A Parsing Machine for PEGs

Sérgio Medeiros[*]
smedeiros@inf.puc-rio.br

Roberto Ierusalimschy
roberto@inf.puc-rio.br

Department of Computer Science
PUC-Rio, Rio de Janeiro, Brazil

ABSTRACT

Parsing Expression Grammar (PEG) is a recognition-based foundation for describing syntax that renewed interest in top-down parsing approaches. Generally, the implementation of PEGs is based on a recursive-descent parser, or uses a memoization algorithm.

We present a new approach for implementing PEGs, based on a virtual parsing machine, which is more suitable for pattern matching. Each PEG has a corresponding program that is executed by the parsing machine, and new programs are dynamically created and composed. The virtual machine is embedded in a scripting language and used by a pattern-matching tool.

We give an operational semantics of PEGs used for pattern matching, then describe our parsing machine and its semantics. We show how to transform PEGs to parsing machine programs, and give a correctness proof of our transformation.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; D.3.1 [**Programming Languages**]: Formal Definitions and Theory

General Terms

Operational semantics, scripting languages

Keywords

parsing machine, Parsing Expression Grammars, pattern matching

1. INTRODUCTION

The Parsing Expression Grammar (PEG) formalism for language recognition [2] has renewed interest in top-down parsing approaches. The PEG formalism gives a convenient syntax for describing top-down parsers for unambiguous languages. The parsers it describes can parse strings in linear time, despite backtracking, by using a memoizing algorithm called Packrat [1]. Although Packrat has guaranteed worst-case linear time complexity, it also has linear *space* complexity, with a rather large constant. This makes Packrat impractical for parsing large amounts of data.

LPEG [7, 11] is a pattern-matching tool for Lua, a dynamically typed scripting language [6, 8]. LPEG uses PEGs to describe patterns instead of the more popular Perl-like ”regular expressions” (regexes). Regexes are a more ad-hoc way of describing patterns; writing a complex regex is more a trial and error than a systematic process, and their performance is very unpredictable. PEGs offer a way of writing patterns that is simple for small patterns, while providing better organization for the more complex patterns.

As pattern-matching tools usually have to deal with much larger inputs than parsing tools, this precluded the use of a Packrat-like algorithm for LPEG. Therefore, LPEG adopts a new approach, using a *virtual parsing machine*, where each pattern translates to a program for this machine. These programs are built at runtime, and can also be dynamically composed into bigger programs (that represent bigger patterns). This piecemeal construction of programs fits both the dynamic nature of Lua programming, and the composability of the PEG formalism.

As each PEG pattern is compiled in a sequence of instructions of our parsing machine during runtime, this makes the model of LPEG much more appropriated to be used by a dynamic language when compared to a more traditional approch, which uses static compilation and linking of programs generated from PEG grammars.

The parsing machine is also fast, with the performance of LPEG being similar to the performance of other parser and pattern-matching tools. Moreover, the machine has a very simple model, which makes its implementation easy, and seems a very good candidate for a Just-In-Time (JIT) compiler.

In a forthcoming paper [7], one of the authors presents the LPEG tool more in depth, giving several examples of its use. That paper also gives an operational semantics of the parsing machine instructions, plus an informal description of how the machine executes a program and some benchmarks comparing LPEG with POSIX regex and PCRE.

The main contribution of the this paper resides on the full formal specification of the parsing machine, and on the proof of correctness of the transformation between PEG patterns

[*]Supported by CNPq

$\varepsilon \in$ Pattern
'.' $\in$ Pattern
'c' $\in$ Pattern
If $p \in$ Pattern then $p?$, $p*$, $p+$, $\&p$, and $!p \in$ Pattern
If p_1 and $p_2 \in$ Pattern then p_1p_2 and $p_1/p_2 \in$ Pattern

Table 1: Definition of PEG patterns

and programs of our machine.

The present paper also discuss some optimizations applicable to our machine which reduce the amount of backtracking without using memoization, and gives the associated transformation. As our machine is explicitly designed for PEGs, its design relies on the fact that PEGs only use limited backtracking.

The rest of this paper is organized as follows: Section 2 reviews some PEG concepts and describes the virtual machine and its operational semantics; Section 3 proves the transformation between simple PEGs (those consisting of a single production) and their corresponding programs is correct; Section 4 augments the virtual machine to enable some optimizations, and proves the correctness of these optimizations; Section 5 extends the previous proofs to cover PEGs in general; Section 6 discusses some related work and presents a benchmark comparing LPEG with other tools; finally, Section 7 summarizes our results.

2. A PARSING MACHINE FOR PEGS

Parsing Expression Grammar (PEGs) is a recognition-based formal foundation for language syntax [2]. Although PEGs are similar to Context Free Grammars (CFGs), a key difference is that, while a CFG generates a language, a PEG describes a parser. PEGs have limited backtracking through an *ordered choice* operator, and unlimited lookahead through *syntactic predicates*.

The original PEG paper uses a grammar-like formal definition of PEGs. We are going to start with a simpler, less powerful definition, which is equivalent to the right-hand side of a PEG production (minus any non-terminals). We call these simple expressions *patterns*. Later in the paper we extend patterns to reintroduce grammars, but in a way that preserves their composability. Table 1 has the abstract syntax of patterns.

We have that ε represents the empty string, which is a pattern that never fails.

The pattern '.' matches any character, while the pattern 'c' matches a given character *c*. The pattern $p?$ represents an optional match and it always succeeds. The repetition pattern $p*$ always succeeds too. It matches the pattern p zero or more times. The other repetition pattern, $p+$, matches p one or more times and can fail.

The symbols ! and & are predicate operators, and do not consume any input. The pattern $!p$ succeeds if the matching of pattern p fails, and succeeds otherwise. The pattern $\&p$ is the opposite, it succeeds when pattern p succeeds, and fails when p fails.

The concatenation of two patterns p_1 and p_2 is indicated simply by the juxtaposition of both patterns. For example, p_1p_2 indicates the concatenation of p_1 and p_2.

The / is the ordered choice operator and allows a limited form of backtracking. A pattern like p_1/p_2, first tries to match the pattern p_1, and if this match succeeds the whole pattern succeeds. Otherwise, we try pattern p_2, and if it succeeds the whole pattern succeeds too, otherwise the whole pattern fails.

Figure 1 has an operational semantics for patterns, where match : Pattern $\times \Sigma^* \times \mathcal{N} \to \mathcal{N} \cup \{nil\}$ is the semantics of a pattern, given a subject and a position (position 1 is the beginning of the subject). If a match succeeds, the result is the position in the subject after the match, otherwise, if the match fails, the result is *nil*.

The looping rule (for repetition) of the semantics precludes us from doing a straightforward structural induction on patterns. We have to do induction on a measure of pattern complexity that we will state as $\mid p \mid + \mid s \mid - i$, that is, the size of the pattern plus the length of the subject, minus the current position. For all semantic rules but the looping rule, it is trivial to see that the antecedent has a smaller measure than the consequent, as the pattern itself is smaller, and the position doesn't decrease. For the looping rule, the induction will hold only when the position advances by at least one; in other words, induction is only possible when $j > 0$. Informally, repetition on a pattern that succeeds but does not advance leads to an infinite loop.

Usually, each PEG pattern can be associated with a recursive-descent parser, but we will take a different approach. Instead, each PEG pattern translates to a program, which executes on a virtual parsing machine.

Our parsing machine for PEGs has more in common with virtual machines for imperative programming languages than with abstract machines in language theory. It executes programs made up of atomic instructions that change machine's state. The machine has a program counter that addresses the next instruction to be executed, a register that holds the current position in the subject, and a stack that the machine uses for pushing both return addresses and backtrack entries. A return address is just a new value for the program counter, while a backtrack entry holds both an address and a position in the subject. The machine has the following basic instructions:

`Char x`: tries to match the character x against the current subject position, advancing one position if successful.

`Any`: advances one position if the end of the subject was not reached; it fails otherwise.

`Choice l`: pushes a backtrack entry on the stack, where l is the offset of the alternative instruction.

`Jump l`: relative jump to the instruction at offset l.

`Call l`: pushes the address of the next instruction in the stack, and jumps to the instruction at offset l.

`Return`: pops an address from the stack and jumps to it.

`Commit l`: commits to a choice, popping the top entry from the stack, throwing it away, and jumping to the instruction at offset l.

`Fail`: forces a failure. When any failure occurs, the machine pops the stack until it finds a backtrack entry, then uses that entry plus the stack as the new machine state.

Figure 2 presents the operational semantics of the parsing machine as a relation among machine states. The program $\mathcal{P}$ that the machine is executing, and the subject $\mathcal{S}$, are implicit. The state is either a triple $\mathcal{N} \times \mathcal{N} \times$ Stack, with the next instruction to execute (pc), the current position in the subject (i), and a stack (e), or $\textbf{Fail}\langle e\rangle$, a failure state with stack e. Stacks are lists of $\mathcal{N} \cup \mathcal{N} \times \mathcal{N}$, where a stack position of form $\mathcal{N}$ represents a return address, while a stack position of form $\mathcal{N} \times \mathcal{N}$ represents a backtrack entry.

Matching a character

$$\frac{s[i] = \text{‘c’}}{\texttt{match}\ \text{‘c’}\ s\ i = \text{i+1}}\ \textbf{(ch.1)} \qquad \frac{s[i] \neq \text{‘c’}}{\texttt{match}\ \text{‘c’}\ s\ i = \texttt{nil}}\ \textbf{(ch.2)}$$

Matching any character

$$\frac{i \leq |s|}{\texttt{match}\ \texttt{.}\ s\ i = \text{i+1}}\ \textbf{(any.1)} \qquad \frac{i > |s|}{\texttt{match}\ \texttt{.}\ s\ i = \texttt{nil}}\ \textbf{(any.2)}$$

Not Predicate

$$\frac{\texttt{match}\ p\ s\ i = \texttt{nil}}{\texttt{match}\ !p\ s\ i = \text{i}}\ \textbf{(not.1)} \qquad \frac{\texttt{match}\ p\ s\ i = \text{i+j}}{\texttt{match}\ !p\ s\ i = \texttt{nil}}\ \textbf{(not.2)}$$

And Predicate

$$\frac{\texttt{match}\ p\ s\ i = \text{i+j}}{\texttt{match}\ \&p\ s\ i = \text{i}}\ \textbf{(and.1)} \qquad \frac{\texttt{match}\ p\ s\ i = \texttt{nil}}{\texttt{match}\ \&p\ s\ i = \texttt{nil}}\ \textbf{(and.2)}$$

Concatenation

$$\frac{\texttt{match}\ p_1\ s\ i = \text{i+j} \qquad \texttt{match}\ p_2\ s\ i+j = \text{i+j+k}}{\texttt{match}\ p_1p_2\ s\ i = \text{i+j+k}}\ \textbf{(con.1)}$$

$$\frac{\texttt{match}\ p_1\ s\ i = \text{i+j} \qquad \texttt{match}\ p_2\ s\ i+j = \texttt{nil}}{\texttt{match}\ p_1p_2\ s\ i = \texttt{nil}}\ \textbf{(con.2)} \qquad \frac{\texttt{match}\ p_1\ s\ i = \texttt{nil}}{\texttt{match}\ p_1p_2\ s\ i = \texttt{nil}}\ \textbf{(con.3)}$$

Ordered Choice

$$\frac{\texttt{match}\ p_1\ s\ i = \texttt{nil} \qquad \texttt{match}\ p_2\ s\ i = \texttt{nil}}{\texttt{match}\ p_1/p_2\ s\ i = \texttt{nil}}\ \textbf{(ord.1)}$$

$$\frac{\texttt{match}\ p_1\ s\ i = \text{i+j}}{\texttt{match}\ p_1/p_2\ s\ i = \text{i+j}}\ \textbf{(ord.2)} \qquad \frac{\texttt{match}\ p_1\ s\ i = \texttt{nil} \qquad \texttt{match}\ p_2\ s\ i = \text{i+k}}{\texttt{match}\ p_1/p_2\ s\ i = \text{i+k}}\ \textbf{(ord.3)}$$

Repetition

$$\frac{\texttt{match}\ p\ s\ i = \text{i+j} \qquad \texttt{match}\ p*\ s\ i+j = \text{i+j+k}}{\texttt{match}\ p*\ s\ i = \text{i+j+k}}\ \textbf{(rep.1)} \qquad \frac{\texttt{match}\ p\ s\ i = \texttt{nil}}{\texttt{match}\ p*\ s\ i = \text{i}}\ \textbf{(rep.2)}$$

Figure 1: Operational Semantics of Patterns

The relation $\xrightarrow{\text{Instruction}}$ relates two states when the instruction addressed by pc in the antecedent state matches the label, and the guard (if present) is valid. The transitive closure of this relation is an execution of the machine.

3. TRANSFORMING PATTERNS TO PROGRAMS

As was mentioned earlier, each PEG pattern translates to a program that executes in a parsing machine. The process of translating a pattern into a program is bottom up, and done at runtime. The simplest patterns are translated to simple programs, and then programs are combined according to rules specific for 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. In our implementation the process is fully incremental, and combining programs is a simple matter of concatenating their texts.

To represent this compilation process, we will use a transformation function Π, from Pattern to Program. Given a pattern p, we have that $\Pi(p)$ represents the program associated with pattern p. We will also use the notation $|\Pi(p)|$, which means the number of instructions of the program $\Pi(p)$.

The next step is to prove the correctness of the transformation that converts a PEG pattern to a corresponding program for the virtual parsing machine.

All transformations use concatenation to build bigger programs from smaller ones. Formally, given programs $\mathcal{P}_1$ and $\mathcal{P}_2$, such that:

$$\langle pc_1,\, i,\, e\rangle \xrightarrow{\mathcal{P}_1} \langle pc_1 + |\mathcal{P}_1|,\, i+j,\, e\rangle$$

$$\langle pc_2,\, i+j,\, e\rangle \xrightarrow{\mathcal{P}_2} \langle pc_2 + |\mathcal{P}_2|,\, i+j+k,\, e\rangle$$

The concatenation $\mathcal{P}_1\mathcal{P}_2$ has the following property:

$$\langle pc,\, i,\, e\rangle \xrightarrow{\mathcal{P}_1\mathcal{P}_2} \langle pc + |\mathcal{P}_1| + |\mathcal{P}_2|,\, i+j+k,\, e\rangle$$

This property holds when all the jumps of a program are for instructions of the program, or for the first instruction of the next program.

Considering $\langle pc,\, i,\, e\rangle$ as the machine's initial state, where pc points to the first instruction of $\Pi(p)$, we have that a pattern p is equivalent to a program $\Pi(p)$, if when p succeeds, then the following relation is true:

$$\texttt{If match } p\ s\ i = \mathrm{i} + \mathrm{j} \texttt{ then}$$

$$\langle pc,\, i,\, e\rangle \xrightarrow{\Pi(p)} \langle pc + |\Pi(p)|,\, i+j,\, e\rangle$$

We have also that, if p fails, then the following relation should hold:

$$\texttt{If match } p\ s\ i = \texttt{nil then}$$

$$\langle pc,\, i,\, e\rangle \xrightarrow{\Pi(p)} \textbf{Fail}\langle e\rangle$$

It can be noticed that in both cases the stack remains the same after the execution of the program $\Pi(p)$.

We start proving that the transformation Π is correct for simple patterns, and after we prove the transformation is also correct for more complex patterns.

Let's then take a look on a PEG pattern that matches a certain c character. The transformation here is $\Pi(\text{‘c’}) \equiv$ `Char` c, such that we can express the pattern ‘c’ directly by a `Char` instruction.

Considering first the case when the `match` function succeeds, what we want to prove is:

$$\texttt{If match } \text{‘c’}\ s\ i = \mathrm{i} + \mathrm{j} \texttt{ then}$$

$$\langle pc,\, i,\, e\rangle \xrightarrow{Char} \langle pc+1,\, i+1,\, e\rangle$$

In this case, as we want to match just one character, we have that j must be 1. Looking Figure 1, we can see the rule ($ch.1$) is the only one involving the successful matching of a character. By this rule, we know that $s[i] = \text{‘c’}$.

It is trivial to see that the semantics of the `Char` instruction guarantees that our program will advance the current subject position by 1, and will not change the stack, when the matching of ‘c’ succeeds.

Considering now the case where the matching fails, we need to prove that:

$$\texttt{If match } p\ s\ i = \texttt{nil then}$$

$$\langle pc,\, i,\, e\rangle \xrightarrow{Char} \textbf{Fail}\langle e\rangle$$

The semantic rule ($ch.2$) is the only rule related to the unsuccessful matching of a character. By this rule, we know that $s[i] \neq \text{‘c’}$.

Again, we can rely on the semantics of the `Char` instruction, which will lead the parsing machine to a state indicating that a fail occurred, when the pattern does not match. This completes the proof.

The case of the `Any` instruction is very similar to the case of the `Char` instruction, because there is also a direct transformation from a pattern p that matches any character, to a program that consists only of an `Any`. The correctness of this transformation can be proved with the help of the semantic rules ($any.1$) and ($any.2$).

The next proof is about the concatenation of two PEG patterns, p_1 and p_2. We state the following transformation related to the concatenation of two patterns:

$$\Pi(p_1\, p_2) \equiv \Pi(p_1)\ \Pi(p_2)$$

In other words, when transforming p_1p_2 to a program, we will first execute the program $\Pi(p_1)$, and then program $\Pi(p_2)$. Considering the case where `match` succeeds, what we are going to prove is:

$$\texttt{If match } p_1p_2\ s\ i = \mathrm{i} + \mathrm{j} + \mathrm{k} \texttt{ then}$$

$$\langle pc,\, i,\, e\rangle \xrightarrow{\Pi(p_1p_2)} \langle pc + |\Pi(p_1)| + |\Pi(p_2)|,\, i+j+k,\, e\rangle$$

The rule ($con.1$) is the one related to the successful concatenation of two patterns. By this rule, we know that p_1 and p_2 matched, with the matching of p_1 advancing j positions of the input string, and the matching of p_2 advancing more k positions.

Let $\langle pc,\, i,\, e\rangle$ be the initial virtual machine state. The corresponding program $\Pi(p_1p_2)$ of our PEG machine will execute the following sequence of transitions, where in both transitions we use the induction hypothesis:

$$\xrightarrow{\Pi(p_1)} \langle pc + |\Pi(p_1)|,\, i+j,\, e\rangle$$

$$\xrightarrow{\Pi(p_2)} \langle pc + |\Pi(p_1)| + |\Pi(p_2)|,\, i+j+k,\, e\rangle$$

This proves the first case.

The other case is when p_1p_2 fails. What we have to prove

$$
\begin{array}{rcll}
\langle pc,\, i,\, e\rangle & \xrightarrow{\texttt{Char } x} & \langle pc+1,\, i+1,\, e\rangle & \mathcal{S}[i] = x \\
\langle pc,\, i,\, e\rangle & \xrightarrow{\texttt{Char } x} & \textbf{Fail}\langle e\rangle & \mathcal{S}[i] \neq x \\
\langle pc,\, i,\, e\rangle & \xrightarrow{\texttt{Any}} & \langle pc+1,\, i+1,\, e\rangle & i+1 \leq |\mathcal{S}| \\
\langle pc,\, i,\, e\rangle & \xrightarrow{\texttt{Any}} & \textbf{Fail}\langle e\rangle & i+1 > |\mathcal{S}| \\
\langle pc,\, i,\, e\rangle & \xrightarrow{\texttt{Choice } l} & \langle pc+1,\, i,\, (pc+l, i) : e\rangle & \\
\langle pc,\, i,\, e\rangle & \xrightarrow{\texttt{Jump } l} & \langle pc+l,\, i,\, e\rangle & \\
\langle pc,\, i,\, e\rangle & \xrightarrow{\texttt{Call } l} & \langle pc+l,\, i,\, (pc+1) : e\rangle & \\
\langle pc_0,\, i,\, pc_1 : e\rangle & \xrightarrow{\texttt{Return}} & \langle pc_1,\, i,\, e\rangle & \\
\langle pc,\, i,\, h : e\rangle & \xrightarrow{\texttt{Commit } l} & \langle pc+l,\, i,\, e\rangle & \\
\langle pc,\, i,\, e\rangle & \xrightarrow{\texttt{Fail}} & \textbf{Fail}\langle e\rangle & \\
\textbf{Fail}\langle pc : e\rangle & \xrightarrow{any} & \textbf{Fail}\langle e\rangle & \\
\textbf{Fail}\langle (pc, i_1) : e\rangle & \xrightarrow{any} & \langle pc,\, i_1,\, e\rangle &
\end{array}
$$

Figure 2: Operational Semantics of the Parsing Machine

is:

$$\texttt{If match } p_1 p_2\ s\ i = \texttt{nil then}$$
$$\langle pc,\, i,\, e\rangle \xrightarrow{\Pi(p_1 p_2)} \textbf{Fail}\langle e\rangle$$

From Figure 1, we can see there are two rules related to the concatenation which result in a fail. Let's first consider the rule (*con*.3). In this case, the pattern p_1 fails, and the machine will do the following transition:

$$\langle pc,\, i,\, e\rangle \xrightarrow{\Pi(p_1)} \textbf{Fail}\langle e\rangle$$

With $\Pi(p_2)$ not being executed.

Let's now consider the rule (*con*.2), in which p_1 succeeds, advancing j positions of the input, but the pattern p_2 fails. This is represented by the following sequence of transitions:

$$\xrightarrow{\Pi(p_1)} \langle pc + |\Pi(p_1)|,\, i+j,\, e\rangle$$
$$\xrightarrow{\Pi(p_2)} \textbf{Fail}\langle e\rangle$$

What proves the transformation for the pattern $p_1\, p_2$ is correct.

Let's now see the case of the ordered choice operator. Given the PEG pattern p_1/p_2, we have the following transformation:

$$
\begin{array}{rl}
\Pi(p_1/p_2) \equiv & \texttt{Choice } |\Pi(p_1)| + 2 \\
& \Pi(p_1) \\
& \texttt{Commit } |\Pi(p_2)| + 1 \\
& \Pi(p_2)
\end{array}
$$

The transformation says that $\Pi(p_1/p_2)$ is equivalent to a program where the first instruction is a `Choice`, followed by the program $\Pi(p_1)$, then by a `Commit` instruction, and finally by program $\Pi(p_2)$.

Considering the case when p_1/p_2 succeeds, what we are going to prove is:

$$\texttt{If match } p_1/p_2\ s\ i = \texttt{i + j then}$$
$$\langle pc,\, i,\, e\rangle \xrightarrow{\Pi(p_1/p_2)} \langle pc + |\Pi(p_1)| + |\Pi(p_2)| + 2,\, i+j,\, e\rangle$$

There are two rules related to the success of an ordered choice. Let's first consider the rule (*ord*.2). We know that p_1 matched, advancing j positions of the input subject.

The corresponding sequence of transitions, assuming $\langle pc,\, i,\, e\rangle$ as the initial state, where pc points to the first instruction of $\Pi(p_1/p_2)$, is shown below:

$$\xrightarrow{Choice} \langle pc+1,\, i,\, (pc + |\Pi(p_1)| + 2, i) : e\rangle$$
$$\xrightarrow{\Pi(p_1)} \langle pc + |\Pi(p_1)| + 1,\, i+j,\, (pc + |\Pi(p_1)| + 2, i) : e\rangle$$
$$\xrightarrow{Commit} \langle pc + |\Pi(p_1)| + |\Pi(p_2)| + 2,\, i+j,\, e\rangle$$

The first thing the program does is to save the current state of the machine, because if p_1 fails, we have to backtrack. As p_1 succeeds, we just discard the backtrack entry and jump to the end of the program, using the `Commit` instruction.

We can notice that after the complete execution of the program, the stack remains the same, with the backtrack entry being discarded:

Analyzing now the rule (*ord*.3), we have that p_1 fails, and p_2 succeeds, matching k characters, such that $j = k$ in this case.

The associated sequence of transitions is:

$$\xrightarrow{Choice} \langle pc+1,\, i,\, (pc + |\Pi(p_1)| + 2, i) : e\rangle$$
$$\xrightarrow{\Pi(p_1)} \textbf{Fail}\langle (pc + |\Pi(p_1)| + 2, i) : e\rangle$$
$$\xrightarrow{Fail} \langle pc + |\Pi(p_1)| + 2,\, i,\, e\rangle$$
$$\xrightarrow{\Pi(p_2)} \langle pc + |\Pi(p_1)| + |\Pi(p_2)| + 2,\, i+k,\, e\rangle$$

The program first pushes a backtrack entry, and tries to match p_1. As p_1 fails, the machine backtracks, and then matches p_2, what advances k positions of the input string, as expected.

Considering now the case where the `match` function fails, what we have to prove is:

$$\texttt{If match } p_1/p_2\ s\ i = \texttt{nil then}$$
$$\langle pc,\, i,\, e\rangle \xrightarrow{\Pi(p_1/p_2)} \textbf{Fail}\langle e\rangle$$

The rule (*ord*.1) is the only one related to this case. By this rule, we know that neither p_1 nor p_2 match.

The sequence of transitions associated with this case is shown below:

$$\begin{aligned}
&\xrightarrow{Choice} \langle pc+1,\, i,\, (pc+|\Pi(p_1)|+2, i) : e\rangle\\
&\xrightarrow{\Pi(p_1)} \textbf{Fail}\langle (pc+|\Pi(p_1)|+2, i) : e\rangle\\
&\xrightarrow{Fail} \langle pc+|\Pi(p_1)|+2,\, i,\, e\rangle\\
\langle pc+|\Pi(p_1)|+2,\, i,\, e\rangle &\xrightarrow{\Pi(p_2)} \textbf{Fail}\langle e\rangle
\end{aligned}$$

This sequence is similar to the previous one, but now as p_2 fails, the machine goes to a fail state. This completes the proof of the ordered choice.

We will now consider the transformation of the not predicate. Let p be a pattern, we have that:

$$\begin{aligned}
\Pi(!p) \equiv\ & \texttt{Choice}\ |\Pi(p)|+3\\
& \Pi(p)\\
& \texttt{Commit}\ 1\\
& \texttt{Fail}
\end{aligned}$$

Considering the case where the pattern $!p$ fails, we have that:

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

The semantic rule that corresponds to this case is ($not.2$). By this rule, we know the matching of p succeeds, although the pattern $!p$ fails. Therefore, we have the following sequence of transitions:

$$\begin{aligned}
&\xrightarrow{Choice} \langle pc+1,\, i,\, (pc+|\Pi(p)|+3, i) : e\rangle\\
&\xrightarrow{\Pi(p)} \langle pc+|\Pi(p)|+1,\, i+j,\, (pc+|\Pi(p)|+3, i) : e\rangle\\
&\xrightarrow{Commit} \langle pc+|\Pi(p)|+2,\, i+j,\, e\rangle\\
&\xrightarrow{Fail} \textbf{Fail}\langle e\rangle
\end{aligned}$$

The program first saves the current machine state, and next tries to match p. As p succeeds, the following `Commit` discards the backtrack entry, and the `Fail` instruction is then executed.

The other case, when pattern $!p$ succeeds, corresponds to the semantic rule ($not.1$). In this case, we have that $j = 0$. By ($not.1$), we know the matching of p fails, and the machine restores the entry pushed by the initial `Choice` instruction, what leads to the final state $\langle pc+|\Pi(p)|+3,\, i,\, e\rangle$.

With the transformation of the not predicate proved correct, given a pattern p, we can trivially define the and predicate as $!!p$. In the same way, as the concatenation and the ordered choice transformations of patterns were already proved correct too, we can define the predicate $p?$ as $\&pp\ /\ !p$.

The next step corresponds to the transformation of the repetition pattern. Let p be a PEG pattern, such that the construction $p*$ means zero or more repetitions of p. In this case, we have the following transformation from pattern to the virtual machine instructions:

$$\begin{aligned}
\Pi(p*) \equiv\ & \texttt{Choice}\ |\Pi(p)|+2\\
& \Pi(p)\\
& \texttt{Commit}\ -(|\Pi(p)|+1)
\end{aligned}$$

What we are going to prove first is:

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

This case is related to the semantic rule ($rep.2$), where $p*$ does not match an input s from position i. We have the following sequence of transitions:

$$\begin{aligned}
&\xrightarrow{Choice} \langle pc+1,\, i,\, (pc+|\Pi(p)|+2, i) : e\rangle\\
&\xrightarrow{\Pi(p)} \textbf{Fail}\langle (pc+|\Pi(p)|+2, i) : e\rangle\\
&\xrightarrow{Fail} \langle pc+|\Pi(p)|+2,\, i,\, e\rangle
\end{aligned}$$

We can see that the current subject position did not change with the execution of $\Pi(p*)$.

The other case we have to prove is:

$$\begin{aligned}
&\texttt{If match}\ p*\ s\ i = \mathrm{i{+}j{+}k}\ \texttt{then}\\
&\langle pc,\, i,\, e\rangle \xrightarrow{\Pi(p*)} \langle pc+|\Pi(p*)|,\, i+j+k,\, e\rangle
\end{aligned}$$

This case is related to the semantic rule ($rep.1$), and we have that $*p$ matches an input s from position i. The sequence of transitions is presented below:

$$\begin{aligned}
&\xrightarrow{Choice} \langle pc+1,\, i,\, (pc+|\Pi(p)|+2, i) : e\rangle\\
&\xrightarrow{\Pi(p)} \langle pc+1+|\Pi(p)|,\, i+j,\, (pc+|\Pi(p)|+2, i) : e\rangle\\
&\xrightarrow{Commit} \langle pc,\, i+j,\, e\rangle\\
&\xrightarrow{\Pi(p*)} \langle pc+|\Pi(p*)|,\, i+j+k,\, e\rangle
\end{aligned}$$

As we can see, p succeeds the first time, matching j characters, where we are considering $j > 0$. After this matching, we have a less complex pattern, considering our measure of pattern complexity as $\mid p \mid + \mid s \mid - \ i$. Then the machine executes `Commit`, coming back to the beginning of the program, and we use induction on the pattern complexity, reaching the final state $\langle pc+|Pi(p*)|,\, i+j+k,\, e\rangle$.

Since we have proved that the transformation from a pattern $p*$ to a program in our PEG machine was correct, and as we have also made the corresponding proof for the concatenation of patterns, we can simply define the pattern $p+$ as $p\,p*$.

4. OPTIMIZATIONS

As we have just proved, the set of instructions that we have been using is enough to define a virtual parsing machine for PEGs. But it would be convenient to define some extra instructions, so we can have more efficient implementations of the patterns. In short, we will define new instructions to get faster programs.

Figure 3 presents the semantics of the new set of instructions. To illustrate the use of these extra instructions, we will now redefine some of the patterns using the new set of instructions.

Let's start redefining the not predicate. In the new definition, we use the instruction `FailTwice`, which discards the top entry of the stack (in the case of the not predicate, it discards the entry pushed by the previous `Choice` instruction), and then fails, like the `Fail` instruction. The new

$$
\begin{aligned}
\langle pc_0,\, i_0,\, (pc_1, i_1) : e\rangle &\xrightarrow{\texttt{PartialCommit } l} \langle pc_0 + l,\, i_0,\, (pc_1, i_0) : e\rangle\\
\langle pc,\, i,\, h : e\rangle &\xrightarrow{\texttt{FailTwice}} \mathbf{Fail}\langle e\rangle\\
\langle pc_0,\, i_0,\, (pc_1, i_1) : e\rangle &\xrightarrow{\texttt{BackCommit } l} \langle pc_0 + l,\, i_1,\, e\rangle
\end{aligned}
$$

Figure 3: Operational Semantics of Extra Instructions

transformation is:

$$
\begin{aligned}
\Pi(!p) \equiv\ & \texttt{Choice}\ |\Pi(p)| + 2\\
& \Pi(p)\\
& \texttt{FailTwice}
\end{aligned}
$$

With the new transformation, the sequence of transitions when $!p$ fails, where the corresponding semantic rule is $(not.2)$, becomes:

$$
\begin{aligned}
&\xrightarrow{Choice} \langle pc+1,\, i,\, (pc + |\Pi(p)| + 2, i) : e\rangle\\
&\xrightarrow{\Pi(p)} \langle pc + |\Pi(p)| + 1,\, i+j,\, (pc + |\Pi(p)| + 2, i) : e\rangle\\
&\xrightarrow{FailTwice} \mathbf{Fail}\langle e\rangle
\end{aligned}
$$

Using `FailTwice`, we can see the number of transitions was smaller in this case. Comparing with the previous sequence of transitions of the not predicate, we can see that this transformation is also correct.

Another pattern that uses the extra instructions is the repetition pattern $p*$. One problem of the previous definition of $p*$ is that we are always pushing a new backtrack entry and discarding it when the pattern p matches.

We can notice from the previous definition of $p*$, that when we push a new backtrack entry, the only thing that changes is the current subject position, thus, instead of discarding the backtrack entry, we could have just updated it. This is exactly what the `PartialCommit` instruction does. By using it, we have the following transformation:

$$
\begin{aligned}
\Pi(p*) \equiv\ & \texttt{Choice}\ |\Pi(p)| + 2\\
& \Pi(p)\\
& \texttt{PartialCommit}\ -|\Pi(p)|
\end{aligned}
$$

As an example, we show below the new sequence of transitions for the repetition pattern when $p*$ matches an input s from position i, which corresponds to the semantic rule $(rep.1)$:

$$
\begin{aligned}
&\xrightarrow{Choice} \langle pc+1,\, i,\, (pc + |\Pi(p)| + 2, i) : e\rangle\\
&\xrightarrow{\Pi(p)} \langle pc + 1 + |\Pi(p)|,\, i+j,\\
&\qquad\qquad (pc + |\Pi(p)| + 2, i) : e\rangle\\
&\xrightarrow{PartialCommit} \langle pc+1,\, i+j,\, (pc + |\Pi(p)| + 2, i+j) : e\rangle\\
&\xrightarrow{\Pi(p*)} \langle pc + |\Pi(p*)|,\, i+j+k,\, e\rangle
\end{aligned}
$$

Notice that the `Choice` instruction executes just one time, as the `PartialCommit` instruction leads the machine back to the beginning of $\Pi(p)$. The machine does less transitions, and avoids the use of an expensive `Choice` instruction at each repetition step.

The `BackCommit` instruction is a variant of the `Commit` instruction. As the `Commit` instruction, the `BackCommit` instruction restores a backtrack entry, but it does not jump to the position pc_1 (as indicated in Figure 3). Instead of this, the `BackCommit` instruction receives a label l and the machine goes to the position $pc_0 + l$. This instruction is used in a more efficient implementation of the and predicate.

The current implementation of the parsing machine has also some specific instructions for character sets. There is a `Charset` instruction, which matches a character if it belongs to a set of characters. Another instruction related to character sets is `Span`, which is used to implement the repetition of a pattern that represents a character set.

These two instructions improve the performance of the machine when dealing with character sets, and can be easily translated to an equivalent program that uses the `Char` instruction.

4.1 Head-fail instructions

Although we have already extended the set of instructions of the virtual machine, and improved the definition of the patterns accordingly, we still can go further. Our PEG machine is designed for pattern-matching, so it should be fast when dealing with lexical structures, which sometimes is a bottleneck of PEGs implementations [13].

Because of this, there is a third group of instructions: the head-fail optimization instructions, which are presented in Figure 4. This group of instructions were created to address the problem of *head fail*s. If a pattern fails when doing its first check, we say it was a head fail. As an example, if we search for the word "edge" in an English language text, almost 90% of all fails may be head fails, given the typical frequency of around 11% for the letter 'e'. The head-fail instructions avoid the backtracking associated with a head fail.

The `TestChar` instruction, for example, checks if a given character is equal to the current subject position. In case it is, the machine matches the character and then executes the next instruction. In case it is not, the `TestChar` jumps to the offset. The advantage of using the `TestChar` instruction is that we can save the cost of a backtracking when a head-fail occurs.

As a consequence of the introduction of the head-fail optimization instructions, we need to modify the `Choice` instruction, adding an *offset* to it. Now, when this instruction saves the current machine state, it saves the current subject position minus a given offset. Thus, the new operational semantics of the `Choice` instruction is:

$$
\langle pc,\, i,\, e\rangle \xrightarrow{\texttt{Choice } l\ o} \langle pc+1,\, i,\, (pc + l, i - o) : e\rangle
$$

All the previous uses of `Choice` have an offset of zero.

Making use of the `TestChar` instruction, we can then optimize the ordered choice, as well as other patterns. Given patterns p_1 and p_2, let's consider an ordered choice of the form 'c'p_1/p_2, i.e., we first try to match a character 'c', and if we succeed then we try p_1, otherwise we try pattern p_2.

Instead of generating a trivial ordered choice code, we can

$$\begin{array}{llll}
\langle pc,\, i,\, e\rangle & \xrightarrow{\texttt{TestChar}\ x\ l} & \langle pc+1,\, i+1,\, e\rangle & \mathcal{S}[i] = x \\
\langle pc,\, i,\, e\rangle & \xrightarrow{\texttt{TestChar}\ x\ l} & \langle pc+l,\, i,\, e\rangle & \mathcal{S}[i] \neq x \\
\langle pc,\, i,\, e\rangle & \xrightarrow{\texttt{TestAny}\ n\ l} & \langle pc+1,\, i+n,\, e\rangle & i+n \leq |\mathcal{S}| \\
\langle pc,\, i,\, e\rangle & \xrightarrow{\texttt{TestAny}\ n\ l} & \langle pc+l,\, i,\, e\rangle & i+n > |\mathcal{S}|
\end{array}$$

Figure 4: Operational Semantics of Head-fail Instructions

generate the following optimized code:

$$\begin{aligned}
\Pi(\text{‘c’}p_1/p_2) \equiv\ & \texttt{TestChar}\ c\ \ |\Pi(p_1)|+3 \\
& \texttt{Choice}\ |\Pi(p_1)|+2\quad 1 \\
& \Pi(p_1) \\
& \texttt{Commit}\ |\Pi(p_2)|+1 \\
& \Pi(p_2)
\end{aligned}$$

The sequence of transitions below illustrates the case when ‘c’ fails, considering that p_2 matches:

$$\begin{aligned}
&\xrightarrow{TestChar}\langle pc+|\Pi(p_1)|+3,\, i,\, e\rangle \\
&\xrightarrow{\Pi(p_2)}\langle pc+|\Pi(p_1)|+|\Pi(p_2)|+3,\, i+k,\, e\rangle
\end{aligned}$$

As the matching of the character ‘c’ fails, the machine jumps directly to the beginning of $\Pi(p_2)$, without the need to execute the `Choice` instruction only to backtrack later.

We can do similar optimizations for other patterns, such as the not predicate.

Taking again the ordered choice as example, let's suppose that we have an ordered choice of the form ‘c_1’p_1/‘c_2’p_2, where c_1 and c_2 are different characters. In this case, we have that if c_1 matches, we know that c_2 will not match, and we do not need to push a backtrack entry at all. The corresponding sequence of instructions:

$$\begin{aligned}
\Pi(\text{‘}c_1\text{’}p_1/\text{‘}c_2\text{’}p_2) \equiv\ & \texttt{TestChar}\ c_1\ \ |\Pi(p_1)|+2 \\
& \Pi(p_1) \\
& \texttt{Jump}\ |\Pi(p_2)|+2 \\
& \Pi(\text{‘}c_2\text{’}) \\
& \Pi(p_2)
\end{aligned}$$

The following sequence of transitions shows the case when ‘c_1’ and p_1 match:

$$\begin{aligned}
&\xrightarrow{TestChar}\langle pc+1,\, i+1,\, e\rangle \\
&\xrightarrow{\Pi(p_1)}\langle pc+|\Pi(p_1)|+1,\, i+j+1,\, e\rangle \\
&\xrightarrow{Jump}\langle pc+|\Pi(p_1)|+|\Pi(p_2)|+|\Pi(\text{‘}c_2\text{’})|+2,\, i+j+1,\, e\rangle
\end{aligned}$$

First the `TestChar` instruction is executed, and then the machine executes $\Pi(p_1)$. After this, as there is no backtrack entry to discard, the machine simply jumps to end of the program.

5. GRAMMARS

In the previous sections we were considering only simple patterns in our formalization. Using only the abstract syntax in Table 1 we can't build patterns that reference themselves; the only way to use another pattern is to include it. We will now correct this by extending our previous definition of patterns with *grammars*.

We first introduce a countably infinite set of variables, or non-terminals, called V. We will refer to members of this set with the notation A_k, and extend our set of patterns to include all variables. We now define a grammar as a partial function from the set of variables to the set of patterns, such that $g(A_k)$ represents the pattern associated with non-terminal A_k of grammar g. Finally, we now extend the domain of our *match* function to Grammar $\times$ Pattern $\times\, \Sigma^* \times \mathcal{N}$. Figure 5 shows the semantics of the extended *match*.

We also introduce another new kind of pattern, the *closed grammar*, a pair Grammar $\times\ V$. Informally, a closed grammar matches a subject if the pattern referenced by the variable matches the subject, with all variables resolved in the grammar. Thus, if A_k is a non-terminal of grammar g, such that all variables of pattern $g(A_k)$ are resolved in g, then (g, A_k) is a closed grammar. Closed grammars build patterns from grammars, and allows different grammars to be composed using the standard PEG operators. Figure 5 also has the semantics of *match* for closed grammars.

Now we will present how to translate these concepts to our parsing machine, and prove the correctness of our translation. Our transformation Π will now operate on the domain Grammar $\times\ \mathcal{N} \times$ Pattern, where $\Pi(g, i, p)$ is the translation of pattern p in the context of grammar g and with position i relative to the beginning of the closed grammar which contains it (if the pattern is not part of a closed grammar then the value can be arbitrary). We will show shortly how both g and i are used to resolve variable references, representing the "linking" step of the transformation from patterns to programs.

Extending the transformations given in the previous sections is straightforward, and a matter of passing g to subprograms and keeping i correctly updated so as to keep its intended meaning. Appendix A gives the extended transformation. This extension is conservative, so the previous proofs will remain valid once we prove the correctness of the transformation for the patterns we introduced in this section.

To give the results of Π for variables and closed grammars, we first define a transformation Π' from Grammar $\times\ \mathcal{N}$ to Program. This is the invariant part of the transformation of all closed grammars (g, A_n) based on the grammar g. We

Matching a character
$$\frac{s[i] = \text{`c'}}{\texttt{match}\ g\ \text{`c'}\ s\ i = \text{i+1}}\ (\textbf{ch.1}) \qquad \frac{s[i] \neq \text{`c'}}{\texttt{match}\ g\ \text{`c'}\ s\ i = \texttt{nil}}\ (\textbf{ch.2})$$

Matching any character
$$\frac{i \leq |s|}{\texttt{match}\ g\ .\ s\ i = \text{i+1}}\ (\textbf{any.1}) \qquad \frac{i > |s|}{\texttt{match}\ g\ .\ s\ i = \texttt{nil}}\ (\textbf{any.2})$$

Not Predicate
$$\frac{\texttt{match}\ g\ p\ s\ i = \texttt{nil}}{\texttt{match}\ g\ !p\ s\ i = \text{i}}\ (\textbf{not.1}) \qquad \frac{\texttt{match}\ g\ p\ s\ i = \text{i+j}}{\texttt{match}\ g\ !p\ s\ i = \texttt{nil}}\ (\textbf{not.2})$$

And Predicate
$$\frac{\texttt{match}\ g\ p\ s\ i = \text{i+j}}{\texttt{match}\ g\ \&p\ s\ i = \text{i}}\ (\textbf{and.1}) \qquad \frac{\texttt{match}\ g\ p\ s\ i = \texttt{nil}}{\texttt{match}\ g\ \&p\ s\ i = \texttt{nil}}\ (\textbf{and.2})$$

Concatenation
$$\frac{\texttt{match}\ g\ p_1\ s\ i = \text{i+j} \qquad \texttt{match}\ g\ p_2\ s\ i+j = \text{i+j+k}}{\texttt{match}\ g\ p_1 p_2\ s\ i = \text{i+j+k}}\ (\textbf{con.1})$$

$$\frac{\texttt{match}\ g\ p_1\ s\ i = \text{i+j} \qquad \texttt{match}\ g\ p_2\ s\ i+j = \texttt{nil}}{\texttt{match}\ g\ p_1 p_2\ s\ i = \texttt{nil}}\ (\textbf{con.2}) \qquad \frac{\texttt{match}\ g\ p_1\ s\ i = \texttt{nil}}{\texttt{match}\ g\ p_1 p_2\ s\ i = \texttt{nil}}\ (\textbf{con.3})$$

Ordered Choice
$$\frac{\texttt{match}\ g\ p_1\ s\ i = \texttt{nil} \qquad \texttt{match}\ g\ p_2\ s\ i = \texttt{nil}}{\texttt{match}\ g\ p_1/p_2\ s\ i = \texttt{nil}}\ (\textbf{ord.1})$$

$$\frac{\texttt{match}\ g\ p_1\ s\ i = \text{i+j}}{\texttt{match}\ g\ p_1/p_2\ s\ i = \text{i+j}}\ (\textbf{ord.2}) \qquad \frac{\texttt{match}\ g\ p_1\ s\ i = \texttt{nil} \qquad \texttt{match}\ g\ p_2\ s\ i = \text{i+k}}{\texttt{match}\ g\ p_1/p_2\ s\ i = \text{i+k}}\ (\textbf{ord.3})$$

Repetition
$$\frac{\texttt{match}\ g\ p\ s\ i = \text{i+j} \qquad \texttt{match}\ g\ p*\ s\ i+j = \text{i+j+k}}{\texttt{match}\ g\ p*\ s\ i = \text{i+j+k}}\ (\textbf{rep.1}) \qquad \frac{\texttt{match}\ g\ p\ s\ i = \texttt{nil}}{\texttt{match}\ g\ p*\ s\ i = \text{i}}\ (\textbf{rep.2})$$

Variables
$$\frac{\texttt{match}\ g\ g(A_k)\ s\ i = \text{i+j}}{\texttt{match}\ g\ A_k\ s\ i = \text{i+j}}\ (\textbf{var.1}) \qquad \frac{\texttt{match}\ g\ g(A_k)\ s\ i = \texttt{nil}}{\texttt{match}\ g\ A_k\ s\ i = \texttt{nil}}\ (\textbf{var.2})$$

Closed Grammars
$$\frac{\texttt{match}\ g\ g(A_k)\ s\ i = \text{i+j}}{\texttt{match}\ g'\ (g, A_k)\ s\ i = \text{i+j}}\ (\textbf{cg.1}) \qquad \frac{\texttt{match}\ g\ g(A_k)\ s\ i = \texttt{nil}}{\texttt{match}\ g'\ (g, A_k)\ s\ i = \texttt{nil}}\ (\textbf{cg.2})$$

Figure 5: Operational Semantics of PEG Patterns with Grammars

define $\Pi'(g,i)$ to be

$$\begin{aligned}
&\Pi(g, i, g(A_1))\\
&\texttt{Return}\\
&\vdots\\
&\Pi(g, i + \sum_{j=1}^{k-1} \mid \Pi(g, x, A_j) \mid +1, g(A_k))\\
&\texttt{Return}\\
&\vdots\\
&\Pi(g, i + \sum_{j=1}^{n-1} \mid \Pi(g, x, A_j) \mid +1, g(A_n))\\
&\texttt{Return}
\end{aligned}$$

where $A_1, \ldots, A_k, \ldots, A_n$ are all variables where g has an image, ordered according to any arbitrary (but consistent) ordering. The x as the position for the sizes of subprograms means that the size of a program is independent on the position.

Given Π' we can define a function o : Grammar $\times\, V \rightarrow \mathcal{N}$ as $\sum_{j=1}^{k-1}(\mid \Pi(g,x,A_j) \mid +1)$ (where the 1 represents the `Return` instruction), which takes a grammar g and a variable A_k defined in g, and gives the position of the program derived from A_k in $\Pi'(g,i)$, relative to i.

With Π' and o, it is easy to extend Π to variables and closed grammars:

$$\Pi(g, i, A_k) \equiv \texttt{Call}\ o(g, A_k) - i$$

$$\begin{aligned}
\Pi(g', i, (g, A_k)) \equiv\ & \texttt{Call}\ o(g, A_k)\\
& \texttt{Jump}\ \mid \Pi'(g, x) \mid +1\\
& \Pi'(g, 2)
\end{aligned}$$

The 2 in $\Pi'(g, 2)$ keeps the invariant that all positions are relative to the first instruction of the closed grammar.

Now we will prove the correctness of our transformations, beginning with closed grammars. We first have to prove that:

$$\begin{aligned}
&\texttt{If match}\ g'\ (g, A_k)\ s\ i = \texttt{i+j then}\\
&\langle pc,\ i,\ e\rangle \xrightarrow{\Pi(g',l,(g,A_k))} \langle pc{+} \mid \Pi(g', l, (g, A_k)) \mid,\ i+j,\ e\rangle
\end{aligned}$$

This case is covered by the semantic rule ($cg.1$), and we have the following sequence of transitions:

$$\begin{aligned}
&\xrightarrow{Call} \langle pc + o(g, A_k),\ i,\ (pc+1) : e\rangle\\
&\xrightarrow{\Pi(g,o(g,A_k),A_k)} \langle pc + o(g, A_k) + \mid \Pi(g, o(g, A_k), A_k) \mid,\ i+j,\\
&\qquad (pc+1) : e\rangle\\
&\xrightarrow{Return} \langle pc+1,\ i+j,\ e\rangle\\
&\xrightarrow{Jump} \langle pc{+} \mid \Pi'(g, x) \mid +2,\ i+j,\ e\rangle
\end{aligned}$$

The second transition follows from the way we constructed the o function, with $pc + o(g, A_k)$ addressing the first instruction of $\Pi(g, o(g, A_k), A_k)$. Now we use our induction hypothesis, via the semantic rule ($cg.1$). The last transition uses the identity $\mid \Pi'(g, x) \mid +2 = \mid \Pi(g', l, (g, A_k)) \mid$. The proof for the failure case is analogous, via rule ($cg.2$).

For variables, we are only interested in the use of variables that are embedded in a closed grammar; other uses are left unspecified. This is not a problem in practice, as the semantics of `match` have that `match` $g\ A_k\ s\ i =$ `match` $g\ (g, A_k)\ s\ i$. We have to prove that:

$$\begin{aligned}
&\texttt{If match}\ g\ A_k\ s\ i = \texttt{i+j then}\\
&\langle pc,\ i,\ e\rangle \xrightarrow{\Pi(g,l,A_k)} \langle pc{+} \mid \Pi(g, l, A_k) \mid,\ i+j,\ e\rangle
\end{aligned}$$

This case is covered by the semantic rule ($var.1$), and the derivation for it is straightforward:

$$\begin{aligned}
&\xrightarrow{Call} \langle pc + o(g, A_k) - l,\ i,\ (pc+1) : e\rangle\\
&\xrightarrow{\Pi(g,o(g,A_k),A_k)} \langle pc + o(g, A_k) -\ l + \mid \Pi(g, o(g, A_k), A_k) \mid,\\
&\qquad i+j,\ (pc+1) : e\rangle\\
&\xrightarrow{Return} \langle pc+1,\ i+j,\ e\rangle
\end{aligned}$$

Again, given how o is defined, $o(g, A_k) - l$ is the offset from the current position to the first instruction of $\Pi(g, o(g, A_k), A_k)$. From there the proof follows by induction, via the semantic rule ($var.1$). The failure case is analogous, via rule ($var.2$).

6. RELATED WORK

The main influence of our parsing machine was Knuth's parsing machine [9]. But unlike Knuth's machine, where calls to non-terminals and backtrack entries are the same thing, our machine has a clear distinction between backtrack entries and calls. This distinction is essential to be able to compile patterns without rewriting them first.

One formal basis for pattern matching is regular expressions [14]. Pure regular expressions, however, have several limitations that make the description of some patterns difficult. Because of this, most pattern-matching tools make ad-hoc extensions to the original regular expressions (the extended regular expressions are popularly called regexes) losing their formal basis. As we did not want such a mixture of formal and ad-hoc features, we decided to use a PEG-based approach instead of regexes.

If we compare our PEG machine with other PEG-based approaches, a key difference is that most of them are more focused on parsing instead of pattern-matching. One well known approach is Packrat [1], which uses a memoization algorithm. Packrat is used in several parsing applications, such as *Rats!* [4], a parser generator based on PEGs. As we mentioned earlier, the linear space cost of Packrat makes it unsuitable for a pattern-matching application, where we have to deal with larger inputs than in parsing applications.

Another use of PEGs for parsing is described by [13], which implements a Java parser directly from the PEG definitions of the Java syntax. The author of [13] also points that during the implementation of the parser, he noticed much of the backtracking and repetitions were done when dealing with lexical structures, which suggests that some optimizations should be done to improve the performance of the parser (Rats!, for example, does not use PEG to define identifiers, keywords, and operators [13]). Our machine addressed this problem by defining instructions like `PartialCommit`, which is used in repetitions, and the head-fail optimization instructions, which avoid the backtracking when a pattern fails in its first character.

A PEG-based approach that is more similar to ours, in the sense that it also focus on pattern-matching, is OMeta [15].

Size of input	LPEG	Lex/Yacc	leg
470k	27	20	27
2460k	107	100	147
4950k	217	207	297

Table 2: Time for Parsing Arithmetic Expressions

OMeta extends the definition of PEGs to deal with arbitrarily-structured data instead of character strings. OMeta, however, does not define any new approach to parse the data, but relies on a recursive-descent parser without memoization.

Another related work is parser combinators [5], a popular approach for building recursive-descent parsers in the world of functional programming. By using parsing combinators it is possible to define a parser through the combination of several parsers, what is similar to our approach of combining programs. But parser combinators use unrestricted backtracking, with a large space and time cost, and implementations on strict languages is harder [5].

One drawback of parser combinators is that most implementations are not efficient in space or time [10]. Some approaches that address the performance issue depend mainly on lazy evaluation [10], which implies the use of memoization.

Our PEG machine, on the other hand, is easily implemented in a low level language, with a good performance. To show this point, we will compare the performance of LPEG with two other tools. One of the tools is `Lex/Yacc`, and the other one is `leg` [12], a PEG-based tool that generates a recursive-descent parser. Both tools produce a C program, which is statically compiled. LPEG, on the other hand, compiles its patterns dynamically, thus it is possible to modify a pattern or create a new one during runtime.

When using Lex/Yacc, we took the traditional approach, where Lex deals with the lexical part and Yacc with the syntactic one.

To eliminate the overhead of reading the input files, all tests read all the input before doing the parsing. To measure the parsing time we used the *clock* function in C, and the *os.clock* function in Lua. Each program was run three times against each input, and the mean time was considered.

The first parser that we defined was a parser of arithmetic expressions, whose PEG definition is shown below:

```
start    <- (exp '\n')+
exp      <- factor (factorOp factor)*
factor   <- term (termOp term)*
term     <- number space / '(' exp ')' space
factorOp <- [+-] space
termOp   <- [*/] space
termOp   <- [*/] space
number   <- '-'?[0-9]+
space    <- [\t ]*
```

Table 2 presents the parsing time (in milliseconds) of each tool considering different input sizes. As we can see, the performances of LPEG and Lex/Yacc were similar, and both were faster than leg.

The second parser was a parser of a list, where each element of the list was a number or another list, and the elements were separated by spaces. The PEG definition of the parser is shown below:

Size of input	LPEG	Lex/Yacc	leg
610k	20	20	27
3050k	93	113	150
6130k	187	227	303

Table 3: Time for Parsing Lists

Size of input	LPEG	Lex/Yacc	leg
620k	33	23	40
2440k	130	110	147
3680k	187	173	227

Table 4: Time for Parsing a Simple Language

```
start  <- (list '\n')+
list   <- '(' space (term (space term)*)? ')' space
term   <- list / number
number <- '-'?[0-9]+
space  <- [\t ]*
```

The performance of the parsers is presented in Table 3. For this simple definition of a list LPEG presented the best performance.

The last parser was a parser of a simple language (similar to Scheme), which is presented in [3]. We considered only the syntax that appears on page 71, plus the *if-then-else* on page 80. The grammar is presented below:

```
list    <- (program '\n')+
program <- exp
exp     <- number / if exp then exp else exp / id
    / primitive space '(' exp (',' space exp)* ')' space
primitive <- '+' / '-' / '*' / "add1" / "sub1"
number    <- [0-9]+ space
space     <- [ \t]*
space1    <- [ \t]+
letter    <- [a-zA-Z_]
if        <- "if" space1
then      <- "then" space1
else      <- "else" space1
reserved  <- ("if" / "then" / "else" / "add1" / "sub1")
             !(letter / number)
id        <- !reserved  letter (letter / [0-9])*  space
```

The performance of each parser can be seen in Table 4. In this case, Lex/Yacc had the best performance, followed by LPEG and leg, respectively.

From the benchmarks, we can see that LPEG and Lex/Yacc have a very similar performance. We consider LPEG as a better choice to do simple tasks. LPEG is also a simpler tool than Lex/Yacc, and should be easier to use it.

7. CONCLUSIONS

We presented a new approach for implementing PEGs, by compiling them to programs for a virtual parsing machine. Our motivation was to use PEGs as a formal basis for pattern-matching, which needed an implementation with good space *and* time costs.

We gave the operational semantics of our parsing machine, and a transformation from patterns to programs for the machine, along with the corresponding proof of correctness.

In our implementation, this transformation is a compilation process that is done during runtime. The compiler builds more complex programs from simpler ones, following rules specific to each PEG operator.

The parsing machine's performance is competitive with the performance of popular parsing tools.

As the machine has a very simple execution model, somewhat similar to a real CPU, it seems a perfect candidate for a Just-In-Time (JIT) compiler. A JIT compiler should increase considerably the performance of our implementation.

Another future work is to extend the execution model of our machine to handle infinite loops, and to make the use of patterns with left recursion possible (without rewriting the patterns).

One point that was not discussed here was the use of semantic actions during the matching process. They don't affect the matching process but are very useful in practice. The use of semantic actions is presented in [7]. We will extend the formal model of the machine to include actions.

8. ACKNOWLEDGMENTS

We would like to thank Fabio Mascarenhas for many suggestions and useful comments about early versions of this paper.

9. REFERENCES

[1] B. Ford. Packrat parsing:: simple, powerful, lazy, linear time, functional pearl. *SIGPLAN Not.*, 37(9):36–47, 2002.

[2] B. Ford. 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, 2004. ACM.

[3] D. P. Friedman, C. T. Haynes, and M. Wand. *Essentials of programming languages (2nd ed.)*. Massachusetts Institute of Technology, Cambridge, MA, USA, 2001.

[4] R. Grimm. Better extensibility through modular syntax. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 38–51, New York, NY, USA, 2006. ACM.

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

[6] R. Ierusalimschy. *Programming in Lua, Second Edition*. Lua.Org, 2006.

[7] R. Ierusalimschy. A Text Pattern-Matching Tool based on Parsing Expression Grammars. *Software - Practice and Experience*, 2008 (to appear).

[8] R. Ierusalimschy, L. H. de Figueiredo, and W. C. Filho. Lua - an extensible extension language. *Software - Practice and Experience*, 26(6):635–652, 1996.

[9] D. E. Knuth. Top-down syntax analysis. *Acta Inf.*, 1:79–110, 1971.

[10] D. J. P. Leijen and H. J. M. 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.

[11] LPEG: Parsing Expression Grammars For Lua. Available at http://www.inf.puc-rio.br/˜roberto/lpeg, 2008. Visited on April 2008.

[12] peg/leg - recursive-descent parser generators for C. Available at http://piumarta.com/software/peg/, 2008. Visited on June 2008.

[13] R. R. Redziejowski. Parsing expression grammar as a primitive recursive-descent parser with backtracking. *Fundamenta Informaticae*, 3–4(79):513–524, 2007.

[14] K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968.

[15] A. Warth and I. Piumarta. 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, 2007. ACM.

APPENDIX

A. EXTENDED TRANSFORMATION FROM PATTERNS TO PROGRAMS

Matching a Character

$$\Pi(g, i, \text{`c'}) \equiv \texttt{Char}\ c$$

Concatenation

$$\Pi(g, i, p_1\, p_2) \equiv \Pi(g, i, p_1)\ \Pi(g, i + |\Pi(g, x, p_1)|, p_2)$$

Ordered Choice

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

Not Predicate

$$\begin{aligned}\Pi(g, i, !p) \equiv\ & \texttt{Choice}\ |\Pi(g, x, p)| + 2\\ & \Pi(g, i+1, p)\\ & \texttt{FailTwice}\end{aligned}$$

Repetition

$$\begin{aligned}\Pi(g, i, p*) \equiv\ & \texttt{Choice}\ |\Pi(g, x, p)| + 2\\ & \Pi(g, i+1, p)\\ & \texttt{PartialCommit}\ - |\Pi(g, x, p)|\end{aligned}$$

Variables

$$\Pi(g, i, A_k) \equiv \texttt{Call}\ o(g, A_k) - i$$

Closed Grammars

$$\begin{aligned}\Pi(g', i, (g, A_k)) \equiv\ & \texttt{Call}\ o(g, A_k)\\ & \texttt{Jump}\ \mid \Pi'(g, x) \mid +1\\ & \Pi'(g, 2)\end{aligned}$$