Efficient List Matching using PEGs

Sérgio Medeiros[1], **Fabio Mascarenhas**[1], **Roberto Ierusalimschy**[1]

[1]Department of Computer Science – PUC-Rio – Rio de Janeiro – Brazil

{smedeiros,roberto}@inf.puc-rio.br, mascarenhas@acm.org

Abstract. *Parsing Expression Grammars (PEGs) are formalism for language recognition that renewed interest in top-down parsing approaches. We extend the PEG formalism with an operational semantics for pattern matching on list-structured data. We also present a virtual parsing machine that can efficiently execute PEGs compiled to machine instructions, and a proof of the soundness of the machine's soundness. Finally, we also introduce a formalization of captures and semantic actions for PEGs, and present benchmarks of our implementatin of the parsing machine.*

1. Introduction

Parsing Expression Grammars (PEGs) [Ford 2004] are a formalism for language recognition which renewed interest in top-down parsing approaches. The PEG formalism gives a convenient syntax for describing top-down parsers for unambiguous languages. The core of the formalism is a form of limited backtracking via *ordered choice*. The limited backtracking gives PEGs the property of composability, meaning that a PEG can be used by another PEG just by making sure the names of their productions do not clash. This also lets PEG implementations integrate cleanly with hand-written parsers. These properties make PEGs very attractive for building dynamically-extensible parsers.

LPEG [Ierusalimschy 2009] is a pattern-matching tool for the Lua language [Ierusalimschy 2006] that 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 [Medeiros and Ierusalimschy 2008]. LPEG builds these programs at runtime, dynamically composing smaller programs into bigger programs, thus taking advantage of the composability of PEGs. PEGs limited backtracking is also reflected on this parsing machine through its use of a stack to manage backtrack information.

This paper is a continuation of a previous work [Medeiros and Ierusalimschy 2008] where we presented an alternative operational semantic for PEGs (based on natural semantics), and a formal specification of the parsing machine, along with a correctness proof for our transformation of PEGs to programs of the machine.

Both the original PEG formalism and our previous parsing machine assume that the subject a PEG recognizes is a string of characters. In this paper, we extend both formalisms to parse structured data in the form of Lisp-style lists (a possibly empty list where each element can be an atom or another list). We show that our previous results hold when matching lists or slices of lists that are isomorphic to strings, and give proofs for the correctness of the translation of our new constructions.

Extending PEGs to match structured data make PEGs useful for a larger part of the compilation pipeline. 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, then either compiling it to a target language (either an intermediate language close to machine code or a naive translation to machine code). Using the same abstraction (PEGs) for the whole pipeline of simple domain-specific languages can make them easier for programmers to design and implement.

Adding a few extra instructions to the parsing machine enables transformation of certain pattern classes to more efficient programs, and we also present these optimizations. We restrict ourselves to optimizations on list patterns, as our previous work has covered more general parsing machine optimizations. For each optimization, we present a proof of its correctness and a benchmark showing how effective it is.

We also present an extension to the PEG formalism to include a mix of standard pattern matchers' captures and parser generators' semantic actions, by which we can extract useful information from structured data, instead of just knowing if it fits a pattern. We do this lazily, with the match just collecting enough information to extract captures and execute actions later (in a typical implementation, right after the match finishes). This means that semantic actions are free to have side-effects even in the presence of backtracking.

Finally, we review related work on parsing structured data and on semantic actions for PEGs. We also present another set of benchmarks comparing our extended LPEG with a PEG-based tool of similar power, showing an order-of-magnitude improvement.

The rest of this paper is organized as follows: Section 2 extends PEGs with list patterns; Section 3 describes the virtual parsing machine; proves the transformation between PEGs and their corresponding programs for the machine is correct; Section 4 describes optimizations for list patterns in our machine; Section 5 extends PEGs with lazy semantic actions and shows how to execute them; Section 7 reviews some related work; finally, Section 8 summarizes our results.

2. Extending PEGs for Lists

In [Medeiros and Ierusalimschy 2008], we described a `match` relation for PEGs, where we defined the relation `match` on $\text{Grammar} \times \text{Pattern} \times \Sigma^* \times \mathcal{N} \times (\mathcal{N} \cup \{\texttt{fail}\})$. Given a grammar (a map from variables to patterns), a pattern, a subject string, and a position (the subject starts at position 1), the relation gives us either the position after the match (possibly the length of the subject plus 1) or `fail`, depending on whether the pattern matches the subject or not. We use $\text{match}\ G\ p\ s\ i \rightsquigarrow j$ to indicate that $(G, p, s, i, j) \in \texttt{match}$.

In this paper, we are considering lists to be Lisp-style lists, made up from atoms and other lists. We will restrict ourselves to only characters as atoms. We represent the empty list as $\{\}$. Lists are inductively defined by the operator $:$ (also known as cons). If c is an atom and l is a list then $c : l$ is a list, and if l_1 and l_2 are lists then $l_1 : l_2$ is a list. The first argument of : is the head of the list and the second is the tail. We will use $\{e_1 e_2 \ldots e_n\}$ as another way to write $e_1 : e_2 : \ldots : e_n : \{\}$.

We will use $|l|$ to represent the number of elements of list l, and its inductive

'.' $\in$ Pattern
'c' $\in$ Pattern
If $(A, p) \in$ Grammar then $A \in$ Pattern
If $p \in$ Pattern then p^*, $!p$, $\&p$, and $\{p\} \in$ Pattern
If p_1 and $p_2 \in$ Pattern then $p_1 p_2$ and $p_1 / p_2 \in$ Pattern

Table 1. Syntax of PEG patterns

definition is trivial. We will use $l[i]$ to denote the i^{th} element of the list, starting from 1, which also has a straightforward inductive definition. List concatenation is represented by juxtaposition ($l_1 l_2$ is the list with all elements of l_1 followed by all elements from l_2, not to be confused with $l_1 : l_2$, which is a list with l_1 as the first element followed by the elements of l_2).

Now we can define a new relation $\texttt{match}_L$ on Grammar $\times$ Pattern $\times$ List $\times$ $\mathcal{N} \times (\mathcal{N} \cup \{\texttt{fail}\})$. This relation gives us the position in the list after trying to match a pattern given a starting position and a grammar, or `fail`if the list does not match the pattern. As with `match`, we also have a notation $\text{match}_L\ G\ p\ s\ i \rightsquigarrow j$ to indicate that $(G, p, s, i, j) \in \texttt{match}_L$.

Table 1 presents the abstract syntax of PEG patterns with the new *list pattern* $\{p\}$, and Figure 1 presents an operational semantics of $\texttt{match}_L$ given as rules on the pattern structure. Except for rules $list.1$ and $list.2$, all semantic rules of $\texttt{match}_L$ are taken straight from `match`. We have that $\text{match}_L\ G\ p\ s\ i \rightsquigarrow j$ if we can use the rules to build a finite derivation tree for this proposition.

Rules $list.1$ and $list.2$ formalize the notion that a list pattern $\{p\}$ only succeeds if the current element of the subject is also a list and if p matches this whole sublist from the first element. Thus the pattern $\{$'a''b'$\}$ matches the list $\{$'a''b'$\}$ but not the list $\{$'a''b''c'$\}$, or the atom 'a'.

The set of character strings is isomorphic to the set of lists where all elements are characters plus the empty list. Lets define $map_{s \to l}$ as the natural mapping from strings to lists, that is, the empty string maps to the empty list, and a string 'c_1' ... 'c_n' maps to $\{$'c_1' ... 'c_n'$\}$. In other words, if s is a string, $|s| = |map_{s \to l}(s)|$ and $s[i] = map_{s \to l}(s)[i]$ for $1 \leq i \leq |s|$. We then have

$$\text{match}\ G\ p\ s\ i \rightsquigarrow j \ \text{ iff } \ \text{match}_L\ G\ p\ map_{s \to l}(s)\ i \rightsquigarrow j.$$

with a trivial inductive proof on the height of the derivation tree, given the semantic rules are identical. In other words, our extended semantics is identical to the regular PEG semantics when we are dealing with string-like lists as subjects.

3. Parsing Machine

In [Medeiros and Ierusalimschy 2008] we also presented the semantics for a parsing machine for PEGs, along with a transformation from PEG patterns to programs that this machine executes. The parsing machine is the core of LPEG, our implementation of PEGs, and is emphasizes implementation efficiency without sacrificing the expressive and parsing power of PEGs.

Character $\dfrac{s[i] = \text{`}c\text{'}}{\text{match}_L\ g\ \text{`}c\text{'}\ s\ i \rightsquigarrow \text{i+1}}$ **(ch.1)** $\dfrac{s[i] \neq \text{`}c\text{'}}{\text{match}_L\ g\ \text{`}c\text{'}\ s\ i \rightsquigarrow \texttt{fail}}$ **(ch.2)**

Any Element $\dfrac{i \leq |s|}{\text{match}_L\ g\ .\ s\ i \rightsquigarrow \text{i+1}}$ **(any.1)** $\dfrac{i > |s|}{\text{match}_L\ g\ .\ s\ i \rightsquigarrow \texttt{fail}}$ **(any.2)**

Not Predicate $\dfrac{\text{match}_L\ g\ p\ s\ i \rightsquigarrow \texttt{fail}}{\text{match}_L\ g\ !p\ s\ i \rightsquigarrow \text{i}}$ **(not.1)** $\dfrac{\text{match}_L\ g\ p\ s\ i \rightsquigarrow \text{i+j}}{\text{match}_L\ g\ !p\ s\ i \rightsquigarrow \texttt{fail}}$ **(not.2)**

And Predicate $\dfrac{\text{match}_L\ g\ p\ s\ i \rightsquigarrow \text{i+j}}{\text{match}_L\ g\ \&p\ s\ i \rightsquigarrow \text{i}}$ **(and.1)** $\dfrac{\text{match}_L\ g\ p\ s\ i \rightsquigarrow \texttt{fail}}{\text{match}_L\ g\ \&p\ s\ i \rightsquigarrow \texttt{fail}}$ **(and.2)**

Concatenation $\dfrac{\text{match}_L\ g\ p_1\ s\ i \rightsquigarrow \text{i+j} \quad \text{match}_L\ g\ p_2\ s\ i+j \rightsquigarrow \text{i+j+k}}{\text{match}_L\ g\ p_1p_2\ s\ i \rightsquigarrow \text{i+j+k}}$ **(con.1)**

$\dfrac{\text{match}_L\ g\ p_1\ s\ i \rightsquigarrow \text{i+j} \quad \text{match}_L\ g\ p_2\ s\ i+j \rightsquigarrow \texttt{fail}}{\text{match}_L\ g\ p_1p_2\ s\ i \rightsquigarrow \texttt{fail}}$ **(con.2)** $\dfrac{\text{match}_L\ g\ p_1\ s\ i \rightsquigarrow \texttt{fail}}{\text{match}_L\ g\ p_1p_2\ s\ i \rightsquigarrow \texttt{fail}}$ **(con.3)**

Ordered Choice $\dfrac{\text{match}_L\ g\ p_1\ s\ i \rightsquigarrow \texttt{fail} \quad \text{match}_L\ g\ p_2\ s\ i \rightsquigarrow \texttt{fail}}{\text{match}_L\ g\ p_1/p_2\ s\ i \rightsquigarrow \texttt{fail}}$ **(ord.1)**

$\dfrac{\text{match}_L\ g\ p_1\ s\ i \rightsquigarrow \text{i+j}}{\text{match}_L\ g\ p_1/p_2\ s\ i \rightsquigarrow \text{i+j}}$ **(ord.2)** $\dfrac{\text{match}_L\ g\ p_1\ s\ i \rightsquigarrow \texttt{fail} \quad \text{match}_L\ g\ p_2\ s\ i \rightsquigarrow \text{i+k}}{\text{match}_L\ g\ p_1/p_2\ s\ i \rightsquigarrow \text{i+k}}$ **(ord.3)**

Repetition $\dfrac{\text{match}_L\ g\ p\ s\ i \rightsquigarrow \text{i+j} \quad \text{match}_L\ g\ p^*\ s\ i+j \rightsquigarrow \text{i+j+k}}{\text{match}_L\ g\ p^*\ s\ i \rightsquigarrow \text{i+j+k}}$ **(rep.1)** $\dfrac{\text{match}_L\ g\ p\ s\ i \rightsquigarrow \texttt{fail}}{\text{match}_L\ g\ p^*\ s\ i \rightsquigarrow \text{i}}$ **(rep.2)**

Variables $\dfrac{\text{match}_L\ g\ g(A_k)\ s\ i \rightsquigarrow \text{i+j}}{\text{match}_L\ g\ A_k\ s\ i \rightsquigarrow \text{i+j}}$ **(var.1)** $\dfrac{\text{match}_L\ g\ g(A_k)\ s\ i \rightsquigarrow \texttt{fail}}{\text{match}_L\ g\ A_k\ s\ i \rightsquigarrow \texttt{fail}}$ **(var.2)**

Closed Grammars $\dfrac{\text{match}_L\ g\ g(A_k)\ s\ i \rightsquigarrow \text{i+j}}{\text{match}_L\ g'\ (g, A_k)\ s\ i \rightsquigarrow \text{i+j}}$ **(cg.1)** $\dfrac{\text{match}_L\ g\ g(A_k)\ s\ i \rightsquigarrow \texttt{fail}}{\text{match}_L\ g'\ (g, A_k)\ s\ i \rightsquigarrow \texttt{fail}}$ **(cg.2)**

List $\dfrac{\text{match}_L\ g\ p\ s[i]\ 1 \rightsquigarrow |s[i]|\text{+}\ 1}{\text{match}_L\ g\ \{p\}\ s\ i \rightsquigarrow \text{i+1}}$ **(list.1)** $\dfrac{\text{match}_L\ g\ p\ s[i]\ 1 \neq |s[i]|\ \text{+}\ 1}{\text{match}_L\ g\ \{p\}\ s\ i \rightsquigarrow \texttt{fail}}$ **(list.2)**

Figure 1. Operational semantics of PEGs with list patterns

The machine has a program counter to address the next instruction to execute, a register to hold the current position in the subject, and a stack that the machine uses for pushing call 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 position in the subject. The machine's instructions manipulate on the program counter, position register and the stack.

To be able to translate list patterns to the parsing machine, we need to add another register to hold the current subject. We also need to be able to store *list frames* in the stack, a list frame is a subject and a position. These two additions let we save the current subject and position in the stack before starting to process a sub-list that is an element of the current subject. We also need to add two new instructions:

`Open` pushes a list frame on the stack, then changes the subject to the element at the current position. The current position becomes 1, the start of the new subject. Fails if the element at the current position is not a list.

`Close` if the current position points past the end of the subject, pops the top entry from the stack (which will be a list frame), setting the subject to the subject in the popped frame and the position to the position in the frame plus one. Fails if the current position is not pointing past the end of the subject.

Formally, the program counter, the subject and position registers, and the stack

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

Figure 2 presents the operational semantics of the 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.

$$
\begin{array}{rcll}
\langle pc,\, s,\, i,\, e\rangle & \xrightarrow{\texttt{Char}\ x} & \langle pc+1,\, s,\, i+1,\, e\rangle & s[i] = x \\
\langle pc,\, s,\, i,\, e\rangle & \xrightarrow{\texttt{Char}\ x} & \mathbf{Fail}\langle e\rangle & s[i] \neq x \\
\langle pc,\, s,\, i,\, e\rangle & \xrightarrow{\texttt{Any}} & \langle pc+1,\, s,\, i+1,\, e\rangle & i+1 \leq |s| \\
\langle pc,\, s,\, i,\, e\rangle & \xrightarrow{\texttt{Any}} & \mathbf{Fail}\langle e\rangle & i+1 > |s| \\
\langle pc,\, s,\, i,\, e\rangle & \xrightarrow{\texttt{Choice}\ l} & \langle pc+1,\, s,\, i,\, (pc+l, s, i) : e\rangle & \\
\langle pc,\, s,\, i,\, e\rangle & \xrightarrow{\texttt{Jump}\ l} & \langle pc+l,\, s,\, i,\, e\rangle & \\
\langle pc,\, s,\, i,\, e\rangle & \xrightarrow{\texttt{Call}\ l} & \langle pc+l,\, s,\, i,\, (pc+1) : e\rangle & \\
\langle pc,\, s,\, i,\, pc_1 : e\rangle & \xrightarrow{\texttt{Return}} & \langle pc_1,\, s,\, i,\, e\rangle & \\
\langle pc,\, s,\, i,\, h : e\rangle & \xrightarrow{\texttt{Commit}\ l} & \langle pc+l,\, s,\, i,\, e\rangle & \\
\langle pc,\, s,\, i,\, e\rangle & \xrightarrow{\texttt{Fail}} & \mathbf{Fail}\langle e\rangle & \\
\mathbf{Fail}\langle pc : e\rangle & \xrightarrow{\textit{any}} & \mathbf{Fail}\langle e\rangle & \\
\mathbf{Fail}\langle (s, i) : e\rangle & \xrightarrow{\textit{any}} & \mathbf{Fail}\langle e\rangle & \\
\mathbf{Fail}\langle (pc, s, i) : e\rangle & \xrightarrow{\textit{any}} & \langle pc,\, s,\, i,\, e\rangle & \\
\langle pc,\, s,\, i,\, e\rangle & \xrightarrow{\texttt{Open}} & \langle pc+1,\, s[i],\, 1,\, (s, i) : e\rangle & s[i] \in \mathit{List} \\
\langle pc,\, s,\, i,\, e\rangle & \xrightarrow{\texttt{Open}} & \mathbf{Fail}\langle e\rangle & s[i] \notin \mathit{List} \\
\langle pc,\, s,\, i,\, (s_1, i_1) : e\rangle & \xrightarrow{\texttt{Close}} & \langle pc+1,\, s_1,\, i_1+1,\, e\rangle & i = |s|+1 \\
\langle pc,\, s,\, i,\, (s_1, i_1) : e\rangle & \xrightarrow{\texttt{Close}} & \mathbf{Fail}\langle e\rangle & i \neq |s|+1
\end{array}
$$

Figure 2. Operational semantics of the parsing machine

The process of translating a pattern into a program is bottom up, and we do it at runtime. The simplest patterns translate to simple programs, and then we combine programs according to the rules of each PEG operation. Programs are opaque entities for the translation process: the pattern that originated them is not important, as long as the programs are valid. The process is fully incremental, and combining programs is a simple matter of concatenating their texts.

In [Medeiros and Ierusalimschy 2008], we represent the compilation process using a transformation function Π on the domain $\mathrm{Grammar} \times \mathcal{N} \times \mathrm{Pattern}$, where $\Pi(g, i, p)$ is the translation of pattern p in the context of grammar g, with i an index used just for compiling references to other rules in the grammar (the mechanism is fully explained in our previous work). We will also use the notation $|\Pi(g, i, p)|$, which means the number of instructions of the program $\Pi(g, i, p)$.

The proof of correctness of this transformation is an induction on derivation trees. We keep all the transformation rules of the previous work, and it's straightforward to

check the previous proof remains valid. We just have to extend the proof for our two new induction steps related to the $list.1$ and $list.2$ rules of $\texttt{match}_L$.

Given a PEG grammar g, a position i relative to the beginning of the grammar, and a pattern $\{p\}$, we have

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

as its transformation to virtual machine instructions.

Now lets do our two induction steps. First we are going to prove

$$\begin{aligned}&\texttt{If match}_L\ g\ \{p\}\ s\ i = \texttt{i+1 then} \\ &\langle pc,\ s,\ i,\ e\rangle \xrightarrow{\Pi(g,i,\{p\})} \langle pc + |\Pi(g, i, p)| + 2,\ s,\ i+1,\ e\rangle\end{aligned}$$

using the semantic rule $list.1$, where p matches all of the contents of element i in the current subject. We have

$$\begin{aligned}&\xrightarrow{Open} \langle pc + 1,\ s[i],\ 1,\ (s, i) : e\rangle \\ &\xrightarrow{\Pi(g,i+1,p)} \langle pc + |\Pi(g, i, p)| + 1,\ s[i],\ |s[i]| + 1,\ (s, i) : e\rangle \\ &\xrightarrow{Close} \langle pc + |\Pi(g, i, p)| + 2,\ s,\ i+1,\ e\rangle\end{aligned}$$

as the sequence of transitions the machine executes. After the `Open` instruction, the subject becomes $s[i]$, and the current position is 1. By induction we know the execution of $\Pi(g, i+1, p)$ leads to $\langle pc + |\Pi(g, i, p)| + 1,\ s[i],\ |s[i]| + 1,\ (s, i) : e\rangle$, and after the `Close` instruction we reach the final state $\langle pc + |\Pi(g, i, p)| + 2,\ s,\ i+1,\ e\rangle$.

Now we need to prove

$$\begin{aligned}&\texttt{If match}_L\ g\ \{p\}\ s\ i = \texttt{fail then} \\ &\langle pc,\ s,\ i,\ e\rangle \xrightarrow{\Pi(g,i,\{p\})} \textbf{Fail}\langle e\rangle\end{aligned}$$

using semantic rule $list.2$, where p does not match the contents of element i, or matches just part of them. Considering $n < |s[i]| + 1$, if p matches the first n elements of element i then we have

$$\begin{aligned}&\xrightarrow{Open} \langle pc + 1,\ s[i],\ 1,\ (s, i) : e\rangle \\ &\xrightarrow{\Pi(g,i+1,p)} \langle pc + |\Pi(g, i, p)| + 1,\ s[i],\ n,\ (s, i) : e\rangle \\ &\xrightarrow{Close} \textbf{Fail}\langle e\rangle\end{aligned}$$

as the sequence of transitions. As $n < |s[i]| + 1$ the execution of the `Close` instruction leads the machine to a failure state. There is also the case where p fails, with

$$\begin{aligned}&\xrightarrow{Open} \langle pc + 1,\ s[i],\ 1,\ (s, i) : e\rangle \\ &\xrightarrow{\Pi(g,i+1,p)} \langle pc + |\Pi(g, i, p)| + 1,\ s[i],\ n,\ (s, i) : e\rangle \\ &\xrightarrow{Close} \textbf{Fail}\langle (s, i) : e\rangle \\ &\rightarrow \textbf{Fail}\langle e\rangle\end{aligned}$$

being the transitions, and completing our proof.

We have now extended both PEGs and our parsing machine to do matching on list-structured data. In the next section we will review a couple ways of making this more efficient.

4. Optimizations

The previous section showed how to compile list patterns to our parsing machine, and proved the correctness of this transformation. In several cases, though, we can make the resulting program more efficient, avoiding unnecessary pushes and pops on the machine's stack. This section shows another way of compiling some list patterns by using pattern identities and a few extra machine instructions, generating more efficient programs.

Figure 3 lists the instructions we are adding to the machine and their semantics. Let's start with a very common pattern, $\{\text{`}c_1\text{'} \ldots \text{`}c_n\text{'}\}$, that is, a pattern that matches a list of n characters. The current program for this pattern is an `Open` instruction followed by a sequence of `Char` instructions and a `Close` instruction. We can compile this pattern to a single `String` instruction with the string "$c_1 \ldots c_2$" as its argument.

$$
\begin{array}{llll}
\langle pc,\, s,\, i,\, e\rangle & \xrightarrow{\texttt{String}\ x} & \langle pc+1,\, s,\, i+1,\, e\rangle & s[i] = map_{s\to l}(x) \\
\langle pc,\, s,\, i,\, e\rangle & \xrightarrow{\texttt{String}\ x} & \textbf{Fail}\langle e\rangle & s[i] \neq map_{s\to l}(x) \\
\langle pc,\, s,\, i,\, e\rangle & \xrightarrow{\texttt{TestString}\ l\ x} & \langle pc+1,\, s,\, i+1,\, e\rangle & s[i] = map_{s\to l}(x) \\
\langle pc,\, s,\, i,\, e\rangle & \xrightarrow{\texttt{TestString}\ l\ x} & \langle pc+l,\, s,\, i,\, e\rangle & s[i] \neq map_{s\to l}(x) \\
\langle pc,\, s,\, i,\, e\rangle & \xrightarrow{\texttt{NotAny}} & \langle pc+1,\, s,\, i,\, e\rangle & i > |s| \\
\langle pc,\, s,\, i,\, e\rangle & \xrightarrow{\texttt{NotAny}} & \textbf{Fail}\langle e\rangle & i \leq |s|
\end{array}
$$

Figure 3. Operational semantics of extra instructions for optimization

Now that we have a single instruction that matches a whole list of characters (a list that is isomorphic to a string), we can do *head-fail* optimizations on these tests by having a complementary `TestString` instruction that jumps to a label instead of failing if it does not match. This kind of failure is very common when we have a pattern like $\{\text{`}c_1\text{'} \ldots \text{`}c_n\text{'}\}p_1/p_2$, such as when matching against a tree that uses strings in the first element as tags. We can avoid pushing a backtrack entry that will be discarded most of the time when trying to match the tag. The compilation for the pattern above becomes

$$
\begin{aligned}
\Pi(g, x, \{c_1 \ldots c_n\}p_1/p_2) \equiv\ & \texttt{TestString}\ |\Pi(g, x, p_1)| + 3\ \ \text{`}c_1\text{'} \ldots \text{`}c_n\text{'} \\
& \texttt{Choice}\ |\Pi(g, x, p_1)| + 2\ \ \ 1 \\
& \Pi(g, x+2, p_1) \\
& \texttt{Commit}\ |\Pi(g, x, p_2)| + 1 \\
& \Pi(g, x + |\Pi(g, x, p_1)| + 3, p_2)
\end{aligned}
$$

with the `Choice` instruction now taking an offset, which we subtract from the current position when pushing a new backtrack entry. An offset of zero is assumed if an offset is not provided.

In case of an unsuccessful match of $\{\text{`}c_1\text{'} \ldots \text{`}c_n\text{'}\}$ and a successful match of p_2, we have

$$\xrightarrow{TestString} \langle pc + |\Pi(g, x, p_1)| + 3,\ s,\ i,\ e\rangle$$
$$\xrightarrow{\Pi(g,x+|\Pi(g,x,p_1)|+3,p_2)} \langle pc + |\Pi(g, x, \{\text{`}c_1\text{'} \ldots \text{`}c_n\text{'}\}p_1/p_2)|,\ s,\ i+k,\ e\rangle$$

as the sequence of transitions, which is correct by induction on the correctness of p_2's compilation.

Finally, there is an interesting optimization derived from an identity between patterns. The natural way of writing a choice between two list patterns is $\{p_1\}/\{p_2\}$. This compiles to a `Choice` followed by an `Open` instruction, followed by the compilation of p_1 and a `Close`, then a `Commit`, another `Open`, the compilation of p_2, and a final `Close` instruction. But when p_1 fails we end up trying to match p_2 against the same subject, and we have an unnecessary stack pop and push (plus unnecessary changes of the current subject, which can have implementation costs). The naive optimization is to transform this pattern to $\{p_1/p_2\}$, but now if p_1 is a partial match and p_2 is a full match the new pattern will fail where the previous one would succeed. We can correct this by transforming the original pattern to $\{p_1!./p_2\}$.

Now a partial match of p_1 will fail the antecedent of the choice, and if p_2 is a full match then the whole pattern succeeds, as in the original. By using the new `NotAny` instruction to compile !., the new pattern compiles to

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

avoiding the extra work of the original pattern, and incidentally opening the patterns to further head-fail optimization. Full proofs of the identity, and the correctness of compiling !. to `NotAny`, are straightforward inductions on derivation trees.

5. Captures

Recognizing whether a subject fits a pattern is seldom what we just want from a pattern matcher; usually we also want to extract parts of the subject that fit parts of the pattern, or do some computation on these parts. If our subject is an abstract syntax tree for a programming language, for example, we may want to do a pattern-directed evaluation of this tree, or generate a new tree by transforming the old (converting to an intermediate representation closer to machine code, doing optimizations, or naively compiling to machine code, for example). Pattern matchers usually offer a way of extracting parts of the subject via *captures*, while parser generators have *semantic actions* executed when parsing each rule of a grammar. In this section we are going to combine both concepts and add them

If $v \in$ Value then $\langle const,\ v\rangle \in$ Pattern
If $p \in$ Pattern then $\langle simp,\ p\rangle \in$ Pattern
If $p \in$ Pattern and $f \in$ List $\rightarrow$ List then $\langle func,\ f\rangle \in$ Pattern
If $p \in$ Pattern and $f \in$ Value $\times$ Value $\rightarrow$ Value then $\langle fold,\ f\rangle \in$ Pattern

Table 2. Syntax of capture patterns

to our formalization of PEGs. The rest for the section refers to both captures and actions as just *captures*.

We will define four kinds of captures: *simple*, which just captures a slice of the current subject; *constant*, used to inject external values so other captures can use them; *function*, a semantic action proper, which applies a given function to a list of captured values and passes the result to other captures; and *fold*, which folds (or reduces) a list of captured values using a supplied binary function, passing the result to other captures.

In the previous paragraph we say that captures can pass values to other captures, this is due to the nesting property of PEG patterns. A capture is a pattern, too, with the syntax in Table 2, and, except for constant captures, all captures have nested patterns (Value $\equiv$ List $\cup$ Atom). Simple captures take the slice of the subject that their nested capture matches. Function and fold captures consume the values produced by their nested captures. An important characteristic of our specification of captures is that no values are extracted and no functions applied when trying to match patterns; matching capture patterns just pushes information about these captures in a *list of captures*, such as the current subject and position, the function that we have to apply, etc. To get the values and execute the semantic actions we need to evaluate this list after a successful match. This means that the functions used in function and fold captures can have side-effects in languages that have them without causing problems with the backtracking. Furthermore, we will specify captures in our parsing machine in a way that if p matches and yields a list of captures c then compiling p and executing it on the parsing machine will yield the same list of captures, so we will have a single evaluation function for both patterns and programs.

Formally, we have to extend our $\texttt{match}_L$ relation, which now becomes: $\text{Grammar} \times \text{Pattern} \times \text{List} \times \mathcal{N} \times ((\mathcal{N}, Capture^*) \cup \{\texttt{fail}\})$, where $Capture^*$ is a list of captures. Naturally, the notation $\text{match}_L\ G\ p\ s\ i \rightsquigarrow (j, c)$ means $(G, p, s, i, (j, c)) \in \texttt{match}_L$. Figure 4 presents the operational semantics of $\texttt{match}_L$ with captures. We elided the cases where the new rule is just the old one with a list of captures added to the result of the match, as well as the cases where a pattern fails and the list of captures is discarded.

We also have to extend the definition of our parsing machine (Figure 5, adding the list of captures to its non-failure states (k in the figure). Backtrack frames also need to keep the current list of captures along with subject and position. We also add five new instructions to the machine, to add the correct capture tuples in the list of captures.

Compiling capture patterns is straightforward. Pattern $\langle const,\ v\rangle$ compiles to just

Character $\dfrac{s[i] = \text{`}c\text{'}}{\text{match}_L\ g\ \text{`}c\text{'}\ s\ i \rightsquigarrow (\text{i+1},\ \{\})}$ **(ch.1)** **Any Element** $\dfrac{i \leq |s|}{\text{match}_L\ g\ .\ s\ i \rightsquigarrow (\text{i+1},\ \{\})}$ **(any.1)**

Not Predicate $\dfrac{\text{match}_L\ g\ p\ s\ i \rightsquigarrow \texttt{fail}}{\text{match}_L\ g\ !p\ s\ i \rightsquigarrow (\text{i},\ \{\})}$ **(not.1)** **Repetition** $\dfrac{\text{match}_L\ g\ p\ s\ i \rightsquigarrow \texttt{fail}}{\text{match}_L\ g\ p^*\ s\ i \rightsquigarrow (\text{i},\ \{\})}$ **(rep.2)**

$$\frac{\text{match}_L\ g\ p\ s\ i \rightsquigarrow (\text{i+j},\ c) \quad \text{match}_L\ g\ p^*\ s\ i+j \rightsquigarrow (\text{i+j+k},\ c_1)}{\text{match}_L\ g\ p^*\ s\ i \rightsquigarrow (\text{i+j+k},\ c\,c_1)}\ \textbf{(rep.1)}$$

Concatenation $\dfrac{\text{match}_L\ g\ p_1\ s\ i \rightsquigarrow (\text{i+j},\ c_1) \quad \text{match}_L\ g\ p_2\ s\ i+j \rightsquigarrow (\text{i+j+k},\ c_2)}{\text{match}_L\ g\ p_1 p_2\ s\ i \rightsquigarrow (\text{i+j+k},\ c_1\,c_2)}$ **(con.1)**

Const Capture $\dfrac{}{\text{match}_L\ g\ \langle const,\ v\rangle\ s\ i \rightsquigarrow (\text{i},\ \{\langle const,\ v\rangle\})}$ **(const.1)**

Simple Capture $\dfrac{\text{match}_L\ g\ p\ s\ i \rightsquigarrow (\text{i+j},\ c)}{\text{match}_L\ g\ \langle simp,\ p\rangle\ s\ i \rightsquigarrow (\text{i+j},\ (\langle simp,\ s,\ i\rangle) : c\,\{\langle close,\ i+j\rangle\})}$ **(simple.1)**

Function Capture $\dfrac{\text{match}_L\ g\ p\ s\ i \rightsquigarrow (\text{i+j},\ c)}{\text{match}_L\ g\ \langle func,\ f\rangle\ s\ i \rightsquigarrow (\text{i+j},\ \langle func,\ f\rangle\,c\,\{\langle close,\ i+j\rangle\})}$ **(func.1)**

Fold Capture $\dfrac{\text{match}_L\ g\ p\ s\ i \rightsquigarrow (\text{i+j},\ c)}{\text{match}_L\ g\ \langle fold,\ f\rangle\ s\ i \rightsquigarrow (\text{i+j},\ \langle fold,\ f\rangle : c\,\{\langle close,\ i+j\rangle\})}$ **(fold.1)**

Figure 4. Operational semantics of PEGs with capture patterns

$$\begin{array}{rcl}
\langle pc,\ s,\ i,\ e,\ k\rangle & \xrightarrow{\texttt{Choice}\ l\ o} & \langle pc+1,\ s,\ i,\ (pc+l, s, i-o, k) : e,\ k\rangle \\
\textbf{Fail}\langle (pc, s, i, k) : e\rangle & \xrightarrow{any} & \langle pc,\ s,\ i,\ e,\ k\rangle \\
\langle pc,\ s,\ i,\ e,\ k\rangle & \xrightarrow{\texttt{ConstCap}\ v} & \langle pc+1,\ s,\ i,\ e,\ k\,\{\langle const,\ v\rangle\}\rangle \\
\langle pc,\ s,\ i,\ e,\ k\rangle & \xrightarrow{\texttt{SimpCap}} & \langle pc+1,\ s,\ i,\ e,\ k\,\{\langle simp,\ s,\ i\rangle\}\rangle \\
\langle pc,\ s,\ i,\ e,\ k\rangle & \xrightarrow{\texttt{FuncCap}\ v} & \langle pc+1,\ s,\ i,\ e,\ k\,\{\langle func,\ f\rangle\}\rangle \\
\langle pc,\ s,\ i,\ e,\ k\rangle & \xrightarrow{\texttt{FoldCap}\ v} & \langle pc+1,\ s,\ i,\ e,\ k\,\{\langle fold,\ f\rangle\}\rangle \\
\langle pc,\ s,\ i,\ e,\ k\rangle & \xrightarrow{\texttt{CloseCap}} & \langle pc+1,\ s,\ i,\ e,\ k\,\{\langle close,\ i\rangle\}\rangle
\end{array}$$

Figure 5. Operational semantics of instructions for captures

`ConstCap` v. Pattern $\langle simp,\ p\rangle$ compiles to

$$\begin{aligned}\Pi(g, x, \langle simp,\ p\rangle) \equiv\ & \texttt{SimpCap} \\ & \Pi(g, x+1, p) \\ & \texttt{CloseCap}\end{aligned}$$

with analogous compilation for function and fold captures, replacing `SimpCap` with `FuncCap` f or `FoldCap` f, respectively.

Proving that if $\text{match}_L\ G\ p\ s\ i \rightsquigarrow (i+j, c)$ then $\langle pc,\ s,\ i,\ e,\ k\rangle \xrightarrow{\Pi(g,x,p)} \langle pc + |\Pi(g, x, p)|,\ s,\ i+j,\ e,\ kc\rangle$ is also an induction on derivation trees. We will show the case where $\text{match}_L\ G\ \langle simp,\ p\rangle\ s\ i \rightsquigarrow (\text{i+j},\ (\langle simp,\ s,\ i\rangle) : c\,\{\langle close,\ i+j\rangle\})$. The corresponding machine transitions are

$$\begin{aligned}
&\xrightarrow{SimpCap} \langle pc+1,\ s,\ i,\ e,\ k\,\{\langle simp,\ s,\ i\rangle\}\rangle \\
&\xrightarrow{\Pi(g,x+1,p)} \langle pc + |\Pi(g, x, p)| + 1,\ s,\ i+j,\ e,\ k\,(\langle simp,\ s,\ i\rangle) : c\rangle \\
&\xrightarrow{CloseCap} \langle pc + |\Pi(g, x, p)| + 2,\ s,\ i+j,\ e,\ k\,(\langle simp,\ s,\ i\rangle) : c\,\{\langle close,\ i+j\rangle\}\rangle
\end{aligned}$$

$$
\begin{aligned}
\text{eval}\ (\{\}, v) &= (\{\}, v) \\
\text{eval}\ (\langle \mathit{close},\ i\rangle : c, v) &= (\langle \mathit{close},\ i\rangle : c, v) \\
\text{eval}\ (\langle \mathit{const},\ x\rangle : c, v) &= \text{eval}\ (c, v\,\{x\}) \\
\text{eval}\ (\langle \mathit{simp},\ s,\ i\rangle : c, v) &= \text{esimp}\ (s, i, v, (c, \{\})) \\
\text{eval}\ (\langle \mathit{func},\ f\rangle : c, v) &= \text{efunc}\ (f, v, (c, \{\})) \\
\text{eval}\ (\langle \mathit{fold},\ f\rangle : c, v) &= \text{efold}\ (f, v, (c, \{\}))
\end{aligned}
$$

$$
\begin{aligned}
\text{esimp}\ (s, i, v_1, (\langle \mathit{close},\ j\rangle : c, v_2)) &= \text{eval}\ (c, v_1(s[i, j] : v_2)) \\
\text{esimp}\ (s, i, v_1, (c, v_2)) &= \text{esimp}\ (s, i, v_1, \text{eval}\ (c, v_2))
\end{aligned}
$$

$$
\begin{aligned}
\text{efunc}\ (f, v_1, (\langle \mathit{close},\ j\rangle : c, v_2)) &= \text{eval}\ (c, v_1\, f(v_2)) \\
\text{efunc}\ (f, v_1, (c, v_2)) &= \text{efunc}\ (f, v_1, \text{eval}\ (c, v_2))
\end{aligned}
$$

$$
\begin{aligned}
\text{efold}\ (f, v_1, (\langle \mathit{close},\ j\rangle : c, v_2)) &= \text{eval}\ (c, v_1\,\{\mathit{fold}(f, v_2)\}) \\
\text{efold}\ (f, v_1, (c, v_2)) &= \text{efold}\ (f, v_1, \text{eval}\ (c, v_2))
\end{aligned}
$$

Figure 6. Inductive definitions of `eval`, `esimp`, `efunc`, and `efold`

with the second transition using our induction hypothesis.

Now we will define an $eval : Capture^* \times List \rightarrow Capture^* \times List$ function that yields a list of values when applied to a list of captures and an initial list of values (usually empty). Figure 6 is an inductive definition of eval, using convenience functions (also inductively defined) for simple, function, and fold captures. The notation $s[i, j]$ takes a slice of the list s from position i to position j (including element i but not element j). Function $fold : (Value \times Value \rightarrow Value) \times List \rightarrow Value$ is the common left fold function.

After applying `eval` to the list of captures constructed by match_L (or our machine) and an empty list of values, the result is an empty list of captures and a list of captured values.

6. Performance

In this section we present some performance numbers of our extension to LPEG, an implementation of our parsing machine as a library for the Lua programming language.

First we will evaluate the optimizations of Section 4 with a few simple benchmarks. The results are summarized on the left graph of Figure 7, showing the running time of the benchmarks for the base LPEG, LPEG with the string optimization, and LPEG with the string and list choice optimization.

The benchmarks are on the vertical axis: *string* uses a concatenation of string-like patterns (150,000 iterations), *head-fail* uses an ordered choice of these patterns (15,000 iterations), and *choice* uses an ordered choice of lists of string-like patterns (40,000 iterations). The benchmarks with ordered choice use subjects that match each alternative of the choice. The graph shows a moderate improvement on running times when the optimizations are used (close to half the time in some cases).

Now we will evaluate LPEG compared to hand-written recursive parsers (in Lua) and to OMeta/JS, another PEG-based pattern matcher that matches structured data. For this evaluation we will use two benchmarks, the first is a simple evaluator of an arithmetic AST. This grammar, in LPEG syntax, is

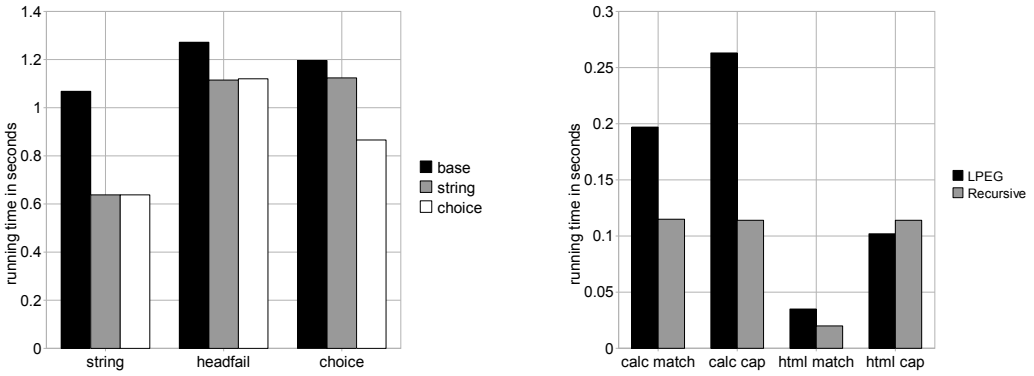

Figure 7. Benchmark results

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

with `<exp>` a non-terminal pattern, `<.>` a simple capture of pattern `.`, and `-> add` a function capture. A string like `¨add¨` is sugar for `{'a''d''d'}`.

The second benchmark parses a tree representing HTML-like data and extracts a list of all values of `src` attributes from `img` tags and `href` attributes of `a` tags. The grammar is

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

We run both benchmarks both with captures and semantic actions and for matching only. The right graph of Figure 7 summarizes the results. They show that list processing with LPEG is about as efficient as writing your own list-processing code. We do not show the results for OMeta/JS on the graph because of scale, running times were 8.3, 8.65, 1.78, and 2.06 seconds, respecively.

7. Related Work

OMeta is an object-oriented language that uses PEGs as a control mechanism [Warth and Piumarta 2007], and is embedded in other languages (with a JavaScript embedding being the most mature). OMeta also allows pattern matching on list-structured data, and OMeta's authors have an alternative operational semantics for list matching and semantic actions for PEGs [Warth 2009].

While in our semantics a match only produces values if there are captures, in OMeta captures are implicit in the semantic rules (OMeta patterns always produce values). OMeta also executes semantic actions during pattern matching, so the programmer has to be careful with side effects on semantic actions.

OMeta has a kind of semantic action called a *semantic predicate*, which can interfere with the match. We did not include it in this paper due to space, but LPEG has a similar mechanism called a *match-time capture*.

Several programming languages, mostly in the ML family, include built-in pattern matching facilities to help define functions over structured data [Peyton Jones 1987, Peyton Jones 2003]. The patterns are limited, not providing any looping of repetition operators, or the ability to reference other patterns. Pattern variables capture data from the subject, and are bound to parameters of the functions.

Scheme has an implementation of pattern matching that works on lists that is also primarily intended for destructuring data [Wright 1996]. It has repetition, but patterns still can't reference other patterns, so it also is strictly less powerful than PEGs. Wright's patterns also can only capture data in the form of pattern variables.

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* [Hosoya and Pierce 2001, Hosoya et al. 2005], an extension to algebraic types that includes repetition. The patterns that describe these types can reference other patterns (recursion is also supported), 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 [Hutton and Meijer 1998]. 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. Their unrestricted backtracking can have a large time and space cost (naive implementations are particularly prone to space leaks [Leijen and Meijer 2001]).

8. Conclusions

We presented an extension of Parsing Expression Grammars that allows matching of list-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.

Continuing a previous work, we also extended a parsing virtual machine for PEGs with new instructions for compiling list patterns, and proved the correctness of this machine and the compilation. A few non-essential instructions were also added to the machine to enable two list-specific optimizations.

We also added captures and semantic actions to PEGs, augmenting both the previous operational semantics and the parsing machine. Captures and semantic actions do not produce values, but promises of these values. A separate evaluation step executes the captures and semantic actions and produces the values. This makes side-effects in semantic actions safe in the presence of backtracking.

We benchmark list matching and our optimizations on LPEG, an implementation of our parsing machine for the Lua language. The optimizations are show to be effective, and list matching using LPEG to be about as efficient as a hand-written recursive parser, even when in the presence of captures and semantic actions. Our benchmarks also show

an order-of-magnitude improvement over OMeta/JS, another PEG-based pattern matcher that operates over lists.

LPEG has a richer set of capture patterns than we have presented here, such as position captures (useful for error reporting), argument captures (useful for passing state to semantic actions), named captures, and the math-time captures mentioned in the previous section. A formal treatment of these extra capture patterns is straightforward and will be included in a future work.

References

Ford, B. (2004). Parsing expression grammars: a recognition-based syntactic foundation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 111–122, New York, NY, USA. ACM.

Hosoya, H. and Pierce, B. (2001). Regular expression pattern matching for XML. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 67–80, New York, NY, USA. ACM.

Hosoya, H., Vouillon, J., and Pierce, B. C. (2005). Regular expression types for XML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(1):46–90.

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

Ierusalimschy, R. (2006). *Programming in Lua, Second Edition*. Lua.Org.

Ierusalimschy, R. (2009). A text pattern-matching tool based on Parsing Expression Grammars. *Software - Practice and Experience*, 39(3):221–258.

Leijen, D. J. P. and Meijer, H. J. M. (2001). Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-35, Department of Information and Computing Sciences, Utrecht University.

Medeiros, S. and Ierusalimschy, R. (2008). A parsing machine for PEGs. In *DLS '08: Proceedings of the 2008 symposium on Dynamic languages*, pages 1–12, New York, NY, USA. ACM.

Peyton Jones, S. (2003). *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press.

Peyton Jones, S. L. (1987). *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Warth, A. (2009). *Experimenting with Programming Languages*. PhD thesis, University of California Los Angeles.

Warth, A. and Piumarta, I. (2007). Ometa: an object-oriented language for pattern matching. In *DLS '07: Proceedings of the 2007 symposium on Dynamic languages*, pages 11–19, New York, NY, USA. ACM.

Wright, A. K. (1996). Pattern Matching for Scheme. Technical report, Department of Computer Science, Rice University.