Taming Context-Sensitive Languages with Principled Stateful Parsing

Nicolas Laurent   Kim Mens
Université catholique de Louvain, ICTEAM
{nicolas.laurent, kim.mens}@uclouvain.be

Abstract

Historically, true context-sensitive parsing has seldom been applied to programming languages, due to its inherent complexity. However, many mainstream programming and markup languages (C, Haskell, Python, XML, and more) possess context-sensitive features. These features are traditionally handled with ad-hoc code (e.g., custom lexers), outside of the scope of parsing theory.

Current grammar formalisms struggle to express context-sensitive features. Most solutions lack *context transparency*: they make grammars hard to write, maintain and compose by hardwiring context through the entire grammar. Instead, we approach context-sensitive parsing through the idea that parsers may *recall* previously matched input (or data derived therefrom) in order to make parsing decisions. We make use of mutable *parse state* to enable this form of recall.

We introduce *principled stateful parsing* as a new transactional discipline that makes state changes transparent to parsing mechanisms such as backtracking and memoization. To enforce this discipline, users specify parsers using formally specified primitive state manipulation operations.

Our solution is available as a parsing library named *Autumn*. We illustrate our solution by implementing some practical context-sensitive grammar features such as significant whitespace handling and namespace classification.

Categories and Subject Descriptors   D.3.4 [*Programming Languages*]: Processors—Parsing

Keywords   stateful parsing, grammars, context sensitivity, data dependence, parsing expressions

Nicolas Laurent is a research fellow of the Belgian fund for scientific research (F.R.S.-FNRS).

1. Introduction

In this section, we review the notion of context-sensitive parsing (1.1), describe how we tackle it through recall and stateful parsing (1.2), and situate our implementation (1.3).

In what follows, we use the term *parser* to refer to any unit of functionality that can match some input text (both at the the lexical and grammatical levels) and produce a result based on this match. We assume that simple parsers can be combined into increasingly complex parsers.

1.1 Context-Sensitive Parsing

True context-sensitive parsing is seldom applied to programming languages. Writing context-sensitive grammars can prove challenging, for instance the grammar for the language $a^n b^n c^n$ is notoriously tricky [10]. In addition, most language features can be adequately expressed using the much more tractable context-free grammars. Nevertheless, many mainstream languages exhibit context-sensitive features. Here are a few examples:

- In C, in order to determine whether the statement `x*y;` is the product of x by y, or rather the declaration of a variable y which is a pointer to type x, one must analyze the type definitions preceding the statement.
- In Haskell and Standard ML, programmers can introduce operators with custom precedence and associativity. The parser needs to interpret these definitions in order to be able to parse the remainder of the input.
- Since Python has significant indentation, a Python parser needs to detect when the indentation level increases or decreases.
- In XML, opening tags must be matched with corresponding closing tags. For instance, `<foo></foo>` is valid while `<foo></bar>` is not. As such, an XML parser must memorize the names of open tags, at arbitrary levels of nesting.
- Many network protocols, including TCP, make use of length-delimited fields whose length is not known in advance but indicated by a length field that precedes them.

Most parsing tools cannot adequately handle these syntactic peculiarities, leading to all sorts of hacks, and sometimes to the rejection of parsing tools altogether. There are a few exceptions however, which we review in Section 2. Section 3 deals with the key properties almost all these solutions lack, namely *context transparency*, hence calling for a new solution.

1.2 Recall and Stateful Parsing

While context sensitivity was first characterized by Chomsky [5], his Context-Sensitive Grammars (CSG) are only of little help, due to the intricate coding that they require[1]. A CSG is made of rewrite rules $X \rightarrow Y$ where X and Y are strings of mixed terminals and non-terminals. These rules must be non-contracting: Y, as a string of symbols, must not be shorter than X.[2] As a matter of fact, these grammars were never meant to describe programming languages, but natural languages, where the shape of the rules make much more sense. In particular, it is difficult to encode *recall* constraints: for instance, requiring the same string of tokens to appear at two different locations in the sentence (assuming the string is not fixed in advance).

We propose instead to approach context sensitivity through the notion of *recall*, i.e. the ability to accept sentences based on relationships between some of their parts. This is more easily understood in parsing terms as the capability to make parsing decisions based on previously matched input.

We enable recall by allowing users to write parsers which can manipulate mutable state. However, unlike solutions deployed by existing parsing tools such as ANTLR [24] and Rats! [9], we are *principled* about state use. Indeed, parsing algorithms do not proceed linearly. When faced with a choice, they may speculatively try an alternative, and need to backtrack if this alternative does not succeed. When backtracking happens, all changes made to the state during the speculative execution need to be reversed. Parsers may also memoize the result of a speculative execution. In a stateful model, these results need to include the state changes incurred by the execution. As will be explained in Section 4, we satisfy these requirements by introducing primitive operations to manipulate mutable state in a principled way.

1.3 Implementation

We implement our stateful parsing approach as a PEG-like [8] (top-down recursive descent) functional-style [3] parser-combinator library named *Autumn*, which can be used with Java, Kotlin, and other Java-compatible languages. Implementation details are given in Section 5 and a detailed use case is worked out in Section 6.

[1] The same holds for a large body of work on *mildly context-sensitive grammars*. [16]

[2] In reality, CSG rules are not required to be non-contracting, but non-contracting grammars and CSG describe the same set of languages. [6]

[3] Parsers are merely functions. Moreover, custom parsers can be defined.

2. Related Work

As the problem of context-sensitive features in programming languages is not new, it is not surprising that several solutions have been proposed. We review these solutions in order to better put our contributions in perspective. We do not purport to review the entire body of work on context-sensitive parsing, but only the approaches closest to our goal. In particular, we left out the literature on context-sensitive lexical analysis (e.g., [3, 30]) which by definition only handles a small subset of all context sensitivity issues.

2.1 Backtracking Semantic Actions

Parsing with backtracking semantic actions [29] is an approach that extends a (general) backtracking LR parser with reversible semantic actions. Upon backtracking, state changes are reversed. Two important restrictions apply: state changes can only occur during term reduction, and the state can only affect the parse through semantic conditions that trigger backtracking.

Compared to our approach, the difference in implementation (top-down recursive versus LR) has far-reaching consequences. Backtracking semantic actions reverse state changes rather than making snapshots of the state. Accordingly it becomes impossible to compare state snapshots. These capabilities are very useful in the context of PEG parsing, as they enable the definition of custom parsers, left-recursion and memoization; but less so within an LR system where custom parsers cannot be defined.

We also believe our *top-down recursive-descent* model to be more intuitive in the presence of state. Logically, state changes occur as parser are invoked, from left to right and from top to bottom. Backtracking semantic actions, on the other hand, are executed upon term reduction. This means that a parser may modify the state before other parsers that matched input on its left.

Despite these caveats, we consider parsing with backtracking semantic actions [29] to be the safest and most convenient system for context-sensitive parsing among those presented in this section.

2.2 Data-dependent grammars

Jim et al. [15] proposed data-dependent grammars, a formalism which permits context sensitivity by allowing rules to be parameterized by semantic values. A parameterized non-terminal appearing on the right-hand side of a rule acts as a form of function call that also returns a semantic value. These semantic values are computed by *semantic actions* written in a general-purpose programming language. There are also *semantic predicates* which can make grammar branches succeed or fail depending on a semantic value.

Data-dependent grammars can be compiled to a format accepted by a target parsing tool, which must support fairly general semantic actions. In subsequent work [14], the authors introduced a new kind of automaton that can be used to implement parsers recognizing data-dependent grammars. These techniques are put to work in a tool called Yakker.

Data-dependent grammars, though theoretically compelling, suffer from usability issues. The value-passing model means that the parse state needs to be threaded throughout the grammar. Making a rule dependent on a new semantic value means that all rules through which this rule is reachable might need to be modified to pass this value around. Maintainability-wise, this is far from ideal. Moreover, it harms composability, as a rule must be aware of all states it has to pass through.

Afroozeh and Izmaylova [1] show how advanced parser features such as lexical disambiguation filters, operator precedence, significant indentation and conditional preprocessor directives can be translated to data-dependent grammars. Quite clearly, the task is non-trivial and one comes away with the feeling that dependent grammars are better suited as an elegant calculus to be targeted by parsing tool writers rather than as a paradigm that fits the needs of tool users. The machinery implementing the formalism is also distinctively non-trivial, involving a multi-stage transformation into a continuation routine or into a new kind of automaton. In contrast, our approach consists of a lightweight library that can be layered on top of a general-purpose programming language.

Finally, we note that the much older Definite Clause Grammars (DCGs) [25] formalism works on almost exactly the same principle, but building upon logic programming. Accordingly, it suffers from similar limitations.

2.3 Monadic parsers

Monadic parsing [12] is a well-known way to build functional-style parser-combinator libraries, made popular by Haskell libraries such as Parsec [21]. In this paradigm, the type of a parser is a function parameterized by a result type, i.e. with signature $string \rightarrow (string, result)$, where the parameter string is the input text and the returned string is the input remaining after parsing. The parser type is also a monad instance, meaning there is a `bind` function whose signature, in Haskell notation, is:

```
Parser r1 -> (r1 -> Parser r2) -> Parser r2
```

where `r1` and `r2` are result types. This function takes a parser as first parameter, and a function which transforms the result of the parse into another parser as second parameter. When invoked, the parser returned by `bind` will invoke the first parser, pass its result (of type `r1`) to the function, then invoke the parser this function returns, yielding a result of type `r2`.

The important point about monadic parsers is that they can handle context sensitivity. Indeed, the second parameter to `bind` (the function) returns a parser from a result. This means that the behaviour of the parser returned by `bind` depends on data acquired during the parse: this is a form of *recall*.

An in-depth analysis of this aspect was done by Atkey [2]. In particular, he formalizes monadic parsers by introducing *active right-hand sides*, which are the right-hand sides of rules that can contain monadic combinators. These combinators generate grammar fragments at parse-time (much like a monadic parser generates a new parser), hence the term *active*. While monadic parsing seems at first sight very similar to the data-dependent grammars from Section 2.2, Atkey [2] carefully contrasts the two approaches:

> *We characterise their [Jim et al.] approach as refining context-free grammars: each Yakker grammar has an underlying context-free grammar with regular right-hand sides, and the constraints allow for sophisticated data-dependent filtering of parses. In contrast, we consider active right-hand sides that generate the grammar as the input is read.*

Nevertheless, monadic parsers suffer from the same pitfalls as data-dependent grammars: the state is threaded through the grammar (or code), leading to poor maintainability and composability.

2.4 Attribute Grammars

Attribute grammars [18] associate attributes to AST nodes (assuming an AST node per matched grammar rule). The attributes can be synthesized: their value derived from the attributes of subnodes, or inherited: their value computed by a parent node. The formalism supports context-sensitive parsing through production guards predicated over attributes.

However, attribute grammars are not context-transparent. To enable recall, they need to propagate the recalled value from the definition site to the use site, through a chain of of synthesized and inherited attributes. Even reference attributed grammars [11], which allow attributes to contain references to nodes, do not fully solve this distribution problem.

2.5 Stateful Parsing

Manipulating parse-wide state can be an effective solution to the problem of data dependence: the data depended upon can be written in the state when encountered and read or even altered later on.

Broadly speaking, we can distinguish two big classes of stateful parsing tools. First, there are parser combinator libraries that allow users to write their own subparsers. Notable examples include Parboiled [7], Lua Peg [13] and Scala's parser combinators [22]. Since these custom parsers are implemented in a general-purpose programming language, they can manipulate state, even though the libraries make no provision for this. Second, there are parsing tools that provide very general semantic actions and semantic predicates. Notable examples include Bison [28] and ANTLR [24]. These work much like their counterpart in Yakker (cf. Section 2.2), except that instead of returning a value, semantic actions may modify a global state object.

Unfortunately, most parsing tools in both categories do not make the necessary provisions for dealing with backtracking and memoization: if the parser backtracks over a construct that made state changes (semantic action or custom parser), these changes need to be undone; if the parser can memoize the result of a construct, state changes need to be memoized as well. In the absence of such guarantees, a construct can only access state which it is sure has not been corrupted by changes that should have been discarded. It must also be sure that some state-altering construct was not skipped due to memoization. These are tricky propositions to verify even for medium-sized grammars, and every change to the grammar threatens to falsify them.

One may think that solving the backtracking problem is simply a matter of inserting a construct that reverses state changes whenever a rule fails. However, a rule can be backtracked over even if it succeeded. It suffices that one of the rules through which our rule was reached fails. Hence this scheme would entail, for each state-altering construct, the modification of every rule through which it can be reached.

2.6 Rats!

Rats! [9] is a fully-memoizing (*packrat*) PEG parser. Rats! is, to the best of our knowledge, the only stateful parsing tool that provides some guarantees for state usage, by ensuring that state changes are discarded if certain conditions are met.

For this purpose, Rats! introduces *transactions* that wrap rules under which state changes might occur. A transaction can either succeed, in which case its state changes are retained, or fail, in which case the changes are discarded. Rats! also requires that a nonterminal invoked at a given position within a transaction must always modify the state in the same way, no matter how that nonterminal was reached. Combining transactions with this requirement ensures that Rats! will never have to discard the memoization of a rule, hence upholding the linear-time guarantee of packrat parsers.

In spite of its advantages, this scheme has two important pitfalls. First, it requires nonterminal invocations at a given position to always return the same result. This precludes parsing expressions that modify the behaviour of the parsing expression they invoke. However, this capability is valuable in practice. For example, we use it to enable left-recursion handling in our library (cf. Section 5.4).

Second, state changes are not memoized. If a rule succeeds after applying a state change, but the enclosing transaction fails, the changes are lost. If we wanted to call the rule at the same position again, the memoized result would be used and it does not include the state changes. This means that a state change cannot safely be referenced by two different transactions, and that transactions cannot be re-tried after a state change higher up in the grammar hierarchy.

3. Context Transparency

As the previous section has shown, enabling the definition of context-sensitive languages without jeopardizing maintainability, composability or even safety is no easy feat. We put forward the notion of *context transparency* as the gold standard that a context sensitive parsing mechanism needs to meet in order to be considered sufficiently practical.

A grammatical construct is **context-transparent** if it is unaware of the context shared between its ancestors and its descendants.

Data-dependent grammars, monadic parsers, DCGs and attribute grammars are not context-transparent because of the need to explicitly pass values around. For instance, consider two data-dependent grammars[4]: a grammar for a Python-like language with significant indentation, in which the rules for block-level constructs (statements, definitions) are paremeterized by the indentation level; and a grammar for a generic macro definition language (e.g., GNU M4). We want to compose these two languages such that macro definitions may appear anywhere where definitions can appear in our Python-like language. Additionally, we want macro bodies to include Python-like code.

The issue is that the rules in the macro language grammar know nothing about indentation level, yet the indentation level needs to be shared between the block holding the macro definition and the Python-like code appearing inside macro definitions. In this case, the lack of context transparency would force us to rewrite all rules in the macro language grammar to carry around the indentation level.

Stateful parsers also are not context-transparent, as they must ensure that no unforeseen backtracking or memoization takes place. For instance, if a parser a manipulates the state and its callers do not expect it to backtrack, it cannot be swapped for a parser $c(a)$ (where c is some parser combinator) without first ensuring that $c(a)$ never backtracks over a.

Lack of context-transparency makes grammars hard to reason about, hence hard to write and to maintain: refactoring, extending or composing grammars becomes particularly challenging, because each change to a rule might entail the need to modify all rules through which it is (transitively) reachable. In stateful parsers, such changes are liable to introduce undesired backtracking or memoization.

We suggest a simple solution: use stateful parsing (which does not thread context through the grammar), but undo state changes upon backtracking and allow the memoization of state changes. And to achieve this, we introduce a new context sensitivity handling discipline: *principled* stateful parsing.

4. Principled Stateful Parsing

In Section 1, we established the relevance of context-sensitive parsing and introduced the notion of *recall* as a way to express context-sensitive features in terms of backreferences to previously matched input. We enable recall by storing the matched input (or data derived thereof) in a mutable data store: the *parse state*. This section expounds how *principled stateful parsing* is able to work with parse state while avoiding the usual pitfalls of stateful parsing (cf. sections 2.5 and 3).

[4] The same reasoning applies to monadic parsers, DCGs and attribute grammars.

4.1 Intuition

Before diving into a formal explanation, we present the remarkably simple intuition behind the approach.

The point of using state is to pass context around implicitly, without the need to hardwire context in the grammar, hence achieving context transparency (cf. Section 3).

If the execution of a parser were linear, simply reading/writing to this state would suffice. Unfortunately, parsers must sometimes perform speculative executions that may fail further down the line, a phenomenon called backtracking. When backtracking occurs, all state changes in the speculative execution being backtracked over must be reversed. Hence, we need an operation that can take a **snapshot** of the state at a given point, and an operation that can **restore** the state described by such a snapshot.

Given these requirements, it helps to think of the parse state as a log of the operations applied to the state, which can be snapshot and rolled back as required. Appropriately, this is also how we formalize the parse state.

Additionally, it is sometimes desirable to save the result of a speculative execution (whether it failed or not), i.e., the state changes it induced: a *delta* acquired by performing a **diff** between the states before and after the execution. It is also necessary to be able to **merge** these changes back into the state. The most straightforward application of the *diff* and *merge* capabilities is the memoization of parse results. However, other valuable use cases exist, such as longest-match parsing and left-recursive parsing (see Section 5.4).

This motivates the need for four primitive state-manipulation operations: **snapshot**, **restore**, **diff** and **merge**. These operations are described in section 4.2.3.

Principled stateful parsing is an approach where parsers behave transactionally: each parser invocation either succeeds or leaves the state untouched. Additionally, it is possible to generate and merge deltas corresponding to state changes made by parser invocations. All this is made possible through the use of formally specified state manipulation operations.

4.2 Formalization

We formalize our approach using the Z notation [26], though eschewing its schema calculus in favor of a purely functional presentation.[5] The Z notation is a formal specification language that builds on top of Zermelo-Frankel set theory, first-order logic and simply typed lambda calculus. As such, Z can be seen as a language where functions can be defined in lambda calculus extended with predicates from first-order logic and set theory. Formal assertions over the functions can be made using the same notation. We also note that in Z, all types used in the lambda calculus are sets.

Since we adopt the functional parser-combinator approach (cf. Section 1), parsers are simply functions manipulating parse state (Section 4.2.1) whose set-theoretic signature is given in Section 4.2.2. Section 4.2.3 formally specifies the primitive state-manipulation operations that were briefly introduced in Section 4.1. Finally, Section 4.2.4 gives the semantics of parser invocation by specifying the *call* operation, which maps a parser (as defined in 4.2.2) to a single state transformation.

4.2.1 Parse State

At the core of our approach lies the notion of parse state. The parse state abstracts over a general mutable data store. We do not place any constraint on the data within the store. This is formalized as follows.

$$
\begin{array}{|l}
[\mathit{CHANGE}] \\
\mathit{STATE} = \operatorname{seq} \mathit{CHANGE}
\end{array}
$$

The square brackets introduce the abstract set *CHANGE* of all state changes. What exactly constitutes a state change (most likely the mutation of a memory location) is an implementation concern that is not relevant to the formalization.

STATE is the set of possible parse states: i.e., of possible configurations of our mutable store. We represent a parse state as a sequence of state changes. This means that a state can be seen as a log of the operations over the mutable store it represents, assuming some well-defined initial state.

In Z, the set of sequences of items from the set S is written *seq S* and corresponds to the powerset of pairs $(i, s) \in \mathbb{N} \times S$, or equivalently to the powerset of partial functions $\mathbb{N} \rightarrowtail\!\!\!\!\!\!\;\mapsto S$. In each sequence, the indices are unique and consecutive.

In practice, an implementation of the approach will want to use parse state to reify important parsing notions, such as input position. We consciously avoided making our formalism needlessly specific, hence the absence of some usual parsing notions such as input position. This enables using our approach to parse non-linear inputs (e.g., object graphs), or perform computations that only bear nominal resemblance to traditional parsing, even though this direction is outside the scope of the current paper.

4.2.2 Parsers

A parser represents a computation over the parse state that either succeeds or fails, and has side effects on the parse state, in the form of *state changes*, as introduced in the previous section.

$$
\begin{array}{|l}
\mathit{TRANSFORM} = \mathit{STATE} \rightarrow \mathit{STATE} \\
\mathit{PARSER} = \mathit{STATE} \rightarrow \operatorname{seq} \mathit{TRANSFORM} \\
\mathit{RESULT} ::= \mathit{success} \mid \mathit{failure} \\
\mathit{result} : \mathit{STATE} \rightarrow \mathit{PARSER} \rightarrow \mathit{RESULT}
\end{array}
$$

Formally, a parser is a function from a state — the current state at the time of invocation — to a sequence of

[5] To improve the presentation, we took some liberty with the Z layout (but not with the notation). A machine-understandable version of the specification is available online [20].

transformations, which move from one state to another. This amounts to defining a parser in terms of its execution trace.

Two things seem to be missing from this definition. First, it does not say if the parse succeeds when run over a specific state. This property is exposed separately through the *result* predicate rather than as part of the *PARSER* signature. This approach is not significant: it simply makes the math look nicer. Second, the input being parsed does not explicitly appear in the signature. Instead, the input is assumed to be held within the parse state.[6]

A parser is a recognizer of states. It accepts states for which $result\ state = success$ holds. If within the input state one dissociates the *parse input* from the rest of the state (the *context*), one can see that the parser recognizes — hence also defines — different languages depending on the context.

But a parser is also a transformer of states as well: when invoked it performs a $STATE \rightarrow STATE$ transformation. In section 4.2.4 we explain how to derive this transformation from a parser (recall that parsers have type *PARSER* defined as $STATE \rightarrow seq\ TRANSFORM$), as a means of defining the semantics of a parser given its execution trace. We could alternatively have defined *PARSER* as $STATE \rightarrow TRANSFORM$ (with the result being the composition of the transformations in the sequence), or directly as $STATE \rightarrow STATE$. We chose to emphasize the execution trace — a sequence of transformations — instead, because the primitive state operations described in the next section are suppliers of such transformations, to be composed to yield the transformation performed by the parser.

This representation also emphasizes that the parse state is both an input of the parser and an input of the returned transformations. This reflects the fact that a parser is context-sensitive: it chooses which operation to perform depending on the state. This is closely related to the notions of active right-hand sides [2] and monadic parsing [12]. In fact, each operation in the sequence is chosen depending on the state obtained by running the initial state through the composition of all preceding transformations. Abstracting over this makes the specification much simpler, without altering its meaning.

4.2.3 Primitive Operations

We now present six primitive operations (amongst which the four announced in Section 4.1) that parsers can perform.

$$\begin{array}{|l}
SNAPSHOT = \mathrm{seq}\,CHANGE \\
DELTA = \mathrm{seq}\,CHANGE \\
\\
call : PARSER \rightarrow TRANSFORM \\
snapshot : STATE \rightarrow STATE \\
diff : SNAPSHOT \rightarrow STATE \rightarrow DELTA \\
applyChange : CHANGE \rightarrow TRANSFORM \\
restore : SNAPSHOT \rightarrow TRANSFORM \\
merge : DELTA \rightarrow TRANSFORM
\end{array}$$

[6] Nothing precludes the input from being mutable, even though we have not investigated the usefulness of the idea.

Call Of these six, *call* has a special status: it represents the invocation of a parser. We will define this operation in section 4.2.4, hence specifying the semantics of parsers given their execution trace. Note that the signature definition of *call* expands to $PARSER \rightarrow STATE \rightarrow STATE$: a parser must be called with a state as parameter.

Snapshot A snapshot, as the name implies, is a capture of the state at a specific point during the execution. Naturally, this makes *SNAPSHOT*, the set of all snapshots, equivalent to *STATE*. Formally, the *snapshot* operation, which creates such a capture, is simply the identity function.

$$\begin{array}{|l} snapshot = \lambda\, x : STATE \bullet x \end{array}$$

Diff The *diff* operation returns a *DELTA* object representing the difference between a snapshot and the current state, as a set of state changes. As a precondition, this operation requires the snapshot it receives to be a prefix of the current state. This is expressed with the Z built-in *prefix* infix operator. By keeping the deltas append-only, we ensure that a delta can be later *merged* to any state, not just the one corresponding to the snapshot.

$$\begin{array}{|l}
\forall\, sn : \mathrm{dom}\, diff \bullet \forall\, st : \mathrm{dom}\,(diff\ sn) \bullet \\
\quad sn\ prefix\ st
\end{array}$$

Since deltas are state suffixes, *DELTA*, the state of all deltas, is equivalent to *STATE*.

Assuming the precondition is respected, *diff* can be defined as the remainder of the current state after chopping off the prefix corresponding to the snapshot. In Z, the *squash* function packs the indices (left-hand side) of a set of pairs in $\mathbb{N} \times S$, where S is some set, in order to turn this set into a proper sequence. For instance, it turns $\{(2,x),(5,y)\}$ to $\{(1,x),(2,y)\}$.

$$\begin{array}{|l}
diff = \lambda\, sn : SNAPSHOT \bullet \lambda\, st : STATE \bullet \\
\quad squash\,(st \setminus sn)
\end{array}$$

Transformations All operations except *diff* and *snapshot* return a transformation. Recall that we defined *PARSER* as $STATE \rightarrow seq\,TRANSFORM$. The transformations returned by the operations are precisely those which will be part of a parser's execution trace. *diff* and *snapshot* are different because they do not modify the parse state. Instead, *diff* and *snapshot* create new objects, which can be freely passed through the parse state.

ApplyChange The *applyChange* operation is very simple: given a change, it simply returns a transformation that applies this change, by appending it to the change log. It can be defined as follows, using the concatenation operator ($\frown$) to append the change to the old log.

$$\begin{array}{|l}
applyChange = \lambda\, c : CHANGE \bullet \lambda\, st : STATE \bullet \\
\quad st \frown \langle c \rangle
\end{array}$$

This "operation" models the fact that parsers can perform arbitrary state changes.

Restore The *restore* operation takes a snapshot as input and returns a transformation that brings the state to that described by the snapshot.

$$restore = \lambda\, sn : SNAPSHOT \bullet \lambda\, st : STATE \bullet sn$$

Merge The *merge* operation takes a delta as input and returns a transformation that appends this delta to the input state.

$$merge = \lambda\, d : DELTA \bullet \lambda\, st : STATE \bullet st \frown d$$

4.2.4 Parser Invocation Semantics

We now look at how the transformation returned by the *call* operation can be derived from the execution trace returned by a parser. Recall that the *call* operation's signature is $PARSER \rightarrow TRANSFORM$.

We start by defining two helper functions. *composeTwo* maps sequences of transformations of length $n \geq 2$ to a sequence of length $n - 1$ similar to the input sequence, but where the first two items have been replaced by their composition ($s\,1$ and $s\,2$ access the first two items of s while $\fatsemi$ is the relational composition operator). *reduceN* takes a natural n and a sequence of transformations and returns the composition of its n first items, or the identity transformation if $n = 0$. This is achieved by iteratively running the sequence through *composeTwo*, using the Z built-in *iter* operator.

```
$composeTwo = \lambda\, s : \mathrm{seq}\, TRANSFORM \bullet$
    $\langle s\,1 \fatsemi s\,2 \rangle \frown tail\,(tail\, s)$
$reduceN = \lambda\, n : \mathbb{N} \bullet \lambda\, s : \mathrm{seq}\, TRANSFORM \bullet$
    $\mathbf{if}\,(n = 0)\ \mathbf{then}\ \mathrm{id}\, STATE$
    $\mathbf{else}\, iter\,(n - 1)\, composeTwo\, s\, 1$
```

With this in place, we define the result of *call* as the composition of all transformations within the call's execution trace, assuming the parser invocation is successful. Otherwise, the identity transformation is returned. The hash sign (#) is an operator returning the cardinality of a set.

```
$call = \lambda\, p : PARSER \bullet \lambda\, st : STATE \bullet$
    $\mathbf{if}\,(result\, st\, p = success)$
        $\mathbf{then}\, reduceN\,(\#\, p\, st)\,(p\, st)\, st$
    $\mathbf{else}\, st$
```

5. Implementation

We implemented the *principled stateful parsing* approach in a general-purpose parsing library called *Autumn*. It is freely available online [20]. Autumn is implemented in Kotlin, an up-and-coming JVM language that closely matches Java's semantics while reducing boilerplate. Kotlin possesses many features that make it particularly well suited for writing domain-specific languages (DSLs), an ability we exploit to define grammars. We will introduce these features as we encounter them. Our approach is not language-specific and can easily be ported to other languages.

We start by exposing the fundamentals of the Autumn API and how it relates to our formalization (Section 5.1). We then show the API in action on a simple example (Section 5.3). Finally we discuss how the API enables simple left-recursion handling (Section 5.4).

5.1 The Autumn API

In this section, we review how our implementation relates to our formalization of principled stateful parsing (Section 4). Figure 1 shows the key interfaces and classes in our implementation.

```
interface Parser {
    fun parse (ctx: Context): Result
}

sealed class Result {
  object Success: Result()
  open class Failure (val pos: Int, val msg: String)
    : Result()
}

class Context (input: String,
               vararg states: State<*, *>) {
  var pos: Int = 0
  val text: String = input + '\u0000'
  fun <T: State<*,*>> state(klass: Class<T>): T { ... }

  fun snapshot(): Snapshot { ... }
  fun restore(snap: Snapshot) { ... }
  fun diff(snap: Snapshot) { ... }
  fun merge(delta: Delta) { ... }

  ...
}

class Snapshot { ... }
class Delta { ... }

interface State<Snapshot, Delta> {
  fun snapshot(): Snapshot
  fun restore(snap: Snapshot)
  fun diff(snap: Snapshot): Delta
  fun merge(delta: Delta)
}

abstract class Grammar {
  open val whitespace: Parser
    = ZeroMore(CharPred(Char::isWhitespace))
  open val root: Parser
  open val requiredStates: List<State<*,*> = emptyList()
  ...
}
```

Figure 1. Key interfaces and classes in the implementation of Autumn.

Parser We represent a parser by an instance of the `Parser` interface. Implementers must override the `parse` method, which takes a `Context` as parameter and returns a `Result`: either a `Success` or a `Failure` which holds the position at which the failure occurred, together with a diagnostic message. It can also hold custom diagnostic information through subclassing of `Failure`. This gives a lot of control over the error messages that will be shown to the user.

Context Each parse — the invocation of a parser on a complete piece of input text — has an associated `Context` object. The role of this object is to hold the state for the parse. The context is passed down to parsers during parser invocation, so that all parsers may access it. Using a context object rather than global state allows multiple parses to co-exist, potentially in parallel.

The mutable store mentioned in the formalization is represented as a collection of singleton classes implementing the `State` interface. Parsers can retrieve a state instance by calling the `state` method with the proper class object. Note that the state held by the context also includes the input text and the input position, although, as a special provision, these can be accessed directly through the `text` and `pos` properties respectively.

State The `State` interface has four methods: `snapshot`, `restore`, `diff` and `merge`, corresponding to the four key operations introduced in Section 4.1, but only locally for the `State` instance itself. To get the parse-wide semantics of Section 4, we aggregate the state operations over all `State` instances. This is achieved through the methods in the `Context` class that mimic the `State` interface. These methods manipulate the `Snapshot` and `Delta` classes (not to be confused with the eponymous type parameters of interface `State`), which aggregate `State`-level snapshots and deltas.

In our formalization, we represented the mutable store as a log of all operations over the store. In practice, this might not be a good idea, as parsers must compute using the state, meaning the "current state" would need to be re-derived from the whole log at least every time backtracking happens, unless all operations were fully reversible.

The approach we took instead was to give maximum implementation flexibility to the programmer: he can choose, for each `State` instance, the most appropriate strategy to create snapshots and deltas, as well as to restore/merge them back in. Maintaining a log of reversible operations is only one possibility among many.

However, having the programmer implement the `State` interface for each data structure he wishes to manipulate would be tedious and repetitive. Hence, we supply base implementations for common use cases, such as:

- `CopyState`: for states that are records containing just a few fields, which we can afford to copy every time, and which can be treated as a unit (i.e., a delta cannot represent that a field changed while the others retained their previous value — the value of every field is systematically captured).
- `StackState`: represents a stack as a singly linked list (which is naturally immutable). Snapshots and deltas are represented as nodes in the list. The list is treated as a unit.
- `MonotonicStack`: Similar to `StackState`, but adds the restriction that `diff` must only be called with a snapshot that is a suffix of the current stack. Deltas are then prefixes of the stack and can be grafted back at a later time, allowing for granular change handling.
- `MapState`: represents a map as an immutable Hash Array Map Trie (HAMT) [4]. Our implementation is based on that of Steindorfer and Vinju [27] which ensures good performance. The map is treated as a unit.
- `InertState`: represents state that does not change during the parse, or whose change is not significant (e.g., logging logic). All operations are implemented as no-ops.

Grammar Programmers must subclass the `Grammar` class in order to define a new grammar. When doing so, they must define the root parser by overriding `root()`, and provide any required `State` instances by overriding `requiredStates()`. `Grammar` also defines a default `whitespace` parser which consumes any number of characters matching the Java `Char::isWhitespace` predicate.

Parser combinator libraries traditionally struggle with the definition of recursive parsers: it is forbidden to write [`val A = Seq(..., A)`] because `A` is not defined yet when it is evaluated on the right-hand side. Using the `Grammar` initialization logic, we can replace the recursive reference to `A` with the [`!"A"`] operator-overloading syntax: [`val A = Seq(..., !"A")`]. This creates a stub parser which will be patched with a reference to `A` at parse-time. To achieve this, the `Grammar` class maps names to parsers through reflection over its `Parser`-valued fields.

5.2 Contract

Autumn enforces the *principled stateful parsing* guarantees, but only if its interfaces are implemented correctly. In particular:

- Each implementation of `Parser` must either succeed or undo all the state changes it incurred. This is usually achieved through the use of `snapshot()` and `restore()`.
- Each implementation of `State` must implement its operations according to the specification given in Section 4.

5.3 Example

As an illustration, Figure 2 shows the implementation of one of the most fundamental parser combinators, the sequential composition of parsers. The resulting parser calls its sub-parsers sequentially, succeeding if they all succeed. If one of them fails, the parser reverts the state to its initial condition.

5.4 Left-Recursion

When a parser invokes itself (either directly or indirectly through intermediate parsers) without intervening state changes, the result is an infinite loop of parser invocations. This is a well-known problem of top-down recursive parsers, called *left-recursion*. Fortunately, it can be mitigated as follows: start by running the left-recursive parser while failing all left-recursive invocations, then re-run it, using the result of the initial parse as the result of all left-recursive invocations.

```
class Seq (vararg children: Parser): Parser(*children) {
  override fun _parse_(ctx: Context): Result {
    val snapshot = ctx.snapshot()
    for (child in children) {
      val r = child.parse(ctx)
      if (r is Failure) {
        ctx.restore(snapshot)
        return ctx.failure
    }   }
    return Success
}   }
```

Figure 2. Implementation of the sequential parser combinator in Autumn.

Repeat until as much input as possible has been matched. Refer to our earlier paper [19] for more details.

Interestingly, this strategy can be implemented entirely within the stateful parsing paradigm. In particular, when we speak of *result of a parse*, we are really referring to a *delta* of the parse state. These deltas need to be stored in the parse state, so that they can be retrieved by left-recursive invocations. We also need to track, within the state, which left-recursive parsers have been invoked at which input position, so that we may recognize left-recursive invocations.[7]

Recognizing left-recursive invocations is the task of a dedicated parser that must be wrapped around all left-recursive parsers. We do so by annotating recursive parsers with the `!` operator. Since left-recursion requires recursive references (cf. Section 5.1), we can easily check that all such parsers have been properly annotated: a missing annotation will result in an unresolved reference instead of a mystifying infinite loop.

6. Use Case

In this section, we demonstrate the stateful parsing approach with a realistic use case, using our Autumn library. We implement a parser for a simple, yet non-trivial, statically-typed, object-oriented programming language, which we call *Examply*. This imaginary language draws inspiration from Java (its main unit of definition is the class), Kotlin (its postfix type notation and closure notation) and Python (significant indentation). Its full grammar, written with Autumn, can be found online [20]. Examply possesses two common context-sensitive features:

- **Significant whitespace** — Indentation is significant: Like Python, the language does not use curly braces or keywords to delimit blocks of code such as loop or function bodies. Instead, these constructs expect to have their body indented with respect to their first line. Similarly, a decrease in indentation signifies the end of the block. Newlines are also significant, as they are used to separate successive statements and declarations.

[7] We could track the whole state instead of the input position, but we enforce the stronger requirement that a parser invocation has to advance the input position not to be considered left-recursive: unlike other state changes, an increase in input position is a strong proof of progress.

- **Namespace classification** — The parser needs to know which identifiers refer to types. In our language, this is needed because there is an ambiguity between the syntax of function calls — which can receive a closure parameter as an indented block — and the syntax of anonymous classes:

```
val a = myFunction()
    myFunction2()

val b = MyClass()
    var x: Int
    fun foo() { ... }
```

  The body of a class only admits declarations, while the block part of a function call admits all statements (including declarations). These two constructs result in different nodes being added to the abstract syntax tree (AST) produced by the parse.

6.1 Significant Whitespace

We now explain how Examply handles significant whitespace. The code enabling this feature is shown in Figure 3.

The usual whitespace handling strategy is to assume that every parser consumes its trailing whitespace through invocation of the `whitespace` parser. To do so, it is only necessary to ensure that all "primitive" parsers (fullfilling what is traditionally the role of lexical analysis: matching identifiers, literals, keywords and operators) consume their trailing whitespace: then all parsers will do so by transitivity. The `Grammar` class provides some support for this, including lexical analysis emulation (not demonstrated in this paper), and the `[+"lit"]` syntax which evaluates to a parser matching a literal string and any trailing whitespace. Ultimately, this leads to an important guarantee: all parsers are invoked at an input position where no leading whitespace is present.

In order to handle significant whitespace, we maintain two data structures. The first one, `IndentMap`, maps line numbers to two quantities: the input position at which the indentation ends on that line, and the indentation count, which is obtained by expanding tabs to tab stops aligned to multiples of 4. The second data structure, `IndentStack`, is a stack that stores the indentation counts for all enclosing blocks.

`Context.indent` and `Context.istack` are extension properties for the `Context` class and provide syntactic sugar to access the indentation count on the current line, and the indentation stack, respectively.

We build `IndentMap` at the beginning of the parse, through the invocation of the `buildIndentMap` parser. This does not require any special tricks: each parser has access to the whole input (`ctx.text`). The parser does not advance the input position (`ctx.pos`), so that subsequent parsers are free to proceed.

Within the grammar, we use the `indent` parser to require an indented block, and the `dedent` parser to test for the end of an indented block. The implementation of these parsers is straightforward. `indent` checks if the current line is indented with respect to the indentation of the current block (the top of

IndentStack, initialized to 0). If so, it succeeds after pushing the new indentation count onto the stack. dedent checks to see if the indentation count of the current line is less than that of the current block, or if we have reached the end of the input. If so it succeeds, after popping the previous count from the stack.

Finally, the newline parser succeeds if and only if it is invoked at the end of the indentation on the current line or at the end of the input.

```
data class IndentEntry (val count: Int, val end: Int)

class IndentMap: InertState<IndentMap> {
  lateinit var map: Map<Int, IndentEntry>
  fun get(ctx: Context): IndentEntry =
    map[ctx.lineMap.lineFromOffset(ctx.pos)]!!
}

class IndentStack: StackState<Int>()

val Context.indent: IndentEntry
  get() = state(IndentMap::class).get(this)
val Context.istack: IndentStack
  get() = state(IndentStack::class)

val buildIndentMap = Parser { ctx ->
  val map = HashMap<Int, IndentEntry>()
  var pos = 0
  ctx.text.split('\n').forEachIndexed { i, str ->
    val wspace = str.takeWhile {
      it == ' ' || it == '\t' }
    val count = wspace.wspace.expandTabs(4).length
    map.put(i, IndentEntry(count, pos + wspace.length))
    pos += str.length + 1
  }
  ctx.state(IndentMap::class).map = map
  Success
}

val indent = Parser { ctx ->
  val new = ctx.indent.count
  val old = ctx.istack.peek() ?: 0
  if (new > old) Success after { ctx.istack.push(new) }
  else ctx.failure {
    "Expecting indentation > $old positions" }
}

val dedent = Parser { ctx ->
  val new = ctx.indent.count
  val old = ctx.istack.peek() ?: 0
  if (new < old || ctx.pos == ctx.text.length - 1)
    Success after { ctx.istack.pop() }
  else ctx.failure {
    "Expecting indentation < $old positions" }
}

val newline = Predicate {
  indent.end == pos || ctx.pos == ctx.text.length - 1 }
```

Figure 3. Using Autumn to implement significant whitespace handling for Examply.

6.2 Namespace Classification

Examply needs to distinguish between types (i.e., class names) and other identifiers at parse-time, in order to resolve ambiguities. We call this process *namespace classification*: we want to know whether an identifier belongs to the namespace of types or not. Such parse-time tracking is reminiscent of the C language, and is seldom seen in modern languages. In Examply, we could avoid it by adding a keyword (e.g., new) to disambiguate constructor invocation from function invocation. Or we could perform AST-disambiguation passes after parsing. We stress that Examply is designed to showcase the strength of the principled stateful approach, and notably its ability to deal with the quirks of existing languages.

To understand the logistics of namespace classification, we must first define how our imaginary language handles type references:

1. Types are always referenced through a single identifier, except within imports where they are preceded by a package string.
2. A class name can only be referenced after or within its definition.
3. A class definition can appear anywhere other declarations can: at the top-level, within another class, or within a code block.
4. Class definitions are lexically scoped: a class has access to all imported classes and to all classes defined before it within one of its outer scopes (class body or code block).
5. A class has access to all classes defined within its superclass and other ancestors.
6. A class cannot inherit from one of its outer classes.
7. In order to avoid ambiguous type names (for instance, both a class defined in an outer class and a class defined in a superclass could bear the same name), Examply features type aliases that assign an alternate name to an existing type. Type aliases can appear anywhere a class definition can appear, and have the same visibility as class definitions.

Figure 4 shows the code handling namespace classification. We do not engage with its minutiae, but instead give a high-level description of its operation. The code itself should demonstrate that the implementation of these ideas is terse and readable, even to those who ignore the precise semantics of some operations. We note however that in the code, ctx.stack refers to the stack used to construct AST nodes. We occasionally peek in this stack in order to retrieve identifiers.

The main data structure is the TypeStack State instance. It holds a stack of Types, which are pairs formed by a string and a list of other Types. Intuitively, each Type instance represents a class name alongside with a list of classes accessible through it: its inner classes and the inner classes of its ancestors. We call these classes the *private classes* of a class: they cease being accessible once the class definition ends. We define two helper functions over the type stack: isType checks if an identifier refers to a type, and priv returns the private classes of the named class, or an empty list if no such class exists.

Each time we encounter a new type during the parse, it is pushed onto the type stack. This is the the task of the NewType parser, which is applied to identifiers introduced by classes and type aliases. A parameter controls whether the new type is an alias, in which case it inherits the private classes of the aliased class. Note that classes start with an empty list of private classes. This will be updated once the class definition is complete.

Once a scope (a class body, or some code block) is exited, the types introduced within it are not longer accessible. To enforce this, we use the `Scoped` parser: it saves the type stack size, invokes its child parser (corresponding to a scope), then removes any extraneous items from the type stack.

If the scoped body was a class body, the `Type` representing the class on the stack must be updated with a list of its private classes, so that an inner class may access them. This is the role of the `ClassDef` parser. It looks up the class and superclass names on the AST stack, then looks up the list of private classes of the superclass. If found, this list is pushed on the type stack. All of this is done after taking a snapshot of type stack. Subsequently, the `body` parser is invoked and, if successful, a delta of the type stack is generated using the snapshot. This delta corresponds to the private classes of the class, including those introduced by its superclass. The snapshot is restored and the topmost entry on the type stack (which represents the class) is removed and replaced with a new one that binds the class name to its private classes.

A reduced version of this process also needs to happen for anonymous classes: they need to access their superclass' private classes, but no `Type` record must be created for them. The `anonClassInherit` parser fullfills this role, by reusing the `inherit` function.

Finally, to resolve the ambiguity, we use the `classGuard` parser: it performs a lookahead, attempting to match an identifier, and succeeding only this identifier refers to a type.

We defer our assessment of the approach until Section 6.4.

6.3 Putting it all together

To illustrate the use of significant whitespace and namespace classification (as presented in sections 6.1 and 6.2), let's look at two short examples. First, here is how we define an indented code block:

```
val statements =
  Seq(indent, Scoped(!"statement" until dedent))
      .collect<Stmt>()
```

The block starts by an increase in indentation (`indent`), and ends when a decrease in indentation is encountered (`dedent`), parsing statements in the mean time (`!"statement" until dedent`). The `collect` part instructs the parser to collect all statement nodes pushed onto the AST stack and to aggregate them in a list, which is itself pushed onto that stack. We also see that all indented statements form a scope (`Scoped`) in the sense of Section 6.2.

Second, here is how the parser for "block constructors" (i.e., anonymous classes) is defined:

```
val blockCtorBody = Scoped(Seq(anonClassInherit, decls))
val blockCtor
  = Seq(classGuard, iden, paramList, blockCtorBody)
    .build { CtorCall(get(), get(), get()) }
```

```
data class Type (val name: String, val priv: LinkList<Type>)
class TypeStack: MonotonicStack<Type>()
val Context.types: TypeStack
    get() = state(TypeStack::class)

fun isType(ctx: Context, iden: String): Boolean
  = ctx.types.stream().any { it.name == iden }

fun priv(ctx: Context, iden: String): LinkList<Type>
  = ctx.types.stream()
    .filter { it.name == iden }
    .next() ?.priv ?: LinkList()

fun NewType (child: Parser, alias: Boolean = false)
= Parser { ctx ->
    child.parse(ctx) andDo {
      val name = ctx.stack.peek() as String
      val priv = if (alias) priv(ctx, name)
                 else LinkList()
      ctx.types.push(Type(name, priv))
  } }

fun Scoped(body: Parser) = Parser { ctx ->
  val size = ctx.types.size
  body.parse(ctx) andDo { ctx.types.truncate(size) }
}

fun inherit(ctx: Context, name: String)
  = priv(ctx, name).stream().each { ctx.types.push(it) }

fun ClassDef (body: Parser)
= Parser { ctx ->
    val parent = ctx.stack.at(0) as Maybe<SimpleType>
    val name = ctx.stack.at(1) as String
    val snapshot = ctx.types.snapshot()
    if (parent is Some<SimpleType>)
      inherit(ctx, parent.value.name)
    body.parse(ctx) andDo {
      val diff = ctx.types.diff(snapshot)
      ctx.types.restore(snapshot)
      ctx.types.pop()
      ctx.types.push(Type(name, diff))
} }

val anonClassInherit = Perform { ctx ->
  inherit(ctx, ctx.stack.at(1) as String) }

val classGuard = Seq(iden, Predicate { ctx ->
  isType(ctx, ctx.stack.peek() as String) }
).ahead
```

Figure 4. Using Autumn to implement namespace classification for Examply.

The parser is simply guarded with the `classGuard` parser, which checks if there is an identifier at the current input position, and whether this identifier refers to a type. The body of the class can access the superclass' private classes through `anonClassInherit`. The `Scope` wrapper ensures that this access is restricted to the class and does not spread to the code that follows. `decls` refers to an indented block of declarations.

6.4 Discussion

We have implemented two context-sensitive features in for an imaginary but non-trivial programming language. With the code we have shown so far, the rest of the grammar is trivially able to define code delimited by changes in indentation, or by newlines (Section 6.1); or to direct the parse depending on whether an identifier refers to a type (Section 6.2).

All this, by itself, is no mean feat. There are few parsing tools where this is possible to begin with (most of them are presented in Section 2). Significant indentation handling, in particular, is non-trivial because Autumn does not include a lexical analysis layer out of the box.

We also underline that the presented implementations are rather terse, less than 50 lines of code each. Some of that does come from our choice of implementation language, but it also shows that the principled stateful parsing approach does not impose a significant boilerplate overhead. In particular, the ability to reuse state-handling strategies, such as `MonotonicStack` means that *context-transparency* comes almost for free.

The state manipulations operations from sections 4.1 and 4.2.3 are strangely discreet in our examples. Significant whitespace handling does not use them at all, while namespace classification performs a `diff` in order to capture the types introduced by a class body. But because they do not appear in the code does not mean the operations are not used, they are simply hidden from view. The basic contract of principled stateful parsing is that each parser either succeeds or leaves the state untouched. You can convince yourself that all the parsers and parser combinators we introduced satisfy this condition, either by reusing existing combinators, or by delegating the responsibility for this to their single subparser.

It remains that the parsers we introduce do get backtracked over during the parse. As such, their state mostly get saved and restored by other parsers that invoke them, directly or indirectly. It is in fact crucial for the state to get restored when backtracking occurs: we need to know the correct indentation level whenever we backtrack out of a block; we also need to know which identifiers are classes, even when backtracking over a type definition. Granted, given that most constructs in the grammar are guarded by specific keywords, such backtracking occurences should be rare. However, unlike the other, often *accidentally* stateful parsers (cf. Section 2), context transparency ensures that we can evolve the grammar as we see fit, without fear of breaking the mechanisms we just introduced. It is also a pre-requisite for safe grammar composition.

In fact, the scarcity of state operations is a boon: it means that the benefits of our approach come at very little cost, at least implementation-wise. We would also disabuse the reader of the notion that the Autumn codebase hides some devilish complexity in order to make up for this: the whole library [20] is less than 2500 lines of code. All pre-defined parsers live in a single file of less than 500 lines. This file defines around 50 parsers: those corresponding to all basic PEG [8] operators, as well as numerous extensions, notably to work with error messages, AST nodes, ...

6.5 Performance

Performance has not been our focal point, but preliminary testing seems to indicate that performance is within an order of magnitude of mainstream parsing tools such as Rats! [9] and Parboiled [7] for context-free grammars. The implementation currently incurs overhead even for context-free grammars, which we are working to reduce. The overhead scales with the amount of state in use, depending on the `State` implementation details. The costly operations are the creation of snapshots and deltas. In general, memory allocations tend to be the bottleneck, so increased sharing between snapshots results in better performances. Indeed, we've had success with purely functional data structures [23].

Just like PEG parsing without full memoization, parsing has exponential complexity in the worst case. In practice however run times are acceptable, as programming language grammars are fairly deterministic.

7. Conclusion

In this paper, we proposed an approach to tackle the problem of context-sensitive parsing. Our solution, unlike existing ones, possesses the property of *context transparency*: grammatical constructs are unaware of the context shared between their ancestors and their descendants, making it easier to write, evolve and compose context-sensitive grammars.

We proceeded in two parts. First, we allowed parsers to manipulate a mutable data store, so as to enable context-sensitivity through *recall*. Second, we required parsers to behave transactionally: a parser must either succeed, or fail and leave the state unaltered. This transactional discipline, which we call *principled stateful parsing*, prevents parsing mechanisms such as backtracking and memoization to break the guarantee of context transparency.

To enforce the principled stateful parsing discipline, we supplied formally specified state manipulation operations. The role of these operations is to snapshot and restore the state, as well as to create and merge deltas between a snapshot and the current state.

We implemented our approach in a parsing library called *Autumn*, and showed how it can be used in practice to implement common context-sensitive grammar features such as significant whitespace and namespace classification. We underline the low boilerplate and conceptual overhead introduced by the approach.

We view this work as a first step towards bringing disciplined grammarware engineering (as defined by Klint, Lämmel and Verhoef [17]) to context-sensitive parsing.

References

[1] A. Afroozeh and A. Izmaylova. One parser to rule them all. In *ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2015, pages 151–170. ACM, 2015.

[2] R. Atkey. The semantics of parsing with semantic actions. In *Proceedings of the 27th Annual IEEE/ACM Symposium on Logic in Computer Science*, LICS 2015, pages 75–84. IEEE Computer Society, 2012.

[3] J. Aycock and R. N. Horspool. Schrödinger's token. *Software: Practice and Experience*, 31(8):803–814, 2001.

[4] P. Bagwell. Ideal hash trees. Technical Report LAMP-REPORT-2001-001, Ecole polytechnique fédérale de Lausanne, 2001.

[5] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.

[6] N. Chomsky. Formal properties of grammar. In *Handbook of Mathematical Psychology*, chapter 12, pages 360–363 and 367. Wiley, 1963.

[7] M. Doenitz. The Parboiled homepage, 2015. `https://github.com/sirthias/parboiled`.

[8] B. Ford. Parsing expression grammars: A recognition-based syntactic foundation. *SIGPLAN Notices*, 39(1):111–122, Jan. 2004.

[9] R. Grimm. Better extensibility through modular syntax. *SIGPLAN Notices*, 41(6):38–51, June 2006.

[10] D. Grune and C. J. Jacobs. *Parsing Techniques: A Practical Guide, p. 21–23*. Springer, 2nd edition, 2008.

[11] G. Hedin. Reference attributed grammars. *Informatica (Slovenia)*, 24(3), 2000.

[12] G. Hutton and E. Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.

[13] R. Ierusalimschy. A text pattern-matching tool based on parsing expression grammars. *Software: Practice and Experience*, 39(3):221–258, Mar. 2009.

[14] T. Jim and Y. Mandelbaum. A new method for dependent parsing. In *Proceedings of the 20th European Conference on Programming Languages and Systems*, ESOP 2011, pages 378–397. Springer, 2011.

[15] T. Jim, Y. Mandelbaum, and D. Walker. Semantics and algorithms for data-dependent grammars. *SIGPLAN Notices*, 45(1):417–430, Jan. 2010.

[16] L. Kallmeyer. *Parsing Beyond Context-Free Grammars*. Springer, 2010.

[17] P. Klint, R. Lämmel, and C. Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering and Methodology*, 14(3):331–380, July 2005.

[18] D. Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145, 1968.

[19] N. Laurent and K. Mens. Parsing expression grammars made practical. In *Proceedings of the ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2015, pages 167–172. ACM, 2015.

[20] N. Laurent and K. Mens. Taming context-sensitive languages with principled stateful parsing: Artifacts. *Software Language Engineering: Artifacts Track*, 2016. `https://github.com/ncellar/sle2016`.

[21] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-35, Department of Information and Computing Sciences, Utrecht University, 2001.

[22] A. Moors, F. Piessens, and M. Odersky. Parser combinators in Scala. Technical Report CW491, Department of Computer Science, KU Leuven, Feb. 2008.

[23] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA, 1998.

[24] T. Parr, S. Harwell, and K. Fisher. Adaptive LL(*) parsing: The power of dynamic analysis. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 579–598. ACM, 2014.

[25] F. Pereira and D. Warren. Readings in natural language processing. chapter Definite Clause Grammars for Language Analysis, pages 101–124. Morgan Kaufmann, 1986.

[26] J. M. Spivey and J. Abrial. *The Z notation*. Prentice Hall, 1992.

[27] M. J. Steindorfer and J. J. Vinju. Optimizing hash-array mapped tries for fast and lean immutable jvm collections. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 783–800. ACM, 2015.

[28] The Free Software Foundation. The GNU Bison homepage, 2014. `http://www.gnu.org/software/bison/`.

[29] A. D. Thurston and J. R. Cordy. A backtracking LR algorithm for parsing ambiguous context-dependent languages. In *Proceedings of the 2006 conference of the Centre for Advanced Studies on Collaborative Research, October 16-19, 2006, Toronto, Ontario, Canada*, pages 39–53, 2006.

[30] E. Van Wyk and A. Schwerdfeger. Context-aware scanning for parsing extensible languages. In *International Conference on Generative Programming and Component Engineering, GPCE 2007*. ACM, October 2007.