Parsing Expression Grammars for Structured Data

Fabio Mascarenhas[1], Sérgio Medeiros[2], Roberto Ierusalimschy[3]

[1] Department of Computer Science – UFRJ – Rio de Janeiro – Brazil
fabiom@dcc.ufrj.br
[2] Department of Computer Science – UFS – Aracaju – Brazil
sergio@ufs.br
[3] Department of Computer Science – PUC-Rio – Rio de Janeiro – Brazil
roberto@inf.puc-rio.br

Abstract. Parsing Expression Grammars (PEGs) are a formalism for language recognition that renewed academic interest in top-down parsing approaches. LPEG is an implementation of the PEG formalism which compiles PEGs to instructions of a virtual parsing machine, preserving PEG semantics. The LPEG parsing machine has a formal model, and the transformation of PEGs to this model has been proven correct. In this paper, we extend both the PEG formalism and LPEG's parsing machine so they can match structured data instead of just strings of symbols. Our extensions are conservative, and we prove the correctness of the translation from extended PEGs to programs of the extended parsing machine. We also present benchmarks that show that the performance of the extended parsing machine for structured data is competitive with the performance of equivalent hand-written parsers.

Keywords: parsing expression grammars, parsing machine, pattern matching, natural semantics, tree grammars

1 Introduction

Parsing Expression Grammars (PEGs) [1] are an alternative formalism for describing a language's syntax. Unlike Context-Free Grammars, PEGs are unambiguous by construction, and their standard semantics is based on recognizing strings instead of generating them. A PEG can be considered both the specification of a language and the specification of a top-down parser for that language. The core of the PEG formalism is a form of limited backtracking via *ordered choice*: the parser, when faced with several alternatives, will try them in a deterministic order (left to right), discarding remaining alternatives after one of them succeeds. PEGs also introduce a more expressive syntax for grammatical productions, based on the syntax of regexes, and *syntactic predicates*, a form of unrestricted lookahead where the parser checks whether a parsing expression matches the rest of the input without consuming it.

LPEG [2] is a pattern-matching tool for the Lua language [3, 4]. LPEG uses PEGs to describe patterns instead of the more popular Perl-like "regular expressions" (regexes). The implementation of LPEG uses a *virtual parsing machine*, where each pattern translates to a program for this machine. LPEG builds these programs at runtime, dynamically composing smaller programs into bigger programs. The parsing machine uses a stack to manage the information necessary to implement PEG's limited backtracking. A formal model of the parsing machine is available, along with a correctness proof of the translation from PEGs to programs of the machine [5].

Both the original PEG formalism and LPEG's parsing machine assume that the subject a PEG recognizes is a string of characters. In this paper, we extend the PEG formalism and the model of LPEG's parsing machine so they can also parse structured data in the form of Lisp-style lists (a possibly empty list where each element can be an atom or another list). Our extensions are conservative, so the behavior of PEGs when parsing strings is unaffected, and we give correctness proofs for the translation from our extensions to PEGs to their counterparts in the parsing machine.

Extending PEGs to parse structured data makes PEGs more useful for language prototyping [6]. A PEG-based scannerless parser can construct an abstract syntax tree, and PEG-based tree walkers and transformers can implement analysis and optimization passes on this tree. A PEG-based tree walker can also do naive code generation to intermediate languages or machine code. Several applications also need to handle structured data in the form of DOM (document object model) trees obtained from XML or HTML data, and PEGs can be used to extract information from these trees with ad-hoc parsers.

We implemented our extensions as modifications to LPEG, and used this implementation to compare the performance of LPEG with our list extensions against another PEG-based parser and against hand-written tree matchers.

The rest of this paper is organized as follows: Section 2 explains the intuition behind list parsing using PEGs, gives an example, and extends the PEG formalism with list patterns; Section 3 extends the formal model of LPEG's parsing machine, and gives a translation from the extended PEG formalism to instructions of the extended parsing machine; Section 4 describes an optimization for list patterns in LPEG's parsing machine; Section 5 benchmarks our implementation of list parsing on LPEG; Section 6 reviews some related work; finally, Section 7 summarizes our results.

2 Extending PEGs for Lists

A Lisp-style list is a recursive data structure inductively defined by the binary operator *:* (*cons*). The empty list is ε, $a{:}l$ is a list if a is an *atom* (any object that is not a list, such as a character, a string, a number etc.) and l is a list, and $l_1{:}l_2$ is a list if both l_1 and l_2 are lists. The first argument of : is the *head* of the list and the second argument is the *tail*. We will use $\{e_1 e_2 \ldots e_n\}$ as another way to write $e_1{:}e_2{:}\ldots{:}e_n{:}\varepsilon$, where each e_i can be an atom or a list. It is straightforward to represent any tree as a list. We will also use $l_1 l_2$ to denote the concatenation of lists l_1 and l_2, which is the list of all elements of l_1 followed by all elements of l_2.

Parsing expressions are also defined inductively as the empty expression ε, the *any* expression ., a terminal symbol a, a non-terminal symbol A, a concatenation $p_1 p_2$ of two parsing expressions p_1 and p_2, an ordered choice p_1/p_2 between two parsing expressions p_1 and p_2, a repetition p^* of a parsing expression p, or a not-predicate $!p$ of a parsing expression p. A PEG is then a tuple (V, T, P, p_S), where V and T are sets of non-terminals and terminals, respectively, P is a function from non-terminals to parsing expressions, and p_S is the initial parsing expression of the PEG.

Intuitively, ε just succeeds and leaves the subject unaffected; the expression . matches and consumes any terminal and fails when we have reached the end of the subject; a matches and consumes itself and fails otherwise; A tries to match the expression $P(A)$; $p_1 p_2$ tries to match p_1, if it succeeds then tries to match p_2 on part of the subject that p_1 did not consume; p_1/p_2 tries to match p_1, and if it fails tries to match p_2; p^* repeatedly tries to match p until it fails, consuming as much of the subject as it can; finally, $!p$ tries to match p and fails if p succeeds and succeeds if p fails, in any case leaving the subject unaffected. It is easy to see that the result of a match is either failure or a suffix of the subject (not a proper suffix, as the expression may succeed without consuming anything).

Our extension consists of using the set of terminals T as the set of atoms, and having the subject of the match be a list instead of a string of terminals. The result of the match still is a suffix of the subject (which is also a list) or `fail`, a match failure. We also add a new pattern, $\{p\}$, which tries to match the head of the subject, which must be a list, against the pattern p. If p consumes the whole list then the result of matching $\{p\}$ is the tail of the subject. If p does not consume the whole list, or the head of the subject is an atom, then the result of matching $\{p\}$ is `fail`. The abstract syntax of our extended parsing expressions is given below:

$$p \;=\; \varepsilon \;\big|\; . \;\big|\; a \;\big|\; A \;\big|\; p_1\, p_2 \;\big|\; p_1/p_2 \;\big|\; p^* \;\big|\; !p \;\big|\; \{p\}$$

As an example, assume that we want to validate a tree representation of a XML document according to the following DTD (Document Type Definition), which specifies that the document is a `<recipe>` tag with a body composed of a `<title>` tag followed by zero or more `<ingredient>` tags followed by a `<preparation>` tag and optionally by a `<comment>` tag:

```
<!DOCTYPE recipe [
<!ELEMENT recipe (title,ingredient*,preparation,comment?)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT ingredient (ingredient*,preparation)?>
<!ATTLIST ingredient name CDATA #REQUIRED
                     amount CDATA #IMPLIED>
<!ELEMENT preparation (step*)>
<!ELEMENT step (#PCDATA)>
<!ELEMENT comment (#PCDATA)>
]>
```

A `#PCDATA` body means that the body is just plain text (which will be an atom in the tree representation), and `!ATTLIST` lists the attributes of a tag, where `#REQUIRED` means a required attribute and `#IMPLIED` an optional one. In the tree representation, each tag is a list of the tag name (without brackets), a list of attributes, if the tag has attributes, ordered by name, where each attribute is a list of the attribute name and the attribute data (an atom), followed by the body, which is either an atom or a list of tags. For example, take the XML document below:

```
<recipe>
  <title>Caffe Latte</title>
  <ingredient name = "milk" amount = "2 cups" />
  <ingredient name = "espresso" />
  <preparation>
    <step>Heat milk in a saucepan.</step>
    <step>Whisk briskly to create foam.</step>
    <step>Brew espresso.</step>
    <step>Pour in milk.</step>
  </preparation>
</recipe>
```

This XML document has the following tree representation:

$$\begin{array}{l}
\{\text{“recipe”} \\
\quad \{\{\text{“title” “Caffe Latte”}\} \\
\quad\ \ \{\text{“ingredient”}\{\{\text{“amount” “2cups”}\}\{\text{“name” “milk”}\}\}\} \\
\quad\ \ \{\text{“ingredient”}\{\{\text{“name” “espresso”}\}\}\} \\
\quad\ \ \{\text{“preparation”} \\
\qquad\ \ \{\{\text{“step” “Heat milk in a saucepan.”}\} \\
\qquad\ \ \ \ \{\text{“step” “Whisk briskly to create foam.”}\} \\
\qquad\ \ \ \ \{\text{“step” “Brew espresso.”}\} \\
\qquad\ \ \ \ \{\text{“step” “Pour in milk.”}\}\}\}\}
\end{array}$$

We can validate the tree representation against the DTD with the following extended PEG:

$$\begin{aligned}
\text{recipe} &\rightarrow \{\text{“recipe” } \{\text{title ingredient}^* \text{ preparation (comment}/\varepsilon)\}\}\ !. \\
\text{title} &\rightarrow \{\text{“title” atom}\} \\
\text{ingredient} &\rightarrow \{\text{“ingredient” } \{((\{\text{“amount” atom}\}/\varepsilon)\ \{\text{“name” atom}\}\} \\
&\qquad\qquad\qquad\quad (\{\text{ingredient}^* \text{ preparation}\}/\varepsilon)\} \\
\text{preparation} &\rightarrow \{\text{“preparation” } \{\text{step}^*\}\}
\end{aligned}$$

$$\begin{aligned}
\text{step} &\rightarrow \{\text{“step” atom}\} \\
\text{comment} &\rightarrow \{\text{“comment” atom}\} \\
\text{atom} &\rightarrow\ !\{.^*\}\ .
\end{aligned}$$

In the PEG above, quoted strings represent atom patterns, and we use parenthesis to avoid having to worry about precedence of concatenation over ordered choice. It is easy to check that the PEG matches any document that is valid according to the DTD, and fails on any document that is not, assuming it is in the tree representation we described above.

Figure 1 formalizes our extensions with natural semantics for $\overset{\text{PEG}}{\rightsquigarrow}$, the matching relation for PEGs with list expressions. The semantics is presented as a set of inference rules, and $G[p]\,xy \overset{\text{PEG}}{\rightsquigarrow} y$ if and only if there is a finite proof tree for it. The notation $G[p'_S]$ denotes a new grammar (V, T, P, p'_S) that is equal to G except for the initial parsing expression p_S, which is replaced by p'_S.

Rules *list*.1, *list*.2, and *list*.3 formalize the notion that a list pattern $\{p\}$ only succeeds if the head of the subject is also a list and if p exactly matches this other list. Thus the pattern $\{ab\}$ matches the list $\{ab\}$ but not the atom a, the list $\{ac\}$, or the list $\{abc\}$.

It is straightforward to see that our extension is conservative. Let us define $map_{s\to l}$ as a mapping from strings to lists, where the set of atoms is the alphabet of the strings, with the following inductive definition:

$$\begin{aligned}
map_{s\to l}(\varepsilon) &= \varepsilon \\
map_{s\to l}(as) &= a : map_{s\to l}(s)
\end{aligned}$$

We then have the following lemma, where $\rightsquigarrow$ is the standard matching semantics for PEGs. It proves that our extended semantics is identical to the standard semantics when the subject is either the empty list or a list where all the elements are atoms.

Lemma 1. *Given a PEG G and a parsing expression p without list expressions, $G[p]\,s_1s_2 \rightsquigarrow s_2$ if and only if $G[p]\,map_{s\to l}(s_1s_2) \overset{PEG}{\rightsquigarrow} map_{s\to l}(s_2)$ and $G[p]\,s \rightsquigarrow$ `fail` if and only if $G[p]\,map_{s\to l}(s) \overset{PEG}{\rightsquigarrow}$ `fail`.*

Proof. By induction on the height of the respective proof trees.

3 A Parsing Machine for Lists

The core of LPEG, an implementation of PEGs for the Lua language [2], is a virtual parsing machine. LPEG compiles each parsing expression to a program of the parsing machine, and the program for a compound expression is a combination of the programs for its subexpressions. The parsing machine has a formal model of its operation, and proofs that compiling a PEG yields a program that is equivalent to the original PEG [5].

Empty $\dfrac{}{G[\varepsilon]\ \ l \overset{\text{PEG}}{\leadsto} l}$ **(empty.1)** **Any** $\dfrac{}{G[.]\ \ x{:}l \overset{\text{PEG}}{\leadsto} l}$ **(any.1)** $\dfrac{}{G[.]\ \ \varepsilon \overset{\text{PEG}}{\leadsto} \texttt{fail}}$ **(any.2)**

Variable $\dfrac{G[P(A)]\ \ l \overset{\text{PEG}}{\leadsto} X}{G[A]\ \ l \overset{\text{PEG}}{\leadsto} X}$ **(var.1)**

Terminal $\dfrac{}{G[a]\ \ a{:}l \overset{\text{PEG}}{\leadsto} l}$ **(term.1)** $\dfrac{}{G[a]\ \ b{:}l \overset{\text{PEG}}{\leadsto} \texttt{fail}}$, $b \neq a$ **(term.2)**

$\dfrac{}{G[a]\ \ \varepsilon \overset{\text{PEG}}{\leadsto} \texttt{fail}}$ **(term.3)** $\dfrac{}{G[a]\ \ l_1{:}l_2 \overset{\text{PEG}}{\leadsto} \texttt{fail}}$ **(term.4)**

Concatenation $\dfrac{G[p_1]\ \ l_1l_2 \overset{\text{PEG}}{\leadsto} l_2 \qquad G[p_2]\ \ l_2 \overset{\text{PEG}}{\leadsto} X}{G[p_1\, p_2]\ \ l_1l_2 \overset{\text{PEG}}{\leadsto} X}$ **(con.1)** $\dfrac{G[p_1]\ \ l \overset{\text{PEG}}{\leadsto} \texttt{fail}}{G[p_1\, p_2]\ \ l \overset{\text{PEG}}{\leadsto} \texttt{fail}}$ **(con.2)**

Ordered Choice $\dfrac{G[p_1]\ \ l_1l_2 \overset{\text{PEG}}{\leadsto} l_2}{G[p_1\ /\ p_2]\ \ l_1l_2 \overset{\text{PEG}}{\leadsto} l_2}$ **(ord.1)** $\dfrac{G[p_1]\ \ l \overset{\text{PEG}}{\leadsto} \texttt{fail} \qquad G[p_2]\ \ l \overset{\text{PEG}}{\leadsto} X}{G[p_1\ /\ p_2]\ \ l \overset{\text{PEG}}{\leadsto} X}$ **(ord.2)**

Not Predicate $\dfrac{G[p]\ \ l \overset{\text{PEG}}{\leadsto} \texttt{fail}}{G[!p]\ \ l \overset{\text{PEG}}{\leadsto} l}$ **(not.1)** $\dfrac{G[p]\ \ l_1l_2 \overset{\text{PEG}}{\leadsto} l_2}{G[!p]\ \ l_1l_2 \overset{\text{PEG}}{\leadsto} \texttt{fail}}$ **(not.2)**

Repetition $\dfrac{G[p]\ \ l \overset{\text{PEG}}{\leadsto} \texttt{fail}}{G[p^*]\ \ l \overset{\text{PEG}}{\leadsto} l}$ **(rep.1)** $\dfrac{G[p]\ \ l_1l_2l_3 \overset{\text{PEG}}{\leadsto} l_2l_3 \qquad G[p^*]\ \ l_2l_3 \overset{\text{PEG}}{\leadsto} l_3}{G[p^*]\ \ l_1l_2l_3 \overset{\text{PEG}}{\leadsto} l_3}$ **(rep.2)**

List $\dfrac{G[p]\ \ l_1 \overset{\text{PEG}}{\leadsto} \varepsilon}{G[\{p\}]\ \ l_1{:}l_2 \overset{\text{PEG}}{\leadsto} l_2}$ **(list.1)** $\dfrac{G[p]\ \ l_1 \overset{\text{PEG}}{\leadsto} X}{G[\{p\}]\ \ l_1{:}l_2 \overset{\text{PEG}}{\leadsto} \texttt{fail}}$, $X \neq \varepsilon$ **(list.2)**

$\dfrac{}{G[\{p\}]\ \ a : l \overset{\text{PEG}}{\leadsto} \texttt{fail}}$ **(list.3)** $\dfrac{}{G[\{p\}]\ \ \varepsilon \overset{\text{PEG}}{\leadsto} \texttt{fail}}$ **(list.4)**

Fig. 1. Natural Semantics of PEGs Extended with List Patterns

The parsing machine has a register to hold the program counter used to address the next instruction to execute, a register to hold the subject, and a stack that the machine uses for pushing call frames and backtrack frames. A call frame is just a return address for the program counter, and a backtrack frame is an address for the program counter and a subject. The machine's instructions manipulate the program counter, subject, and stack.

The compilation of the following PEG (for a sequence of zero or more as followed by any character other than a followed by a b) uses all the basic instructions of the parsing machine:

$$\begin{aligned} A &\rightarrow\ !a\,.\,B\ /\ aA \\ B &\rightarrow b \end{aligned}$$

This PEG compiles to the following program:

```
    Call A
A:  Choice A1
    Choice A2
    Char a
    Commit A3
A3: Fail
A2: Any
    Call B
    Commit A4
A1: Char a
    Jump A
A4: Return
B:  Char b
    Return
```

The behavior of each instruction is straightforward: `Call` pushes a call frame with the address of the next instruction and jumps to a label, `Return` pops a call frame and jumps to its address, `Jump` is an unconditional jump, `Char` tries to match a character with the start of the subject, consuming the first character of the subject if successful, `Any` consumes the first character of the subject (failing if the subject is ε), `Choice` pushes a backtrack frame with the subject and the address of the label, `Commit` discards the backtrack frame in the top of the stack and jumps, and `Fail` forces a failure. When the machine enters a failure state it pops call frames from the stack until reaching a backtrack frame, then pops this frame and resumes execution with the subject and address stored in it.

To extend the parsing machine for list subjects and patterns, we replaced the `Char` instruction with an equivalent `Atom` instruction, and extended the stack to also store *list frames*; a list frame saves the tail of the current subject in the stack before matching a list pattern against its head. We also added two new instructions that correspond to beginning and finishing a list pattern:

`Open` pushes a list frame on the stack, then sets the head of the subject as the new subject. Fails if the head of the subject is not a list.

`Close` pops the top entry from the stack (a list frame) if the subject is the empty list, setting the subject saved in the frame as the new subject. Fails if the subject is not the empty list.

A list pattern $\{p\}$ compiles to an `Open` instruction followed by the compilation of p and a `Close` instruction.

Formally, the program counter register, the subject register, and the stack form a machine state. We represent it as a tuple $\mathcal{N} \times \text{List} \times \text{Stack}$, in the order above. A machine state can also be a failure state, represented by $\textbf{Fail}\langle e\rangle$, where e is the stack. Stacks are lists of $(\mathcal{N} \times \text{List}) \cup \text{List} \cup \mathcal{N}$, where $\mathcal{N} \times \text{List}$ represents a backtrack frame, List represents a list frame, and $\mathcal{N}$ represents a call frame.

Figure 2 presents the operational semantics of the extended parsing machine as a relation between machine states. The program $\mathcal{P}$ that the machine executes is implicit. The relation $\xrightarrow{\text{Instruction}}$ relates two states when pc in the first state addresses a instruction matching the label, and the guard (if present) is valid.

The formal model of the LPEG parsing machine [5] represents the compilation process using a transformation function Π, where $\Pi(G, x, p)$ is the translation of pattern p in the context of the grammar G, where x is the position where the program starts relative to the start of the compiled grammar. It also uses the notation $|\Pi(G, x, p)|$ to mean the number of instructions in the program $\Pi(G, x, p)$. In our extended model, we have

$$\begin{aligned}\Pi(G, x, \{p\}) \equiv\ & \texttt{Open} \\ & \Pi(G, x+1, p) \\ & \texttt{Close}\end{aligned}$$

as the definition of Π for list patterns and $\Pi(G, x, a) \equiv \texttt{Atom}\, a$ as the definition of Π for atom patterns.

The following lemma is the correctness lemma for our extended model of the parsing machine.

Lemma 2. *Given a grammar G and a parsing expression p, if $G[p]\, l_1 l_2 \overset{\text{PEG}}{\rightsquigarrow} l_2$ then $\langle pc,\, l_1 l_2,\, e\rangle \xrightarrow{\Pi(G,x,p)} \langle pc + |\Pi(G,x,p)|,\, l_2,\, e\rangle$, and if $G[p]\, l \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$ then $\langle pc,\, l,\, e\rangle \xrightarrow{\Pi(G,x,p)} \textbf{Fail}\langle e\rangle$.*

Proof. By induction on the derivation trees for $G[p]\, l_1 l_2 \overset{\text{PEG}}{\rightsquigarrow} l_2$ or $G[p]\, l \overset{\text{PEG}}{\rightsquigarrow} \texttt{fail}$, as applicable. Most cases are identical to the cases in the correctness proof for the original definition of Π. We just need to prove cases *list*.1, *list*.2, and *list*.3.

For case *list*.1, we have $\langle pc,\, l_1,\, e\rangle \xrightarrow{\Pi(G,x,p)} \langle pc + |\Pi(G,x,p)|,\, \varepsilon,\, e\rangle$ by the induction hypothesis. So the machine executes the following sequence of transitions for $\Pi(G, x, \{p\})$:

$$\langle pc,\, l_1 : l_2,\, e\rangle$$

$$
\begin{aligned}
\langle pc,\, a\!:\!l,\, e\rangle &\xrightarrow{\texttt{Atom } a} \langle pc+1,\, l,\, e\rangle\\
\langle pc,\, b\!:\!l,\, e\rangle &\xrightarrow{\texttt{Atom } a} \mathbf{Fail}\langle e\rangle, a \neq b\\
\langle pc,\, \varepsilon,\, e\rangle &\xrightarrow{\texttt{Atom } a} \mathbf{Fail}\langle e\rangle\\
\langle pc,\, a\!:\!l,\, e\rangle &\xrightarrow{\texttt{Any}} \langle pc+1,\, l,\, e\rangle\\
\langle pc,\, \varepsilon,\, e\rangle &\xrightarrow{\texttt{Any}} \mathbf{Fail}\langle e\rangle\\
\langle pc,\, l,\, e\rangle &\xrightarrow{\texttt{Choice } i} \langle pc+1,\, l,\, (pc+i, l)\!:\!e\rangle\\
\langle pc,\, l,\, e\rangle &\xrightarrow{\texttt{Jump } i} \langle pc+i,\, l,\, e\rangle\\
\langle pc,\, l,\, e\rangle &\xrightarrow{\texttt{Call } i} \langle pc+i,\, l,\, (pc+1)\!:\!e\rangle\\
\langle pc_1,\, l,\, pc_2\!:\!e\rangle &\xrightarrow{\texttt{Return}} \langle pc_2,\, l,\, e\rangle\\
\langle pc,\, l,\, h\!:\!e\rangle &\xrightarrow{\texttt{Commit } i} \langle pc+i,\, l,\, e\rangle\\
\langle pc,\, l,\, e\rangle &\xrightarrow{\texttt{Fail}} \mathbf{Fail}\langle e\rangle\\
\mathbf{Fail}\langle pc\!:\!e\rangle &\xrightarrow{\quad} \mathbf{Fail}\langle e\rangle\\
\mathbf{Fail}\langle l\!:\!e\rangle &\xrightarrow{\quad} \mathbf{Fail}\langle e\rangle\\
\mathbf{Fail}\langle (pc, l)\!:\!e\rangle &\xrightarrow{\quad} \langle pc,\, l,\, e\rangle\\
\langle pc,\, l_1 : l_2,\, e\rangle &\xrightarrow{\texttt{Open}} \langle pc+1,\, l_1,\, l_2\!:\!e\rangle\\
\langle pc,\, a\!:\!l,\, e\rangle &\xrightarrow{\texttt{Open}} \mathbf{Fail}\langle e\rangle, a \notin List\\
\langle pc,\, \varepsilon,\, e\rangle &\xrightarrow{\texttt{Open}} \mathbf{Fail}\langle e\rangle\\
\langle pc,\, \varepsilon,\, l\!:\!e\rangle &\xrightarrow{\texttt{Close}} \langle pc+1,\, l,\, e\rangle\\
\langle pc,\, l,\, e\rangle &\xrightarrow{\texttt{Close}} \mathbf{Fail}\langle e\rangle, l \neq \varepsilon
\end{aligned}
$$

Fig. 2. Operational semantics of the parsing machine

$$
\begin{aligned}
&\xrightarrow{Open} \langle pc+1,\, l_1,\, l_2 : e\rangle\\
&\xrightarrow{\Pi(G,x+1,p)} \langle pc+|\mathit{\Pi}(G,x+1,p)|+1,\, \varepsilon,\, l_2 : e\rangle\\
&\xrightarrow{Close} \langle pc+|\mathit{\Pi}(G,x+1,p)|+2,\, l_2,\, e\rangle
\end{aligned}
$$

The final state in the sequence above is the same as $\langle pc{+}|\mathit{\Pi}(G,x,\{p\})|,\, l_2,\, e\rangle$, completing the proof for this case.

Case *list*.2 has two subcases, one where the p succeeds and another where p fails. The first subcase is analogous to *list*.1, except the last transition is to a failure state $\mathbf{Fail}\langle e\rangle$. The second subcase takes the machine to a failure state $\mathbf{Fail}\langle l_2\!:\!e\rangle$ by the induction hypothesis, and the second transition rule for failure states then takes the machine to $\mathbf{Fail}\langle e\rangle$.

Case *list*.3 is a straightforward application of the transition rule for `Close`, completing our proof.

We have extended the PEG formalism and the LPEG parsing machine to parse structured data. In the next section we will further extend the parsing machine to make the result of compiling list patterns more efficient.

4 Optimizing List Patterns

The natural way of expressing a choice between two list patterns is $\{p_1\}/\{p_2\}$. In the extended LPEG parsing machine this compiles to the following sequence of instructions:

$$\begin{aligned}
\mathit{\Pi}(g, x, \{p_1\}/\{p_2\}) \equiv\ & \texttt{Choice}\ |\mathit{\Pi}(g, x, p_1)| + 4 \\
& \texttt{Open} \\
& \Pi(g, x + 2, p_1) \\
& \texttt{Close} \\
& \texttt{Commit}\ |\mathit{\Pi}(g, x, p_2)| + 3 \\
& \texttt{Open} \\
& \Pi(g, x + |\mathit{\Pi}(g, x, p_1)| + 5, p_2) \\
& \texttt{Close}
\end{aligned}$$

When p_1 fails, we are going to try to match p_2 against the same subject, but first the machine will unnecessarily backtrack to the parent of this subject first and then try again. We can try to optimize this pattern so a failure in p_1 will backtrack to the state just before trying to match p_1, and then the machine can proceed by trying to match p_2.

A naive way to do this optimization would be to transform $\{p_1\}/\{p_2\}$ to $\{p_1/p_2\}$, but this has a different behavior from the original pattern; if p_1 matches just a prefix of the subject and p_2 matches the whole subject then the new pattern will fail where the original would succeed. But it is easy to correct this error, we just need to add a predicate after p_1, transforming the original pattern to $\{p_1!.\ /\ p_2\}$. Now the left side of the choice will fail on a partial match of p_1, and the new pattern has the same behavior as the original.

A simple extension to the parsing machine helps with this optimization: we can add a new instruction, `NotAny`, that behaves as the pattern !.. The operational semantics of `NotAny` is simple:

$$\begin{aligned}
\langle pc,\, \varepsilon,\, e\rangle &\xrightarrow{\texttt{NotAny}} \langle pc + 1,\, \varepsilon,\, e\rangle \\
\langle pc,\, l,\, e\rangle &\xrightarrow{\texttt{NotAny}} \textbf{Fail}\langle e\rangle
\end{aligned}$$

With the `NotAny` instruction the optimized version of $\{p_1\}/\{p_2\}$ compiles to the following program, which avoids the extra work of the original one:

$$\begin{aligned}
\Pi(g,x,\{p_1\}/\{p_2\}) \equiv\ & \texttt{Open} \\
& \texttt{Choice}\ |\Pi(g,x,p_1)| + 3 \\
& \Pi(g,x+2,p_1) \\
& \texttt{NotAny} \\
& \texttt{Commit}\ |\Pi(g,x,p_2)| + 1 \\
& \Pi(g,x+|\Pi(g,x,p_1)|+4,p_2) \\
& \texttt{Close}
\end{aligned}$$

Proofs of the correctness of the $\{p_1\}/\{p_2\} \equiv \{p_1!.\ /\ p_2\}$ identity and the correctness of compiling !. to `NotAny` are straightforward inductions on the height of the derivation trees for these patterns.

5 Performance

We modified version 0.9 of LPEG, an implementation of the parsing machine as an interpreter written in the C programming language and embedded in the Lua programming language [2], to operate on structured data encoded as Lua arrays, modifying the implementation of existing instructions of the virtual machine and adding new instructions according to Section 3. The modifications are non-trivial but straightforward, and are easy to port to later versions of LPEG. This section presents some benchmarks on this implementation.

Our first benchmark evaluates the effectiveness of the optimization presented in Section 4. The benchmark builds a parsing expression that is an ordered choice of list patterns, where each list pattern matches a list containing a number of atoms (randomly generated). We then try to match subjects against the full expression, where this subject matches the last choice. We varied the number of atoms in each choice from 1 to 10, and the number of choices in each expression from 2 to 10 (for just one choice the optimization is obviously a no-op). We then divided the 100 results obtained without the optimization by the 100 results obtained with the optimization, yielding the relative speedup in each case (1.00x means no speedup, 1.84x means that the optimized pattern was almost twice as fast).

Table 1 shows the results of this benchmark. The increasing speedup as the number of choices increase shows that the optimization is effective. Also, the more work the correct choice has to do in relation to the failing choices (as represented by the number of atoms) the less effective the optimization becomes.

Our second benchmark evaluates the overhead of list matching using our modified LPEG compared to writing a hand-written tree walker in Lua. The grammars in our benchmark are simple enough that the hand-written matchers do not need to backtrack on choices.

	Atoms									
Choices	1	2	3	4	5	6	7	8	9	10
2	1.19x	1.18x	1.18x	1.17x	1.17x	1.17x	1.17x	1.17x	1.17x	1.16x
3	1.25x	1.24x	1.23x	1.23x	1.23x	1.22x	1.22x	1.21x	1.21x	1.21x
4	1.44x	1.43x	1.41x	1.40x	1.40x	1.40x	1.38x	1.38x	1.37x	1.37x
5	1.49x	1.48x	1.46x	1.46x	1.44x	1.44x	1.43x	1.43x	1.42x	1.42x
6	1.55x	1.53x	1.51x	1.50x	1.48x	1.49x	1.47x	1.46x	1.45x	1.46x
7	1.58x	1.56x	1.54x	1.53x	1.52x	1.51x	1.50x	1.50x	1.51x	1.48x
8	1.76x	1.73x	1.70x	1.70x	1.69x	1.68x	1.67x	1.67x	1.66x	1.65x
9	1.79x	1.77x	1.74x	1.74x	1.74x	1.72x	1.71x	1.70x	1.70x	1.69x
10	1.84x	1.82x	1.80x	1.77x	1.78x	1.77x	1.75x	1.76x	1.75x	1.73x

Table 1. Relative speedup of the choice optimization

The first grammar is a simple evaluator of an abstract syntax tree for simple arithmetic expressions (addition, subtraction, multiplication and division). It is the following grammar in LPEG's RE syntax (extended with list patterns):

```
exp <- { "add" exp exp } -> add / { "sub" exp exp } -> sub /
       { "mul" exp exp } -> mul / { "div" exp exp } -> div /
       { "num" <.> }
```

The `<.>` pattern is a *capture* of pattern ., and pushes the captured value in a capture evaluation stack. The `"add" exp exp  -> add` pattern is a *function* capture; it pops the values captured by the pattern on the left side of `->` and calls the function on the right side with these values, pushing the result. Quoted strings are atoms.

The second grammar parses a tree encoding HTML data (in a manner similar to the example in Section 2 and extracts the values of all `src` attributes from `img` tags and `href` attributes from `a` tags. It is the following LPEG grammar:

```
html <- { tag* }
tag  <- {"a" href html} / {"img" src} / {. . html} / .
href <- { {!"href" . .}* {"href" <.>} .* }
src  <- { {!"src" . .}* {"src" <.>} .* }
```

The third grammar is the DTD example of Section 2, used to validate correct trees. The LPEG grammar is a straightforward translation of the grammar in the example.

The three grammars are matched against randomly generated data, where the size of each dataset is set so the running times are in the same order of magnitude. Figure 3 summarizes the results, with the running times in milliseconds.

The results show that the performance of a PEG matcher is competitive with the performance of a hand-written tree matcher written in the same language as the host language for the LPEG implementation. The number of lines of code in the hand-written matchers range from about the same as the grammar, for the evaluator, to several times the size of the grammar for the DTD validator.

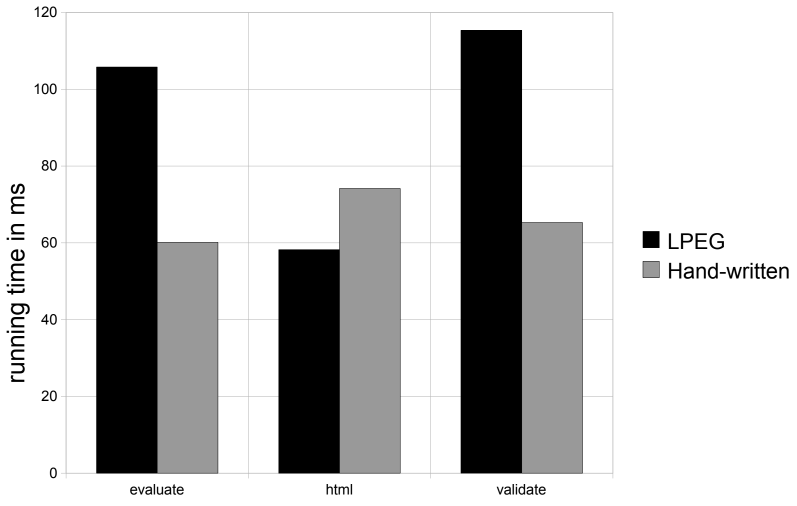

Fig. 3. Comparison of LPEG and hand-written tree matchers

6 Related Work

Sorcerer was a tool that extended EBNF notation with *tree patterns*, where each tree pattern must start with a terminal, its *root* [7]. Sorcerer's grammars must be LL(1); it is implied that for tree patterns this means that the roots of each alternative of a non-terminal must be different, but no formal model for the extension to EBNF and LL(1) grammars is given. It also has semantic and syntactic predicates that can be used to disambiguate alternatives that are not LL(1), but no model for how these additions work is given either. Sorcerer is now part of the AntLR parser generator [8]. Unlike Sorcerer, our extension to PEGs has a formal model, and it does not restrict list patterns.

OMeta is an object-oriented language that uses PEGs as a control mechanism [9], and is embedded in other languages (with a JavaScript embedding, OMeta/JS, being the most mature). OMeta also has pattern matching on structured data, and has a formal model for a fragment of the full language, Core-OMeta [10], that extends PEGs with both list patterns and semantic actions.

Core-OMeta's extensions to PEGs are presented as a single unit, while we focus on just what is necessary for extending PEGs to match structured data. We also extend a parsing machine with new instructions for matching structured data, and give a translation from extended PEGs to programs of this machine.

The OMeta implementations use a traditional combinator-based approach plus memoization for directly executing PEGs. Our extensions could be straightforwardly implemented on an implementation of the parsing machine. On the benchmarks of Figure 3 our implementation is respectively 35, 19, and 36 times faster than OMeta/JS on version 3.2 of V8, a state-of-the-art JavaScript engine [11].

Several programming languages, such as Haskell and the languages in the ML family, include built-in pattern matching facilities to help define functions over structured data [12, 13]. The patterns are limited and do not provide any looping or repetition operators nor the ability to reference other patterns, including self-reference. This makes them strictly less powerful than PEGs.

The Scheme language has an implementation of pattern matching on structured data that, like the pattern matching of Haskell and ML family languages, is also primarily intended for breaking up structured data in its constituent parts [14]. These patterns have a repetition that works like PEGs' repetition operator, but they still cannot reference other patterns, so they also are strictly less powerful than PEGs.

There are several languages for matching and validating XML-structured data that use patterns to describe document schemas, one of them being *regular expression types* [15, 16], an extension to algebraic types that includes repetition. The patterns that describe these types can reference other patterns (including self-reference), but they are restricted to recognizing regular tree languages, a restriction PEGs do not have.

Finally, parser combinators are a popular approach for building recursive-descent parsers in functional programming languages [17]. Parser combinators combine simple parsers using operators such as deterministic or non-deterministic choice, sequencing, chaining, etc. Although originally designed for parsing character streams, parsing combinator libraries can be trivially applied to parsing structured data. But their unrestricted backtracking can have a large time and space cost, with naive implementations being particularly prone to space leaks [18].

7 Conclusions

We presented an extension of Parsing Expression Grammars that allows matching of structured data, instead of just character strings, adding a *list pattern* to standard PEG syntax. We provided an operational semantics for PEGs with this new pattern, and proved that our extensions are conservative.

We also extended the LPEG parsing machine with new instructions for compiling list patterns, and proved the correctness of the compilation of our PEG extensions. We also presented an optimization for list patterns and another instruction for the parsing machine that enables this optimization.

Our extensions were implemented in LPEG 0.9. We benchmarked the effectiveness of the list pattern optimization, showing that the optimization is effective and behaves as expected. We also benchmarked grammars written using our extensions with hand-written parsers for structured data, and showed that the performance of LPEG with list patterns is competitive (from about the same speed to about half as fast) with the performance of hand-written parsers.

LPEG has semantic actions that work efficiently in the presence of PEGs' backtracking in the form of *captures*. Our implementation also extends these semantic actions to work on structured data.

References

1. Ford, B.: Parsing expression grammars: a recognition-based syntactic foundation. In: POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (2004) 111–122
2. Ierusalimschy, R.: A text pattern-matching tool based on Parsing Expression Grammars. Software - Practice and Experience **39**(3) (2009) 221–258
3. Ierusalimschy, R., de Figueiredo, L.H., Filho, W.C.: Lua - an extensible extension language. Software - Practice and Experience **26**(6) (1996) 635–652
4. Ierusalimschy, R.: Programming in Lua, Second Edition. Lua.Org (2006)
5. Medeiros, S., Ierusalimschy, R.: A parsing machine for PEGs. In: DLS '08: Proceedings of the 2008 symposium on Dynamic languages, New York, NY, USA, ACM (2008) 1–12
6. Piumarta, I.: PEG-based transformer provides front-, middle- and back-end stages in a simple compiler. In: Workshop on Self-Sustaining Systems. S3 '10, New York, NY, USA, ACM (2010) 10–20
7. Parr, T.J.: An overview of SORCERER: A simple tree-parser generator. Technical report, University of Minnesota (1994)
8. Parr, T.J.: ANTLR reference manual (2003)
9. Warth, A., Piumarta, I.: OMeta: an object-oriented language for pattern matching. In: DLS '07: Proceedings of the 2007 symposium on Dynamic languages, New York, NY, USA, ACM (2007) 11–19
10. Warth, A.: Experimenting with Programming Languages. PhD thesis, University of California Los Angeles (2009)
11. Google: Design elements of V8 (2011)
12. Peyton Jones, S.L.: The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science). Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1987)
13. Peyton Jones, S.: Haskell 98 Language and Libraries: the Revised Report. Cambridge University Press (2003)
14. Wright, A.K.: Pattern matching for Scheme. Technical report, Department of Computer Science, Rice University (1996)
15. Hosoya, H., Pierce, B.: Regular expression pattern matching for XML. In: POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, New York, NY, USA, ACM (2001) 67–80
16. Hosoya, H., Vouillon, J., Pierce, B.C.: Regular expression types for XML. ACM Transactions on Programming Languages and Systems (TOPLAS) **27**(1) (2005) 46–90
17. Hutton, G., Meijer, E.: Monadic Parsing in Haskell. Journal of Functional Programming **8**(4) (July 1998) 437–444
18. Leijen, D.J.P., Meijer, H.J.M.: 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)