versus one over the Maybe functor for PEGs for representing parse continuations. At a high (and hazy) level, both GLR/GLL and progressive tabling devise compact run-time representations with shared information across alternative continuations and executing multiple continuations breadth-first (reactively) rather than depth-first, while still doing this without eagerly computing more than necessary for a particular input.

8. Conclusion

We have presented PTP, a new streaming execution model for the TDPL family of recursive descent parsers with limited backtracking, together with a linear-time algorithm for computing progressive tables and a dynamic analysis for improving the streaming behavior of the resulting parsers. We have also demonstrated that parsers for both LL and non-LL languages automatically adapt their memory usage based on the amount of lookahead necessary to resolve choices.

A practical performance-oriented implementation will be crucial in order to get a better idea of the applicability of our method. Our prototype evaluation suggests that such an implementation will perform well if based on our optimized algorithm.

Our method fails to be streaming when applied to strictly right-regular grammars. This problem has to be resolved in order to avoid pathological worst cases when developing scannerless parsers, and is future work.

We believe that our method will be useful in scenarios where a streaming parse is desired, either because all of the input is not yet available, or because it is too large to store in memory at once. Furthermore, it can serve as an alternative to Packrat in scenarios where its memory consumption is too high. Possible applications include read-eval-print-loops, implementation of streaming protocols and processing of huge structured data files.

Acknowledgments

This work has been partially supported by The Danish Council for Independent Research under Project 11-106278, "Kleene Meets Church: Regular Expressions and Types"; see `http://diku.dk/kmc`.

The order of authors is insignificant; please list all authors—or none—when citing this paper.

References

[1] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Inc., 1972.

[2] R. Becket and Z. Somogyi. DCGs + Memoing = Packrat Parsing but Is It Worth It? In P. Hudak and D. S. Warren, editors, *Practical Aspects of Declarative Languages*, number 4902 in Lecture Notes in Computer Science, pages 182–196. Springer Berlin Heidelberg, Jan. 2008.

[3] A. Birman and J. D. Ullman. Parsing Algorithms with Backtrack. In *Proceedings of the 11th Annual Symposium on Switching and Automata Theory (SWAT)*, pages 153–174, Washington, DC, USA, 1970. IEEE Computer Society.

[4] J. Cocke and J. Schwartz. Programming languages and their compilers. preliminary notes. second revised version. Courant Institute of Mathematical Sciences, NYU, 1970.

[5] P. Cousot and R. Cousot. Automatic synthesis of optimal invariant assertions: Mathematical foundations. *SIGPLAN Notices*, 12(8):1–12, Aug 1977.

[6] J. Earley. An Efficient Context-free Parsing Algorithm. *Commun. ACM*, 13(2):94–102, Feb. 1970.

[7] B. Ford. Packrat parsing: Simple, Powerful, Lazy, Linear Time. In *ACM SIGPLAN Notices*, volume 37, pages 36–47. ACM, Sept. 2002.

[8] B. Ford. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. *ACM SIGPLAN Notices*, 39(1):111–122, Jan. 2004.

[9] B. B. Grathwohl, F. Henglein, U. T. Rasmussen, K. A. Søholm, and S. P. Tørholm. Kleenex: Compiling Nondeterministic Transducers to Deterministic Streaming Transducers. In *Proc. 43rd Annual Symposium on Principles of Programming Languages (POPL)*, pages 284–297. ACM, 2016.

[10] N. B. B. Grathwohl, F. Henglein, and U. T. Rasmussen. Optimally Streaming Greedy Regular Expression Parsing. In *Proc. 11th International Colloquium on Theoretical Aspects of Computing (ICTAC)*, pages 224–240, September 2014.

[11] R. Ierusalimschy. A Text Pattern-matching Tool Based on Parsing Expression Grammars. *Software Practice and Experience*, 39(3):221–258, Mar. 2009.

[12] T. Kasami. An efficient recognition and syntaxanalysis algorithm for context-free languages. Technical report, DTIC Document, 1965.

[13] K. Kuramitsu. Packrat Parsing with Elastic Sliding Window. *Journal of Information Processing*, 23(4):505–512, 2015.

[14] L. Lee. Fast Context-free Grammar Parsing Requires Fast Boolean Matrix Multiplication. *Journal of the ACM (JACM)*, 49(1):1–15, Jan. 2002.

[15] S. Medeiros and R. Ierusalimschy. A Parsing Machine for PEGs. In *Proceedings of the 2008 Symposium on Dynamic Languages (DLS)*, volume 2, pages 1–12. ACM, 2008.

[16] S. Medeiros, F. Mascarenhas, and R. Ierusalimschy. From regexes to parsing expression grammars. *Science of Computer Programming*, 93, Part A:3–18, Nov. 2014.

[17] M. Might, D. Darais, and D. Spiewak. Parsing with Derivatives: A Functional Pearl. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 189–195, New York, NY, USA, 2011. ACM.

[18] K. Mizushima, A. Maeda, and Y. Yamaguchi. Packrat Parsers Can Handle Practical Grammars in Mostly Constant Space. In *Proc. 9th ACM Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 29–36. ACM, 2010.

[19] T. Parr, S. Harwell, and K. Fisher. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. In *Proc. 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 579–598. ACM, 2014.

[20] R. R. Redziejowski. Some aspects of parsing expression grammar. *Fundamenta Informaticae*, 85(1-4):441–451, 2008.

[21] R. R. Redziejowski. Mouse: From parsing expressions to a practical parser. In *Concurrency Specification and Programming Workshop*, 2009.

[22] R. R. Redziejowski. Cut Points in PEG. *Fundamenta Informaticae*, 143(1-2):141–149, Feb. 2016.

[23] E. Scott and A. Johnstone. GLL Parsing. *Electron. Notes Theor. Comput. Sci.*, 253(7):177–189, Sept. 2010.

[24] S. D. Swierstra. Combinator Parsing: A Short Tutorial. In A. Bove, L. S. Barbosa, A. Pardo, and J. S. Pinto, editors, *Language Engineering and Rigorous Software Development*, number 5520 in Lecture Notes in Computer Science, pages 252–300. Springer Berlin Heidelberg, 2009. DOI: 10.1007/978-3-642-03153-3_6.

[25] S. D. Swierstra, P. R. A. Alcocer, and J. Saraiva. Designing and Implementing Combinator Languages. In S. D. Swierstra, J. N. Oliveira, and P. R. Henriques, editors, *Advanced Functional Programming*, number 1608 in Lecture Notes in Computer Science, pages 150–206. Springer Berlin Heidelberg, Sept. 1998. DOI: 10.1007/10704973_4.

[26] M. Tomita. An Efficient Augmented-context-free Parsing Algorithm. *Comput. Linguist.*, 13(1-2):31–46, Jan. 1987.

[27] D. H. Younger. Recognition and parsing of context-free languages in time n3. *Information and Control*, 10(2):189–208, Feb. 1967.