Bachelor thesis
Computer Science

Radboud University

An efficient parsing machine for PEGs

Author:
Jos Craaijo
s4481674

First supervisor/assessor:
Prof. dr. Herman Geuvers
`herman@cs.ru.nl`

Second assessor:
Dr. Freek Wiedijk
`freek@cs.ru.nl`

January 11, 2018

Abstract

Popular PEG matching algorithms make use of memoization through PACKRAT which results in linear space complexity, making the algorithms unsuitable as an alternative for regular experssions. We introduce a novel Parsing Machine that is optimized for JIT compilation, and show that our implementation of this machine has run time performance similar to that of regular expression matching libraries. We also introduce a tool, `peg-match`, which can be used as a replacement for grep.

Contents

Chapter 1

Introduction

Regular expressions, first introduced by Kleene [9], are often used for pattern matching and input validation. However, regular languages did not provide enough flexibility and expressive power, which lead many libraries to include their own extensions. Surprisingly, almost every "regular" expression library is capable of matching non-regular languages because of these extensions. Few of these extensions have a formal basis, which means that it's difficult determine the complexity of a regular expression.

Additionally, regular expressions can quickly turn into very complex patterns, for example a 6000-character regular expression to match vaild e-mail addresses [1]. The fact that regular expressions offer no way to re-use expressions means that there is no way to logically group parts of the expression or re-use components multiple times.

RFCs often define grammars for formats like IPv6 address notation, domain names or e-mail addresses in augmented BNF form [5]. These grammars are difficult or even impossible to translate to regular expressions because of a lack of recursion in regular expressions.

One alternative to regular expressions are Parsing Expression Grammars (PEGs). First introduced by Ford in 2004 [7], PEGs eliminate all problems we describe above:

1. PEGs are more expressive, so no extensions are needed
2. PEGs support both expression re-use and recursion, which allows for more readable expressions and easy translation from RFC grammars to PEGs.
3. PEGs have a formal basis, making it easy to determine complexity of expressions

PEGs are generally matched using an algorithm called PACKRAT, also introduced by Ford in 2002 [6]. PACKRAT can match any formal language representable by a PEG in linear time, at the cost of memory usage linear to the size of the input because of memoization. The memory usage poses a problem when using PEGs as a replacement for regular expressions, since we need to be able to match simple expressions on very large inputs. If we're matching a 5Gb file, we cannot afford to use another multiple of 5Gb to keep track of the memoization.

One alternative to PACKRAT, introduced by Mereidos and Ierusalimschy in 2008 [11], proposes using a 'parsing machine', where PEGs are translated to a small instruction set that can be executed on the parsing machine. This parsing machine uses very little memory, at the cost of having exponential run-time complexity.

One issue with this parsing machine is that it pushes code addresses on the stack, and jumps to those addresses at a later point in the execution. This is problematic when using Just-In-Time (JIT) compilation to compile the instruction set to machine code, as languages like C# (using the .NET runtime) and Java (using the JVM) enforce a certain degree of memory safety that does not allow jumping to arbitrary addresses on the stack.

In this paper we introduce a new parsing machine that can be efficiently compiled to machine code at runtime, which can be used as a replacement for regular expressions. Our implementation does not jump to addresses on the stack, which means that it does not suffer from the problem described above. We show that our implementation can use just-in-time compilation by building a C# implementation that leverages the .NET JIT compiler to compile PEGs at runtime. We also show that this implementation has performance similar to commonly-used regular expression libraries, like PCRE2 and RE2.

Chapter 2

Preliminaries

2.1 Parsing expression grammars

Parsing expression grammars (PEGs) are a form of formal grammars used to describe formal languages.

Definition 1. A PEG is a tuple $G = (\Sigma, A, R, A_s)$, where:

1. Σ is a finite alphabet over which the grammar will operate
2. A is a set of non-terminal symbols
3. R is a finite function $A \to e$, i.e. it maps a non-terminal to a parsing expression
4. A_s is the start rule

Note that some definitions of PEGs use a start expression instead of a start rule. These definitions are interchangable.

Given Σ and A, the set of parsing expressions $\mathcal{E}$ is inductively defined as follows:

1. Atomic expressions:
   (a) Terminal symbol $c \in \mathcal{E}$, where $c \in \Sigma$, which matches a single character c.
   (b) Non-terminal $A_x \in \mathcal{E}$, which matches if the non-terminal A_x matches.
   (c) Empty string $\varepsilon \in \mathcal{E}$, matching 0 characters. This will always succeed.
2. Given the expressions $e \in \mathcal{E}$, $e_1 \in \mathcal{E}$ and $e_2 \in \mathcal{E}$, a new parsing expression can be constructed by using one of the following operators:
   (a) Concatenation: $e_1e_2 \in \mathcal{E}$, which matches only if e_1 matches, and e_2 matches after e_1.
   (b) Prioritized choice: $e_1/e_2 \in \mathcal{E}$, which first attempts to match e_1. If matching e_1 succeeds, the entire expression succeeds. Only if e_1 fails will we attempt to match e_2.
   (c) Zero-or-more: $e* \in \mathcal{E}$, which matches any number of es in the input string.
   (d) Not: $!e \in \mathcal{E}$, which matches only if e does not match. This operator does not consume any characters in the input string.

The operator precedence is as follows (from strongest to weakest):

1. Suffix operators: zero or more (and syntactic sugar: one or more, optional)
2. Prefix operators: not (and syntactic sugar: and)
3. Concatenation
4. Prioritized choice

2.1.1 Semantics

We define the semantics of PEGs using `match` $g\ p\ s\ i$, where:

1. g is a grammar
2. p is an expression
3. s is the input string
4. i is the position in the input-string (0-based)

`match` will return any natural number greater than or equal to i if p matched at position i in s, or `nil` if matching failed.

$$\text{ch.1}\ \frac{s[i] = c \qquad i < |s|}{\texttt{match}\ g\ \text{'c'}\ s\ i = i + 1}$$

$$\text{ch.2}\ \frac{s[i] \neq c \vee i \geq |s|}{\texttt{match}\ g\ \text{'c'}\ s\ i = \texttt{nil}}$$

$$\text{any.1}\ \frac{i < |s|}{\texttt{match}\ g\ .\ s\ i = i + 1}$$

$$\text{any.2}\ \frac{i \geq |s|}{\texttt{match}\ g\ .\ s\ i = \texttt{nil}}$$

$$\text{not.1}\ \frac{\texttt{match}\ g\ p\ s\ i = \texttt{nil}}{\texttt{match}\ g\ !p\ s\ i = i}$$

$$\text{not.2}\ \frac{\texttt{match}\ g\ p\ s\ i = i + j}{\texttt{match}\ g\ !p\ s\ i = \texttt{nil}}$$

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

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

$$\text{con.3}\ \frac{\texttt{match}\ g\ p_1\ s\ i = \texttt{nil}}{\texttt{match}\ g\ p_1 p_2\ s\ i = \texttt{nil}}$$

$$\text{ord.1}\ \frac{\texttt{match}\ g\ p_1\ \ s\ i = \texttt{nil} \qquad \texttt{match}\ g\ p_2\ \ s\ i = \texttt{nil}}{\texttt{match}\ g\ p_1/p_2\ \ s\ i = \texttt{nil}}$$

$$\text{ord.2}\ \frac{\texttt{match}\ g\ p_1\ \ s\ i = i + j}{\texttt{match}\ g\ p_1/p_2\ \ s\ i = i + j}$$

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

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

$$\text{rep.2}\ \frac{\texttt{match}\ g\ p\ s\ i = \texttt{nil}}{\texttt{match}\ g\ p*\ \ s\ i = i}$$

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

$$\text{var.2}\ \frac{\texttt{match}\ g\ g(A_k)\ \ s\ i = \texttt{nil}}{\texttt{match}\ g\ A_k\ \ s\ i = \texttt{nil}}$$

Example 2. `match g ('T' 'o' 'm') s i` will either match the entire word 'Tom' from the input string s and return $i + 3$, or it will fail and return `nil`.

Example 3. Prioritized choices are fundamentally different from the choice operator (|) in regular expressions. For example, `match g (('a' / 'a' 'a') 'c') s i` will never match the input string `'aac'`, since the match of the first choice inside the prioritized choice, `'a'`, succeeds. The concatenation does, unlike the | operator in regular expressions, not backtrack to try and match the other choice. If we change the expression to `'a' 'c' / 'a' 'a' 'c'`, it does match the word `'aac'`, since matching `'c'` will now fail inside the prioritized choice.

2.1.2 Syntactic sugar

There are also five additional operators that can be defined in terms of the expressions above:

1. Any character: ., which is syntactic sugar for $a_1/a_2/.../a_n$, where $\Sigma = a_1, a_2, ..., a_n$
2. A character class: $[a - z]$, which is syntactic sugar for $a/b/.../z$. This assumes that the alphabet Σ has some logical ordering
3. One-or-more: $e+$, which is syntactic sugar for $ee*$
4. Optional: $e?$, which is syntactic sugar for e/ε
5. And: $\&e$, which is syntactic sugar for $!!e$

2.1.3 Well-formedness

Definition 4. A well-formed grammar is a grammar that contains no directly or mutually left-recursive rules. We define this inductively as follows [7]:

For any grammar $G = (\Sigma, A, R, A_s)$, all atomic expressions are well-formed:

1. ε is well-formed
2. c is well-formed
3. A is well-formed if $R(A)$ is well-formed

The following rules state when composed expressions are also well-formed:

1. e_1e_2 is well-formed if e_1 is well-formed and e_1 will always match at least one character
2. e_1e_2 is well-formed if e_1 is well-formed and if e_1 and e_2 are well-formed and e_2 will always match at least one character
3. e_1/e_2 is well-formed if e_1 is well-formed and e_2 is well-formed
4. $e*$ is well-formed if e is well-formed and e will always match at least one character
5. $!e$ is well-formed if e is well-formed

A grammar G is well-formed if all rules in the gramar are well-formed.

Example 5. `A <- A / 'a' A` is not well-formed, because it allows for direct left-recursion of the non-terminal 'A'.

Example 6. `A <- B / 'a' A; B <- A` is not well-formed, because it is mutually left-recursive through the non-terminal 'B'.

Example 7. `A <- 'a' A 'a' /` ε is well-formed, because it will always read at least one terminal character before recursing, which means that it's not left-recursive.

From now on, we will assume all PEGs are well-formed unless mentioned otherwise.

2.2 PACKRAT

PACKRAT parsing introduces memoization to the matching process. While matching, we maintain a list of matching outcomes of non-terminals. When matching a non-terminal A_x at position p, we first check if we've ever matched A_x at position p before. If we have, we can look up the result in the list of outcomes, and immediately return that. If not, we match A_x and add its outcome to the list.

Using PACKRAT, any PEG grammar can be matched in linear time. However, due to the memoization PACKRAT will end up using linear space.

Chapter 3

The Parsing Machine

3.1 Extended PEG grammar

We introduce a number of extensions to the PEG grammar introduced by Ford. Our aim for these extensions is to make PEG definitions more readable and less repetitive for expressions similar to those commonly used in regexes. All of our extensions were added in the process of building the standard library for our tool `peg-match`, and are based on those use-cases.

In order to not cause the same issues that regular expression libraries did by introducing extensions without formal definitions, we have been careful to ensure that all of our extensions are purely syntactic sugar. Therefore, the expressiveness of the extended PEGs remains the same as the original expressiveness, and each of the extensions below could be translated back to the original PEGs as defined by Ford [7].

3.1.1 Fixed repeat

First, we introduce a shorter notation for repeatedly matching an expression, with an upper and lower limit.

Definition 8. We define `r^k` to mean `r r r ...  r r r` where the number of `r`'s is equal to k. We define `r^k..j` to mean `r^j / r^j-1 / ...  / r^k` where $j >= k$.

Example 9. One example of where these definitions come in helpful, is when defining a grammar for IPv6 addresses. Each IPv6 "hextet" consists of 4 hexadecimal characters. Without the `^` operator, we need to define a "hextet" as `[0-9a-fA-F] [0-9a-fA-F] [0-9a-fA-F] [0-9a-fA-F]`. This can now be simplified to `[0-9a-fA-F]^4`.

Additionally, one of the possible forms of an IPv6 address is a double colon (::) followed by one or more hextets, separated by a single colon. Using the fixed repeat operator `^` we can define this as follows: `':'  (':'  Hextet)^1..7`, which is shorter and easier to read than using a prioritized choice with 7 cases.

3.1.2 Parameterized non-terminals

Parameterized non-terminals simplify the process of combining existing PEGs. While exising PEGs can already be combined by referencing a non-terminal of a different grammar, the included pattern will still be a single, closed unit. This limits extensibility of grammars.

Example 10. Consider the following regular expression `[a-z]+ing`.

Translating this pattern to a PEG is non-trivial, since PEGs will not backtrack. A naive translation to the PEG expression `[a-z]+'ing'` would mean that `[a-z]+` will always match the entire word, leaving 'ing' to always fail. A correct translation would be `(!(!(!'ing') !([a-z] (!'ing' [a-z])* 'ing')) [a-z])* 'ing'`. If we want to (re-)use the same expression structure for matching all words ending in 'ne', we would have to duplicate the expression. This leads to duplicate code, which is undesired.

Parameterized non-terminals solve this problem by allowing for textual substitution of patterns.

Example 11. We might write a generic "until" non-terminal like this:

```
Until<Item, Ending> <- (!(!(!End) !(Repeat (!End Repeat)* End)) Repeat)* End
```

We can now instantiate a particular version of the non-terminal like this:

```
A <- [a-z]
B <- 'ing'
EndsWithIng <- Until<A, B>
```

3.1.3 Inline non-terminals

With parameterized non-terminals, it's counter-intuitive to declare the parameters on a separate line. In example 11 above, `A` and `B` are separated from their context, making it harder to understand the expression. To solve this, we introduce an extension that allows inline definition of non-terminals inside the list of non-terminals of a parameterized pattern: `\(e)`, where `e` is a PEG expression.

Example 12. Our example above can now be simplified to just one line:
`EndsWithIng <- Until<\([a-z]), \('ing')>`
This is merely syntactic sugar for the grammar defined in Example 11.

When reading the grammar, having the non-terminals directly inside the parameterized non-terminal means that you don't need to search through all the rules to find the non-terminals that are being used.

3.1.4 Namespaces

Lastly, since we are introducing a standard library with our tool `peg-match` (see Section 6.1), adding all pre-defined non-terminals to the global namespace might cause naming conficts. It is also not desirable to expose all defined non-terminals publicly. For example, we might need to define an "internal" pattern containing all reseved characters in an URI. We don't want this pattern to be used by anyone else, since we want to be able to change this expression if we find a more efficient way to define it. This means that we cannot have anyone else directly using the expression.

To solve this, we introduce namespaced non-terminals. As far as the PEG grammar is concerned, this change merely allows '::' to appear inside the name of a non-terminal:

Example 13. Our standard library includes a `String` namespace containing common parameterized non-terminals for strings, like `String::Until` (which is similar to the expression defined in Example 11). Our example can now be written as:
`EndsWithIng <- String::Until<\([a-z]), \('ing')>`

Secondly, we introduce the keyword `export`. This keyword can be placed in front of a non-terminal definition to indicate that it should be accessible from other namespaces.

Namespace consistency and non-terminal accessibility is not enforced in the PEG grammar itself. In implementations, It should be enforced when resolving the non-terminal references and substituting the parmeters of the parameterized non-terminals.

3.2 The Parsing Machine

3.2.1 Programs

Before diving into semantics and program generation, we will describe the general design of our parsing machine. The parsing machine shares many similarities with a real computer. It runs one 'program' at a time, where a program C is defined as follows:

Definition 14. $C = (A_s, m)$, where A_s is the starting non-terminal, and m is a finite function $A \to$ InstructionList, where A are non-terminals. These programs are generated by a compiler, which we will describe in Section 4.1.

An InstructionList is a sequence of instructions, which are defined in Section 3.3.

In practice, to execute a program C on the parsing machine, instructions in the program are executed on a state $t = \langle (A_x, pc, V) : e, i \rangle$, where:

- (A_x, pc, V) is a single stack frame, at the top of the stack
  - A_x is the non-terminal that is being executed
  - pc is the program counter, which is the index of the next instruction that will be executed
  - V is a finite function $Var \to \mathbb{N}$ which the program uses to store string indices
- e is the tail of the stack, which may be empty or it may consist of one or more stack frames.
- i keeps track of the position in the string

Each step, an instruction is fetched from $m[A_x][pc]$, and an associated state transformation is applied to the state. This process is repeated until the program terminates.

The parsing machine uses a stack in the same way a computer program would: the stack consists of one or more 'stack frames', where each stack frame stores variables v and a program counter pc. When the stack is empty, the program terminates. If $i \geq 0$, matching was succesful. If $i = \texttt{nil}$, matching failed. Non-terminals are sometimes also called 'variables'. To avoid confusion we will always use 'variable' to refer to a variable in a stack frame, and 'non-terminal' to refer to non-terminals.

When executing a program C, the inital state will be $\langle (A_s, 0, \emptyset), 0 \rangle$, i.e. the matching starts by executing the first instruction of the non-terminal A_s at position 0 in the string, without any variables. An instruction can modify the state, producing a new state. This is denoted using an arrow:

$$\langle (A_x, pc, V) : e, i \rangle \xrightarrow{\texttt{Instruction}\ \text{parameters}} \langle (A_x, pc', V') : e, i' \rangle \tag{3.1}$$

Note that the compiled program C is not included in the state, since it cannot be mutated by an instruction.

3.2.2 Compilation

PEG grammars are compiled into programs using one of three functions:

1. $\Pi_G(G)$ compiles an entire grammar. It takes a grammar G as input, and produces a mapping from patterns to sequences of instructions as output.

2. $\Pi_{NT}(A_x)$ compiles a single non-terminal. It takes a pattern p as input, and produces a sequence of instructions as output.

3. $\Pi_P(p, f)$ recursively compiles an expression. It takes an expression p and a fail label f as input, and produces a sequence of instructions as output.

For PEGs, a compiled program $C = (A_s, \Pi_G(G))$, where A_s is the starting pattern of grammar G.

3.3 Operational Semantics

3.3.1 Auxiliary definitions

Since we want to be able to easily insert and re-order instructions, we cannot use relative or absolute jumping offsets, as this would mean that we need to re-calculate the offset of every instruction each time we insert or remove an instruction. Instead, we mark the jump targets using a special instruction `MarkLabel` L. This instruction does not modify the state, and is only used by other instructions to determine the next value of the program counter *pc*. These `MarkLabel` instructions do not impact performance, as they are translated to offsets when compiling the PEG to machine code.

Definition 15. We define indexOf to be a function $(A \to L) \to \mathbb{N}$, that returns the position of any occurrence of a `MarkLabel` instruction with the matching label:
indexOf$(A_x, l) = i$ such that $m[A_x][i] = \texttt{MarkLabel}\ l$

3.3.2 Semantics

$$\langle (A_x, pc, V) : e, i\rangle \xrightarrow{\texttt{Advance}\ N} \langle (A_x, pc+1, V) : e, i+N\rangle \tag{3.2}$$

$$\begin{aligned}&\langle (A_x, pc, V) : e, i\rangle \xrightarrow{\texttt{BoundsCheck}\ N\ f} \langle (A_x, pc', V) : e, i\rangle \\ &\qquad \text{where } pc' = \begin{cases} pc+1 \text{ if } i+N < |s| \\ \text{indexOf}(f, p) \text{ otherwise}\end{cases}\end{aligned} \tag{3.3}$$

$$\begin{aligned}&\langle (A_x, pc, V) : e, i\rangle \xrightarrow{\texttt{Char}\ C\ N\ f} \langle (A_x, pc', V) : e, i\rangle \\ &\qquad \text{where } pc' = \begin{cases} pc+1 \text{ if } C = s[i+N] \\ \text{indexOf}(f, p) \text{ otherwise}\end{cases}\end{aligned} \tag{3.4}$$

$$\langle (A_x, pc, V) : e, i\rangle \xrightarrow{\texttt{Jump}\ L} \langle (A_x, \text{indexOf}(L, p), V) : e, i\rangle \tag{3.5}$$

$$\langle (A_x, pc, V) : e, i\rangle \xrightarrow{\texttt{MarkLabel}\ L} \langle (A_x, pc+1, V) : e, i\rangle \tag{3.6}$$

$$\langle (A_x, pc, V) : e, i\rangle \xrightarrow{\texttt{StorePosition}\ v} \langle (A_x, pc+1, V[v := i]) : e, i\rangle \tag{3.7}$$

$$\langle (A_x, pc, V) : e, i\rangle \xrightarrow{\texttt{RestorePosition}\ v} \langle (A_x, pc+1, V) : e, V[v]\rangle \tag{3.8}$$

$$\langle (A_x, pc, V) : e, i\rangle \xrightarrow{\texttt{Call}\ A_m} \langle (A_m, 0, \emptyset) : (A_x, pc+1, V) : e, i\rangle \tag{3.9}$$

$$\begin{aligned}&\langle (A_x, pc, V) : e, i\rangle \xrightarrow{\texttt{EndCall}\ f} \langle (A_x, pc', V) : e, i\rangle \\ &\qquad \text{where } pc' = \begin{cases} pc+1 \text{ if } i \neq \texttt{nil} \\ \text{indexOf}(f, p) \text{ otherwise}\end{cases}\end{aligned} \tag{3.10}$$

$$\begin{aligned}&\langle (A_x, pc, V) : e, i\rangle \xrightarrow{\texttt{Return}\ B} \langle e, i'\rangle \\ &\qquad \text{where } i' = \begin{cases} i \text{ if } B = 1 \\ \texttt{nil} \text{ otherwise}\end{cases}\end{aligned} \tag{3.11}$$

Chapter 4

Compiling PEGs

4.1 Program generation

4.1.1 The Π-function

Now that the semantics of the instructions are defined, we can use these instructions to transform grammars and patterns into sequences of instructions that can be executed by the parsing machine.

In the following definitions, v is always a fresh variable (i.e. a variable that has not already been used anywhere in the sequence of instructions), and L, L_1 or L_2 are fresh labels.

Definition 16. We define Π_G, which compiles an entire grammar, as follows:

$$\Pi_G(G) = \{A_x \Rightarrow \Pi_{NT}(A_x) \mid A_x \in G\} \tag{4.1}$$

Definition 17. We define Π_{NT}, which compiles a single non-terminal, as follows:

$$\begin{aligned}\Pi_{NT}(A_x) \equiv\ & \Pi_P(R(A_x), L) \\ & \texttt{Return } 1 \\ & \texttt{MarkLabel } L \\ & \texttt{Return } 0\end{aligned} \tag{4.2}$$

Definition 18. We define Π_P, which recursively compiles an expression, as follows:

$$\begin{aligned}
\Pi_P(c, f) \equiv\ & \texttt{BoundsCheck}\ 0\ f \\
& \texttt{Char}\ c\ 0\ f \\
& \texttt{Advance}\ 1 \\[1em]
\Pi_P(., f) \equiv\ & \texttt{BoundsCheck}\ 0\ f \\
& \texttt{Advance}\ 1 \\[1em]
\Pi_P(p_1 p_2, f) \equiv\ & \Pi_P(p_1, f) \\
& \Pi_P(p_2, f) \\[1em]
\Pi_P(p_1/p_2, f) \equiv\ & \texttt{StorePosition}\ v \\
& \Pi_P(p_1, L_1) \\
& \texttt{Jump}\ L_2 \\
& \texttt{MarkLabel}\ L_1 \\
& \texttt{RestorePosition}\ v\ 0 \\
& \Pi_P(p_2, f) \\
& \texttt{MarkLabel}\ L_2 \qquad (4.3) \\[1em]
\Pi_P(!p_1, f) \equiv\ & \texttt{StorePosition}\ v \\
& \Pi_P(p_1, L_1) \\
& \texttt{Jump}\ f \\
& \texttt{MarkLabel}\ L_1 \\
& \texttt{RestorePosition}\ v\ 0 \\[1em]
\Pi_P(p*, f) \equiv\ & \texttt{MarkLabel}\ L_1 \\
& \texttt{StorePosition}\ v \\
& \Pi_P(p, L_2) \\
& \texttt{Jump}\ L_1 \\
& \texttt{MarkLabel}\ L_2 \\
& \texttt{RestorePosition}\ v\ 0 \\[1em]
\Pi_P(A_x, f) \equiv\ & \texttt{Call}\ A_x \\
& \texttt{EndCall}\ f
\end{aligned}$$

Note that for repetition ($p*$), the same variable may be updated multiple times.

Example 19. Consider the expression `'ab'* 'c'`. If we match this on the input string 'abababc', the parsing machine will write to the variable v used in the repeat 4 times: at input position $0, 2, 4$ and 6. At position 0 we add the variable to the collection, and the next 3 times we update the value of the variable.

We are allowed to overwrite the variable, since PEG backtracking will not backtrack more than one iteration of a repetition, unlike regular expressions.

4.2 Correctness

Definition 20. We define $|\Pi_P(r, f)|$ to be the length of the `InstructionList`, i.e. the number of instructions in the instruction list. This is equivalent to the number of lines, when writing each instruction on its own line, as we have done in the definitions of the Π functions.

Note that we will not prove the correctness of any of the syntactic sugar operators, as they can all be expressed in terms of the operators that we do prove to be correct.

We start by proving the correctness of the program generation for patterns, by providing proof that for any grammar g, for any non-terminal A_x, for any position i in any string s, any expression r and any fail label f the following two implications hold:

Theorem 21.

$$\begin{aligned}
&\text{If } \texttt{match}\ g\ r\ s\ i = i + j \\
&\text{then } \langle (A_x, pc, V) : e, i\rangle \xrightarrow{\Pi_P(A_x,f)} \langle (A_x, pc + |\Pi_P(r, f)|, V') : e, i + j\rangle \\
&\text{where } \forall_{v\in V}[v \notin \Pi_P(A_x, f) \Rightarrow V[v] = V'[v]] \\
&\text{If } \texttt{match}\ g\ r\ s\ i = \texttt{nil} \\
&\text{then } \langle (A_x, pc, V) : e, i\rangle \xrightarrow{\Pi_P(A_x,f)} \langle (A_x, \text{indexOf}(r, f), V') : e, z\rangle \\
&\text{where } \forall_{v\in V}[v \notin \Pi_P(A_x, f) \Rightarrow V[v] = V'[v]] \text{ and } z \in \mathbb{Z}
\end{aligned} \tag{4.4}$$

The constraint on V specifies that only the variables that appear in the InstructionList can be modified by those instructions. We will use this later on, in combination with the fact that Π uses a fresh variable each time one is needed, to infer that certain variables are not changed.

Proof is done by induction over the derivation of `match` $g\ r\ s\ i$.

4.2.1 Base cases

Matching a character

First the case where `match` $g\ c\ s\ i = i+1$ (ch.1). We know that $s[i] =$ 'c' and consequently also that $i < |s|$. This matches the behaviour of the parsing machine, which can be seen by applying the state transitions:

$$\begin{aligned}
\langle (A_x, pc, V) : e, i\rangle &\xrightarrow{\texttt{BoundsCheck}\ 0\ f} \text{(using } i < |s|) \\
\langle (A_x, pc + 1, V) : e, i\rangle &\xrightarrow{\texttt{Char}\ c\ 0\ f} \text{(using } s[i] = \text{'c')} \\
\langle (A_x, pc + 2, V) : e, i\rangle &\xrightarrow{\texttt{Advance}\ 1} \\
\langle (A_x, pc + 3, V) : e, i + 1\rangle &
\end{aligned} \tag{4.5}$$

In the fail case, where `match` $g\ c\ s\ i =$ `nil` (ch.2), either $i \geq |s|$ or $s[i] \neq' c'$. If $i \geq |s|$, the parsing machine will execute the following sequence of transitions:

$$\begin{aligned}
\langle (A_x, pc, V) : e, i\rangle &\xrightarrow{\texttt{BoundsCheck}\ 0\ f} \\
\langle (A_x, \text{indexOf}(A_x, f), V) : e, i\rangle &
\end{aligned} \tag{4.6}$$

If $s[i] \neq$ 'c', the associated sequence of transitions is:

$$\begin{aligned}\langle (A_x, pc, V) : e, i\rangle &\xrightarrow{\texttt{BoundsCheck}\ 0\ f} \\ \langle (A_x, pc+1, V) : e, i\rangle &\xrightarrow{\texttt{Char}\ c\ 0\ f} \\ \langle (A_x, \text{indexOf}(A_x, f), V) : e, i\rangle &\end{aligned} \tag{4.7}$$

This proves the correctness of matching a single character.

Matching any character

The proof for matching any character is very similar. In the case where `match` g . s $i = i{+}1$ (any.1), we know that $i < |s|$. Therefore, the sequence of transitions looks like this:

$$\begin{aligned}\langle (A_x, pc, V) : e, i\rangle &\xrightarrow{\texttt{BoundsCheck}\ 0\ f} \text{ (using } i < |s|) \\ \langle (A_x, pc+1, V) : e, i\rangle &\xrightarrow{\texttt{Advance}\ 1} \\ \langle (A_x, pc+2, V) : e, i+1\rangle &\end{aligned} \tag{4.8}$$

In the fail case, where `match` g . s i = `nil` (any.2), we know that $i \geq |s|$. The parsing machine will execute the following sequence of transitions:

$$\begin{aligned}\langle (A_x, pc, V) : e, i\rangle &\xrightarrow{\texttt{BoundsCheck}\ 0\ f} \\ \langle (A_x, \text{indexOf}(A_x, f), V) : e, i\rangle &\end{aligned} \tag{4.9}$$

This proves the correctness of matching any character.

4.2.2 Inductive cases

Not Predicate

For the not predicate there are once again two rules. In the first case, not.1, we know that `match` g r s i = `nil`. Therefore, the sequence of transitions looks like this:

$$\begin{aligned}\langle (A_x, pc, V) : e, i\rangle &\xrightarrow{\texttt{StorePosition}\ v} \\ \langle (A_x, pc+1, V[V \Rightarrow i]) : e, i\rangle &\xrightarrow{\Pi_P(r, L_1)} \text{ (using the IH)} \\ \langle (A_x, pc+1+|\Pi_P(r,f)|, V'[v \Rightarrow i]) : e, z\rangle &\xrightarrow{\texttt{MarkLabel}\ L_1} \\ \langle (A_x, pc+2+|\Pi_P(r,f)|, V'[v \Rightarrow i]) : e, z\rangle &\xrightarrow{\texttt{RestorePosition}\ v0} \\ \langle (A_x, pc+3+|\Pi_P(r,f)|, V'[v \Rightarrow i]) : e, V[v]+0\rangle &\end{aligned} \tag{4.10}$$

Note that:

1. $pc+1+|\Pi_P(r,f)| = \text{indexOf}(A_x, L_1)$, since all labels are fresh and never re-used in a `MarkLabel` L instruction.
2. Variable v will not be modified by $\Pi_P(A_x, L_1)$, which follows from the IH. This means that $V[v] + 0 = i$.

In the second case, not.2 where `match` $g\ r\ s\ i = i + j$, the parsing machine will execute the following sequence of transitions:

$$\begin{aligned}
\langle (A_x, pc, V) : e, i\rangle &\xrightarrow{\texttt{StorePosition}\ v} \\
\langle (A_x, pc+1, V[v \Rightarrow i]) : e, i\rangle &\xrightarrow{\Pi_P(r, L_1)} \text{ (using the IH)} \\
\langle (A_x, pc+1+|\Pi_P(r,f)|, V'[v \Rightarrow i]) : e, i\rangle &\xrightarrow{\texttt{Jump}\ f} \\
\langle (A_x, \text{indexOf}(A_x, f), V'[v \Rightarrow i]) : e, i+j\rangle &
\end{aligned} \tag{4.11}$$

This proves the correctness of the not predicate.

Concatenation

There are three concatenation rules. In the case of the first rule, con.1 (`match` $g\ r_1 r_2\ s\ i = i + j + k$), we know that `match` $g\ r_1\ s\ i = i + j$ and `match` $g\ r_2\ s\ (i + j) = i + j + k$. The state transition sequence is shown below. In both steps, the induction hypothesis is used:

$$\begin{aligned}
\langle (A_x, pc, V) : e, i\rangle &\xrightarrow{\Pi_P(r_1,f)} \\
\langle (A_x, pc + |\Pi_P(r_1, f)|, V') : e, i+j\rangle &\xrightarrow{\Pi_P(r_2,f)} \\
\langle (A_x, pc + |\Pi_P(r_1, f)| + |\Pi_P(r_2, f)|, V'') : e, i+j+k\rangle &
\end{aligned} \tag{4.12}$$

Note that $|\Pi_P(r_1, f)| + |\Pi_P(r_2, f)| = |\Pi_P(r_1, f)\Pi_P(r_2, f)| = |\Pi_P(r_1 r_2, f)|$, since programs are composed using concatenation.

In the case of the second rule, con.2 (`match` $g\ r_1 r_2\ s\ i =$ `nil`), we know that `match` $g\ r_1\ s\ i = i + j$ and `match` $g\ r_2\ s\ (i + j) =$ `nil`. The state transition sequence is shown below. In both steps, the induction hypothesis is used:

$$\begin{aligned}
\langle (A_x, pc, V) : e, i\rangle &\xrightarrow{\Pi_P(r_1,f)} \\
\langle (A_x, pc + |\Pi_P(r_1, f)|, V') : e, i+j\rangle &\xrightarrow{\Pi_P(r_2,f)} \\
\langle (A_x, \text{indexOf}(A_x, f), V'') : e, z\rangle &
\end{aligned} \tag{4.13}$$

In the case of the third rule, con.3 (`match` $g\ r_1 r_2\ s\ i =$ `nil`), we know that `match` $g\ r_1\ s\ i =$ `nil` The state transition sequence is shown below. In the first step, the induction hypothesis is used:

$$\begin{aligned}
\langle (A_x, pc, V) : e, i\rangle &\xrightarrow{\Pi_P(r_1,f)} \\
\langle (A_x, \text{indexOf}(A_x, f), V') : e, z\rangle &
\end{aligned} \tag{4.14}$$

This proves the correctness of concatenation.

Ordered choice

There are three choice rules. In the case of the first rule, ord.1 (`match` $g\ r_1/r_2\ s\ i =$ `nil`), we know that `match` $g\ r_1\ s\ i =$ `nil` and `match` $g\ r_2\ s\ i =$ `nil`. The state transition sequence is shown below:

$$\begin{aligned}
\langle (A_x, pc, V) : e, i\rangle &\xrightarrow{\texttt{StorePosition}\ V} \\
\langle (A_x, pc+1, V[v \Rightarrow i]) : e, i\rangle &\xrightarrow{\Pi_P(r_1,L_1)} \text{ (using the IH)} \\
\langle (A_x, pc+2+|\Pi_P(r_1,f)|, V'[v \Rightarrow i]) : e, z\rangle &\xrightarrow{\texttt{MarkLabel}\ L_1} \\
\langle (A_x, pc+3+|\Pi_P(r_1,f)|, V'[v \Rightarrow i]) : e, z\rangle &\xrightarrow{\texttt{RestorePosition}\ V0} \\
\langle (A_x, pc+4+|\Pi_P(r_1,f)|, V'[v \Rightarrow i]) : e, i\rangle &\xrightarrow{\Pi_P(r_2,f)} \text{ (using the IH)} \\
\langle (A_x, \text{indexOf}(A_x, f), V''[v \Rightarrow i]) : e, z'\rangle &
\end{aligned} \tag{4.15}$$

Note that in the following two cases $5+|\Pi_P(r_1,f)|+|\Pi_P(r_2,f)| = |\Pi_P(r_1,f)\Pi_P(r_2,f)| = |\Pi_P(r_1/r_2,f)|$, since programs are composed using concatenation.

In the case of the second rule, ord.2 ($\texttt{match}\ g\ r_1/r_2\ s\ i = i+j$), we know that $\texttt{match}\ g\ r_1\ s\ i = i+j$. The state transition sequence is shown below:

$$\begin{aligned}
\langle (A_x, pc, V) : e, i\rangle &\xrightarrow{\texttt{StorePosition}\ V} \\
\langle (A_x, pc+1, V[v \Rightarrow i]) : e, i\rangle &\xrightarrow{\Pi_P(r_1,L_1)} \text{ (using the IH)} \\
\langle (A_x, pc+1+|\Pi_P(r_1,f)|, V'[v \Rightarrow i]) : e, i+j\rangle &\xrightarrow{\texttt{Jump}\ L_2} \\
\langle (A_x, pc+4+|\Pi_P(r_1,f)|+|\Pi_P(r_2,f)|, V'[v \Rightarrow i]) : e, i+j\rangle &\xrightarrow{\texttt{MarkLabel}\ L_2} \\
\langle (A_x, pc+5+|\Pi_P(r_1,f)|+|\Pi_P(r_2,f)|, V'[v \Rightarrow i]) : e, i+j\rangle &
\end{aligned} \tag{4.16}$$

In the case of the third rule, ord.3 ($\texttt{match}\ g\ r_1/r_2\ s\ i = \texttt{nil}$) we know that $\texttt{match}\ g\ r_1\ s\ i = \texttt{nil}$ and $\texttt{match}\ g\ r_2\ s\ i = i+j$. The state transition sequence is shown below:

$$\begin{aligned}
\langle (A_x, pc, V) : e, i\rangle &\xrightarrow{\texttt{StorePosition}\ V} \\
\langle (A_x, pc+1, V[v \Rightarrow i]) : e, i\rangle &\xrightarrow{\Pi_P(r_1,L_1)} \text{ (using the IH)} \\
\langle (A_x, pc+2+|\Pi_P(r_1,f)|, V'[v \Rightarrow i]) : e, z\rangle &\xrightarrow{\texttt{MarkLabel}\ L_1} \\
\langle (A_x, pc+3+|\Pi_P(r_1,f)|, V'[v \Rightarrow i]) : e, z\rangle &\xrightarrow{\texttt{RestorePosition}\ V0} \\
\langle (A_x, pc+4+|\Pi_P(r_1,f)|, V'[v \Rightarrow i]) : e, i\rangle &\xrightarrow{\Pi_P(r_2,f)} \text{ (using the IH)} \\
\langle (A_x, pc+4+|\Pi_P(r_1,f)|+|\Pi_P(r_2,f)|, V''[v \Rightarrow i]) : e, i+j\rangle &\xrightarrow{\texttt{MarkLabel}\ L_2} \\
\langle (A_x, pc+5+|\Pi_P(r_1,f)|+|\Pi_P(r_2,f)|, V''[v \Rightarrow i]) : e, i+j\rangle &
\end{aligned} \tag{4.17}$$

This proves the correctness of ordered choice.

Repetition

There are two repetition rules. In the case of the second rule, rep.2 ($\texttt{match}\ g\ r*\ \ s\ i = i$), we know that $\texttt{match}\ g\ r\ s\ i = \texttt{nil}$. The state transition sequence is shown below:

$$\begin{aligned}
\langle (A_x, pc, V) : e, i\rangle &\xrightarrow{\texttt{MarkLabel}\ L_1} \\
\langle (A_x, pc+1, V) : e, i\rangle &\xrightarrow{\texttt{StorePosition}\ v} \\
\langle (A_x, pc+2, V[v \Rightarrow i]) : e, i\rangle &\xrightarrow{\Pi_P(A_x, L_2)} \text{(using the IH)} \\
\langle (A_x, pc+2+|\Pi_P(A_x, L_2)|, V'[v \Rightarrow i]) : e, z\rangle &\xrightarrow{\texttt{MarkLabel}\ L_2} \\
\langle (A_x, pc+2+|\Pi_P(A_x, f)|, V'[v \Rightarrow i]) : e, z\rangle &\xrightarrow{\texttt{RestorePosition}\ v0} \\
\langle (A_x, pc+4+|\Pi_P(A_x, L_2)|, V''[v \Rightarrow i]) : e, i\rangle &
\end{aligned} \tag{4.18}$$

To prove the first rule, rep.1 ($\texttt{match}\ g\ r*\ \ s\ i = i+j+k$), we know that there must exist a derivation tree of the following shape:

$$\text{rep.1}\,\dfrac{\texttt{match}\ g\ r\ s\ i = i+j_n \qquad \text{rep.1}\,\dfrac{\texttt{match}\ g\ r\ s\ (i+j_n) = i+j_n+j_{n-1} \qquad \text{rep.1}\,\dfrac{\text{rep.1}\,\dfrac{\text{rep.2}\,\dfrac{\texttt{match}\ g\ r\ s\ (i+j_n+j_{n-1}+\ldots+j_1) = \texttt{nil}}{\texttt{match}\ g\ r*\ \ s\ (i+j_n+j_{n-1}+\ldots+j_1) = (i+j_n+j_{n-1}+\ldots+j_1)}}{\ldots}}{\texttt{match}\ g\ r*\ \ s\ (i+j_n+j_{n-1}) = i+j_n+j_{n-1}+\ldots+j_1}}{\texttt{match}\ g\ r*\ \ s\ (i+j_1) = i+j_n+j_{n-1}+\ldots+j_1}}{\texttt{match}\ g\ r*\ \ s\ i = i+j_n+j_{n-1}+\ldots+j_1} \tag{4.19}$$

We provide proof of the following implication by induction over the derivation tree depth n:

Theorem 22.

$$\begin{aligned}
&\text{If } \texttt{match}\ g\ r*\ \ s\ i = i+j_n+j_{n-1}+\ldots+j_1 \\
&\text{then } \langle (A_x, pc, V) : e, p_h\rangle \xrightarrow{\Pi_P(r*, f)} \langle (A_x, pc+|\Pi_P(r*, f)|, V') : e, p_0\rangle \\
&\text{where } \forall_{v \in V}[v \notin \Pi_P(r*, f) \Rightarrow V[v] = V'[v]]
\end{aligned} \tag{4.20}$$

Proof of the base case, where $n = 0$ and $\texttt{match}\ g\ r*\ s\ i = i$, has already been provided above.

The state transition sequence which proves the inductive case is shown below:

$$\begin{aligned}
\langle (A_x, pc, V) : e, i\rangle &\xrightarrow{\texttt{MarkLabel}\ L_1} \\
\langle (A_x, pc+1, V) : e, i\rangle &\xrightarrow{\texttt{StorePosition}\ V} \\
\langle (A_x, pc+2, V[v \Rightarrow i]) : e, i\rangle &\xrightarrow{\Pi_P(r, L_2)} \text{(using main IH)} \\
\langle (A_x, pc+2+|\Pi_P(r, L_2)|, V'[v \Rightarrow i]) : e, i+j_n\rangle &\xrightarrow{\texttt{Jump}\ L_1} \\
\langle (A_x, pc, V'[v \Rightarrow i]) : e, p_{h-1}\rangle &\xrightarrow{\Pi_P(r*, f)} \text{(using both IHs)} \\
\langle (A_x, pc, V'[v \Rightarrow i]) : e, i+j_n+j_{n-1}+\ldots+j_1\rangle &
\end{aligned} \tag{4.21}$$

This proves the correctness of repetition.

Non-terminals

To prove the correctness of non-terminals, an auxiliary theorem is needed. We will prove that for any grammar g, for any non-terminal A_x, for any position i in any string s, the following two implications hold:

Theorem 23.

$$\begin{aligned} \texttt{match}\ g\ r\ s\ i = i+j &\Rightarrow \langle (A_x, 0, \emptyset) : e, i\rangle \xrightarrow{\Pi_{NT}(r)} \langle e, i+j\rangle \\ \texttt{match}\ g\ r\ s\ i = \texttt{nil} &\Rightarrow \langle (A_x, 0, \emptyset) : e, i\rangle \xrightarrow{\Pi_{NT}(r)} \langle e, -1\rangle \end{aligned} \tag{4.22}$$

Proof: The first part is proven by the following state transition sequence (using the definition of Π_{NT}):

$$\begin{aligned} \langle (A_x, 0, \emptyset) : e, i\rangle &\xrightarrow{\Pi_P(r,L_1)} \text{ (using the IH)} \\ \langle (A_x, |\Pi_P(r, L_1)|, V') : e, i\rangle &\xrightarrow{\texttt{Return}\ 1} \\ \langle e, i\rangle & \end{aligned} \tag{4.23}$$

The second part is proven by the following state transition sequence (also using the definition of Π_{NT}):

$$\begin{aligned} \langle (A_x, 0, \emptyset) : e, i\rangle &\xrightarrow{\Pi_P(r,L_1)} \text{ (using the IH)} \\ \langle (A_x, |\Pi_P(r, L_1)|, V') : e, z\rangle &\xrightarrow{\texttt{MarkLabel}\ L_1} \\ \langle (A_x, 1+|\Pi_P(r, L_1)|, V') : e, z\rangle &\xrightarrow{\texttt{Return}\ 0} \\ \langle e, -1\rangle & \end{aligned} \tag{4.24}$$

Use of non-terminals With the help of the theorems described above, we can prove the correctness of non-terminals. There are once again two rules. For the first rule (var.1), $\texttt{match}\ g\ A_m\ s\ i = i+j$, we know that $\texttt{match}\ g\ g(A_m)\ s\ i = i+j$, with the following associated state transition sequence:

$$\begin{aligned} \langle (A_x, pc, V) : e, i\rangle &\xrightarrow{\texttt{Call}\ A_m} \\ \langle (A_m, 0, \emptyset) : (A_x, pc+1, V) : e, i\rangle &\xrightarrow{\Pi_{NT}(A_m)} \text{ (Using the auxiliary theorem above)} \\ \langle (A_x, pc+1, V) : e, i\rangle &\xrightarrow{\texttt{EndCall}\ f} \\ \langle (A_x, pc, V) : e, i+j\rangle & \end{aligned} \tag{4.25}$$

For the second rule (var.2), $\texttt{match}\ g\ A_m\ s\ i = \texttt{nil}$, we know that $\texttt{match}\ g\ g(A_m)\ s\ i = \texttt{nil}$, with the following associated state transition sequence:

$$\begin{aligned} \langle (A_x, pc, V) : e, i\rangle &\xrightarrow{\texttt{Call}\ A_m} \\ \langle (A_m, 0, \emptyset) : (A_x, pc+1, V) : e, i\rangle &\xrightarrow{\Pi_{NT}(A_m)} \text{ (Using the auxiliary theorem above)} \\ \langle (A_x, pc+1, V) : e, -1\rangle &\xrightarrow{\texttt{EndCall}\ f} \\ \langle (A_x, \text{indexOf}(A_x, f), V) : e, \texttt{nil}\rangle & \end{aligned} \tag{4.26}$$

This proves correctness of non-terminals.

4.3 A note on complexity

4.3.1 General complexity

Our parsing machine does not improve worst case time or space complexity, compared to the parsing machine from Medeiros and Ierusalimschy [11]. To demonstrate, the following PEG will use a $O(n)$ memory, where n is the input size: `A <- (.  A)?`. The linear memory usage in this case is because of the recursion. Each time we recurse, we push a new stack frame on the stack, which uses a constant amount of memory depending only on the PEG expression.

We also do not improve on worst case time complexity, which is still exponential. This cannot be improved without adding memoization.

4.3.2 Complexity for simple patterns

For patterns that do not use recursion, we can make a stronger claim about space complexity. We expect this to be the more common case: most pattern-matching tasks use simple expressions and do not require recursion.

We know that the maximum call depth is a constant number independent of the input string, because recusion is not allowed. The maximum stack frame size is determined by the PEG that we're matching, and is also independent of the input string. This means that the memory usage is $O(C * F)$ where C is the maximum call depth and F is the maximum frame size. Note that the memory usage is no longer dependent on the input string.

Chapter 5

Optimizations

It's hard to formally quantify the effectiveness of optimizations, as some improvements might depend on processor architecture. In general, we assume that *executing* less instructions is better. Note that this is different from the number of instruction in the sequence, since the sequence might contain loops. This means that we should optimize in the following way:

1. Failing as early as possible, to ensure that if we fail we have done as little work as possible
2. Delaying instructions as long as possible, to once again ensure that if we fail we have done as little work as possible
3. Removing or skipping any instructions that will not affect the outcome

In our implementation we are using all of the optimizations described below. These benchmarks improve run times by a factor roughly bewteen 4 to 10, according to the benchmarks we performed in Chapter 7.

The general compilation process is shown in Figure 5.1. There are two points where we can apply optimizations. The first is inside the Π function, which transforms a PEG into a compiled program C. These optimizations are described in Section 5.1. The second point is actual 'Optimizations' step, for which we describe all optimizations in Section 5.2. Note that all optimizations happen before the input string is known.

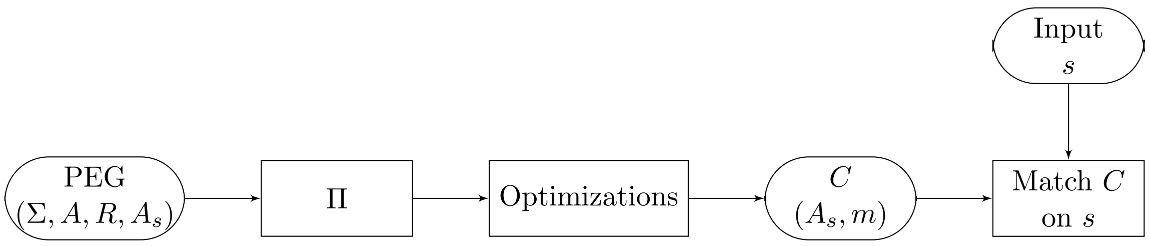

Figure 5.1: The PEG compilation and execution process

5.1 Modifying Π

5.1.1 Character classes

Instead of only allowing single characters to be matched by the `Char` instruction, we can extend it to match any set of characters.

Definition 24. The definition of the `Char` instruction now becomes:

$$\langle (A_x, pc, V) : e, i\rangle \xrightarrow{\texttt{Char}\ C\ N\ f} \langle (A_x, pc', V) : e, i\rangle$$

$$\text{where } pc' = \begin{cases} pc + 1 \text{ if } s[i+N] \in C \\ \text{indexOf(f, p) otherwise} \end{cases} \tag{5.1}$$

In the definition above, C is no longer a single character, but a set of characters.

This optimization allows each implementation to define its own way to check if a character is within a certain set. For example, one common expression in a PEG is `[a-z]`. Instead of producing a prioritized choice for each of the 26 characters, forcing every implementation to check the characters one-by-one, this optimization grants the implementation the freedom to implement a more efficient check. For example, on most platforms `[a-z]` can be translated to a subtract followed by an unsigned comparison.

Example 25. The expression `[a-z]` can now be translated to just three instructions:

$$\begin{aligned} &\texttt{BoundsCheck}\ 0\ f \\ &\texttt{Char}\ \{\text{'a', 'b', 'c', ..., 'z'}\}\ 0\ f \\ &\texttt{Advance}\ 1 \end{aligned} \tag{5.2}$$

5.1.2 Reducing the number of Advance-instructions

As you may have noticed, most instructions contain offsets, which have so far always been 0. We will now use these offset parameters to reduce the number of instructions needed for matching strings (i.e. multiple characters that are matched sequentially).

Example 26. Consider the following sequence of instructions:

$$\begin{aligned} &\texttt{BoundsCheck}\ 0\ f \\ &\texttt{Char}\ \text{'a'}\ 0\ f \\ &\texttt{Advance}\ 1 \\ &\texttt{BoundsCheck}\ 0\ f \\ &\texttt{Char}\ \text{'b'}\ 0\ f \\ &\texttt{Advance}\ 1 \\ &\texttt{BoundsCheck}\ 0\ f \\ &\texttt{Char}\ \text{'c'}\ 0\ f \\ &\texttt{Advance}\ 1 \end{aligned} \tag{5.3}$$

This is unnecessarily verbose. Since all three instructions jump to the same fail label, we know that it does not matter whether the pattern fails after the first, second, fourth, or sixth instruction. We can save four instructions by changing the sequence to:

$$
\begin{aligned}
&\texttt{BoundsCheck}\ 2\ f\\
&\texttt{Char}\ \text{'a'}\ 0\ f\\
&\texttt{Char}\ \text{'b'}\ 1\ f\\
&\texttt{Char}\ \text{'c'}\ 2\ f\\
&\texttt{Advance}\ 3
\end{aligned}
\tag{5.4}
$$

This does not pose a problem to the correctness claims we have made, since we allowed the position i to be any natural number after an instruction fails. It does however significantly improve performance, because we eliminate not only three additions, but also three branching instructions, which can cause costly branch mispredictions on the CPU.

Definition 27. We define expression r to be a 'string', if and only if r contains only concatenations of terminals and the any operator.

Definition 28. A function stringSize(r) is used to determine the size of the string inside r. It is defined as follows:

$$
\begin{aligned}
&\text{stringSize}(c) = 1\\
&\text{stringSize}(.) = 1\\
&\text{stringSize}(r_1r_2) = \begin{cases} \mathit{stringSize}(r_1) + \mathit{stringSize}(r_2) \text{ if } r_1 \text{ is a string} \\ \mathit{stringSize}(r_1) \text{ otherwise} \end{cases}\\
&\text{stringSize}(r) = 0
\end{aligned}
\tag{5.5}
$$

We can now use this function to define Π_S, a function that generates the faster string matching code:

Definition 29.

$$
\begin{aligned}
\Pi_S(., f, n, m) =& \emptyset\\
\Pi_S(c, f, n, m) =& \texttt{Char}\ c\ n\ f \text{ if } n < m\\
\Pi_S(c, f, n, m) =& \begin{matrix}\texttt{Char}\ c\ n\ f\\ \texttt{Advance}\ n\end{matrix} \text{ if } n = m\\
\Pi_S(r_1r_2, f, n) =& \Pi_S(r_1, f, n, m)\\
& \Pi_S(r_2, f, n + \text{stringSize}(r_1), m)\\
\Pi_S(r, f, n) =& \Pi_P(r, f)
\end{aligned}
\tag{5.6}
$$

Definition 30. Finally, we can use Π_S in our previously defined function Π_P:

$$
\Pi_P(r_1r_2, f) = \begin{cases} \begin{matrix}\texttt{BoundsCheck}\ (\text{stringSize}(r_1r_2) - 1)\ f\\ \Pi_S(r_1r_2, f, 0, \text{stringSize}(r_1r_2) - 1)\end{matrix} \text{ if stringSize}(r_1r_2) > 0 \\ \Pi_P(r_1r_2, f) \text{ otherwise} \end{cases}
\tag{5.7}
$$

5.1.3 Non-terminal inlining

If non-terminals are really small, the method invocation overhead is often the main cause of slowdowns. Assume we have some processor-dependent heuristic H_{inline} that tells us when the method invocation overhead is more costly than the code size increase (increased code size might cause slowdowns if the code does not fit inside the processor cache) caused by inlining the invocation, for example by looking at:

1. The size of the non-terminal
2. Whether the non-terminal is recursive
3. How many other non-terminals are called by the non-terminal

Definition 31. We modify the definition of Π to include this heuristic:

$$\begin{aligned} \Pi_P(A_x, f) &\equiv \begin{matrix}\texttt{Call } A_x \\ \texttt{EndCall } f\end{matrix} \text{ if not } H_{\text{inline}}(A_x) \\ \Pi_P(A_x, f) &\equiv \Pi_P(R(A_x), f) \text{ if } H_{\text{inline}}(A_x) \end{aligned} \tag{5.8}$$

This optimization can be especially useful when using parameterized non-terminals.

Example 32. Re-using our example from Section 3.1:
`EndsWithIng <- Until<\([a-z]), \('ing')>`
Because of this optimization, both `[a-z]` and `'ing'` can be inlined, making the parameterized non-terminal as fast as the hand-written expression.

5.2 Transforming instruction sequences

This second set of optimizations is executed after the initial compilation using Π. All optimizations below operate on a single InstructionList (i.e. a single non-terminal), and produce a new list of optimized instructions. An implementation might iteratively run the optimizations until the instructions sequence reaches a form where none of the optimizations have any effect on it.

5.2.1 Normalization

For all following optimizations, we assume that the sequence of instructions is in a normal form. In particular, we want the sequence of instructions to adhere to the following rules:

1. A label refers to a unique position in the code, i.e. a `MarkLabel` instruction will never be followed by another `MarkLabel` instruction.
2. For each `MarkLabel` L_i instruction, there is at least one instruction that can jump to L_i.
3. For each `StorePosition` V_i instruction, there is at least one `RestorePosition` V_i instruction.
4. An instruction is reachable, i.e. there exists at least one execution path that contains the instruction.

Example 33. To illustrate point 4, consider the following sequence of instructions:

$$\begin{aligned} &\texttt{Jump } L \\ &\texttt{Char } [a - z]\ 0\ f \\ &\texttt{Char } \text{':'}\ 1\ f \\ &\texttt{MarkLabel } L \end{aligned} \tag{5.9}$$

The two instructions between the `Jump` and the `MarkLabel` can never be reached, because the `Jump` will always jump over the two instructions. Therefore, no execution path contains the two `Char` instructions, so they should be removed from the instruction sequence.

Note that optimizations will still produce correct results if a sequence of instructions is not in the described normal form. Rather, the optimizations often have preconditions that require the normal form, so not having this form renders the optimizations ineffective. Implementing an algorithm that modifies an instruction sequence to make sure it adheres to the rules above should be trivial.

5.2.2 `Char` ordering

This optimization falls into the category of ensuring that if we fail to match, we fail as early as possible. It should be noted that the effectiveness of this optimization is dependent on the character distribution of the input text, which is unknown when this optimization is used. Therefore, we assume the characters in the string are uniformly distributed. In practice this assumption turns out to be precise enough most of the time.

Example 34. Consider the following sequence of instructions, which match any lower case letter followed by a colon:

$$\begin{aligned}&\texttt{BoundsCheck}\ 2\ f\\&\texttt{Char}\ [a-z]\ 0\ f\\&\texttt{Char}\ \text{':'}\ 1\ f\\&\texttt{Advance}\ 3\end{aligned} \tag{5.10}$$

Assuming an uniformly distributed input, the probability that $s[i]$ is a lower case letter is greater than the probability that $s[i+1]$ is a colon. Therefore it makes more sense to order the instructions like this:

$$\begin{aligned}&\texttt{BoundsCheck}\ 2\ f\\&\texttt{Char}\ \text{':'}\ 1\ f\\&\texttt{Char}\ [a-z]\ 0\ f\\&\texttt{Advance}\ 3\end{aligned} \tag{5.11}$$

For any two `Char` instructions at position pc and $pc+1$ respectively, we swap the instructions if they both have the same fail label, and the second `Char` instruction matches less characters than the first `Char` instruction.

5.2.3 Consolidating BoundsChecks

Consider a subsequence that looks like the following:

$$\begin{aligned}&\texttt{BoundsCheck}\ j\ f\\&\texttt{Char}\ c_1\ x_1\ f\\&\texttt{Char}\ c_2\ x_2\ f\\&\ldots\\&\texttt{Char}\ c_n\ x_n\ f\\&\texttt{BoundsCheck}\ k\ f\end{aligned} \tag{5.12}$$

Since all instructions will jump to the same fail label, we can safely remove the second `BoundsCheck`, and modify the first `BoundsCheck` to `BoundsCheck` $\max(j,k)\ f$.

5.2.4 Deduplication

Given a subsequence of instructions of the following shape:

$$\begin{aligned}&I_1\\&I_2\\&...\\&I_n\\&\texttt{Jump}\ L_k\end{aligned}\tag{5.13}$$

If this subsequence occurs twice or more in the full sequence, we can optimize this by replacing the subsequences by a `Jump` $L_{k'}$ instruction, and then modifying the sequence around the `MarkLabel` L_k instruction to this:

$$\begin{aligned}&\texttt{Jump}\ L_k\\&\texttt{MarkLabel}\ L_{k'}\\&I_1\\&I_2\\&...\\&I_n\\&\texttt{MarkLabel}\ L_k\end{aligned}\tag{5.14}$$

While this optimization does not reduce the number of instructions that are executed, it does reduce the overal instruction list size, which increases the chances that the entire program will fit in the CPU's instruction cache. Additionally, in some cases deduplication renders some instructions unnecessary, which allows other optimizations to remove them.

5.2.5 Delaying BoundsChecks, Advances and StorePositions

The `Advance` and `StorePosition` instructions cannot fail, so we want to execute them as late as possible. The `BoundsCheck` instruction can fail, but this is highly unlikely for pattern-matching tasks, since most of the time we will not be at the end of the file. Therefore, we want to execute the `Char`-instructions before the three instructions above, and the `BoundsCheck` before the `Advance` and `StorePosition` instructions.

We can achieve this by 'pushing' the instructions down. Consider the following subsequence:

$$\begin{aligned}&\texttt{StorePosition}\ V_3\\&\texttt{Char}\ c\ 0\ L_k\end{aligned}\tag{5.15}$$

After storing the position, the machine may take one of two paths. It may successfully match c, and continue executing, or it may fail to match c, and jump to L_k. If we want to push the `StorePosition` instruction down, we can do this by moving it to two locations.

For the successful match case, we push it down one instruction:

$$\begin{array}{l}\texttt{Char}\ c\ 0\ L_k \\ \texttt{StorePosition}\ V_3\end{array} \tag{5.16}$$

For the unsuccesful case, we add a stub before the `MarkLabel` instruction, and patch the `Char` instruction:

$$\begin{array}{l}\texttt{Char}\ c\ 0\ L_{k'} \\ \ldots \\ \texttt{Jump}\ L_k \\ \texttt{MarkLabel}\ L_{k'} \\ \texttt{StorePosition}\ V_3 \\ \texttt{MarkLabel}\ L_k\end{array} \tag{5.17}$$

We can apply this same technique to `BoundsCheck`, `Advance` and `StorePosition` instructions.

5.2.6 Removing unnecessary instructions

Definition 35. An `Advance` N instruction at position *pc* in sequence I is useful if, and only if:

1. $N \neq 0$
2. If $\text{needed}_{adv}(pc) = \text{true}$, where needed_{adv} is defined as follows:

$$\text{needed}_{adv}(pc) = \begin{cases} \text{false if } I[pc] = \texttt{RestorePosition}\ v \\ \text{needed}_{adv}(pc+1) \text{ if } I[pc] = \texttt{Advance}\ N \\ \text{needed}_{adv}(pc+1) \text{ if } I[pc] = \texttt{MarkLabel}\ L \\ \text{needed}_{adv}(indexOf(L)) \text{ if } I[pc] = \texttt{Jump}\ L \\ \text{true otherwise} \end{cases}$$

Informally, we are checking if the instructions will do anything with the modified position. If we know that the modified position will be disregarded without being used anywhere, it's unnecessary.

Definition 36. A `StorePosition` v instruction at position *pc* in sequence I is useful if, and only if:

1. $N \neq 0$
2. If $\text{needed}_{store}(pc) = \text{true}$, where needed_{store} is defined as follows:

$$\text{needed}_{store}(pc) = \begin{cases} \text{true if } I[pc] = \texttt{RestorePosition } v \\ \text{false if } I[pc] = \texttt{StorePosition } v \\ \text{needed}_{store}(indexOf(L)) \text{ if } I[pc] = \texttt{Jump } L \\ \text{needed}_{store}(indexOf(L)) \vee \text{needed}_{store}(pc+1) \text{ if} \\ \qquad I[pc] = \texttt{Char} \\ \qquad \text{or } I[pc] = \texttt{BoundsCheck} \\ \qquad \text{or } I[pc] = \texttt{BeginCall} \\ \text{needed}_{store}(pc+1) \text{ otherwise} \end{cases}$$

Informally, we are checking whether the position that we're storing is used at all before it is overwritten by another value. If it's never used, the instruction is not needed. While this might seem equivalent to removing any `StorePosition` v instruction for which no `RestorePosition` v exists, this optimization also works in cases like the following:

Example 37.

$$\begin{array}{l} \texttt{StorePosition } v \\ \texttt{Jump } L_k \\ \ldots \\ \texttt{MarkLabel } L_k \\ \texttt{StorePosition } v \\ \ldots \\ \texttt{RestorePosition } v \end{array} \tag{5.18}$$

Here, we store the position in v, jump to L_k, and store the position again, then restore it. While there exists a `RestorePosition` v, the first `StorePosition` v can still be removed, because it is overwritten by the second `StorePosition` a few instructions later.

All instructions that are not useful can be removed without changing the semantics of the program.

5.2.7 Jump target optimization

This optimization is aimed at reducing the number of instructions that are executed by skipping over instructions that will not affect the outcome. We do this by performing an analysis of the instruction sequence to determine how far bounds have been checked, and which characters either matched or failed to match at any point in the sequence. We call this process 'backtracing' since it involves following execution paths in the reverse direction back to the start of the execution. We describe this in Section 5.2.7

After doing this, we can use this information to skip over instructions that don't affect the outcome. We call this 'forwardtracing', since it involves simulating the execution of the instruction sequence using the information gathered using backtracing. We describe this in Section 5.2.7

We can also use the information gathered using backtracing to eliminate instructions that are guaranteed to always succeed or fail, which we descibe in Section 5.2.7

Backtracing

Informally, bactracing performs static analysis on a single InstructionList. Given any position *pos* in the InstructionList, we can build a list of all execution paths that will eventually end up executing the instruction at position *pos*. In other words, when representing the InstructionList as a directed flow graph, this list consists of all paths from the start node to the node belonging to the instruction at position *pos*.

Example 38. Consider a situation where there are two paths to the instruction at position *pos*. (For the sake of the example, we assume that there are no `Advance` instructions anywhese in the paths)

1. Path 1 contains a `BoundsCheck` 4 L_f instruction which succeeds, and a `BoundsCheck` 7 L_f instruction which fails
2. Path 3 contains a `BoundsCheck` 2 L_f instruction which succeeds, and a `BoundsCheck` 10 L'_f instruction which fails

From this information, we can establish the bounds at position *pos*: The input string is at least $i + 2$, and no more than $i + 10$ characters long.

We can do this analysis for both `Char` and `BoundsCheck` instructions. Below, we will describe how the information that we retrieve from separate paths can be merged into a something that holds for all paths. We also describe the backtracing algorithm in more detail.

We are using two auxiliary functions. Both are described in Algorithm 1. Jump-Sources returns a list of all locations that can jump to a certain label. We will use this to follow the execution path in reverse. Store-Sources returns a list of all locations where a variable might have been stored, given a `RestorePosition` instruction. This is used to backtrace through a `RestorePosition` instruction.

Definition 39. We can store the known bounds of the input string in a tuple $(B_{>=}, B_{<})$. We define the intersection $\cap$ of two tuples to be:
$(a_{>=}, a_{<}) \cap (b_{>=}, b_{<}) = (\min(a_{>=}, b_{>=}), \max(a_{<}, b_{<}))$
We define subtraction as follows:
$(a_{>=}, a_{<}) - k = (a_{>=} - k, a_{<} - k)$

Definition 40. We can store the characters that have matched and that failed to match in a tuple (C_m, C_f), where C_m and C_f are finite functions $\mathbb{N} \to \mathit{Chars}$. This function maps a position in the string, relative to the current position i, to a list of all characters that (failed to) match at that position.

We define the intersection $(C_m, C_f) \cap (C'_m, C'_f) = (C_m \cap C'_m, C_f \cap C'_f)$.

We define advance$(n, (C_m, C_f))$ to subtract n from all indices of the finite function, and remove all numbers that are now less than 0.

Removing unneeded instructions

We can use the backtracing to determine if there are any `BoundsCheck` or `Char` instructions that will always fail or always succeed. Instructions that always fail can be replaced by an unconditional `Jump`, and instructions that always succeed can be removed entirely.

Algorithm 1 Auxiliary procedures

```
1:  procedure Jump-Sources(I, L)
2:      result ← $\emptyset$
3:      for pos $\in$ 1...$|I|$ do
4:          if I[pos] = Jump $L$ $\vee$ I[pos] = BoundsCheck $N$ $L$ $\vee$ I[pos] = Char $C$ $N$ $L$ then
5:              result ← result $\cup$ {pos}
        return result
6:  procedure Store-Sources(I, pos, v)
7:      stack ← $\emptyset$
8:      processed ← $\emptyset$
9:      result ← $\emptyset$
10:     stack.push(pos)
11:     while  do|stack| > 0
12:         pos ← stack.pop()
13:         if pos $\notin$ processed then
14:             processed ← processed $\cup$ {pos}
15:             instr ← I[pos]
16:             if instr = MarkLabel $L$ then
17:                 stack.pushMultiple(Jump-Sources($L$, pos))
18:                 stack.push(pos $-$ 1)
19:             else if instr = StorePosition $v$ then
20:                 result ← result $\cup$ {pos}
21:             else if instr $\notin$ {Jump $L$, Return $B$, EndCall $L$} then
22:                 stack.push(pos $-$ 1)
        return result
```

Algorithm 2 Backtracing bounds

```
1: procedure CHECK-BOUNDS(I, pos, visited)
2:     if pos $< 0$ then return $(-1, \infty)$
3:     instr $\leftarrow$ I[pos]
4:     if instr = MarkLabel $L$ then
5:         if $L \in$ visited then return $(-1, \infty)$
       return $\bigcap_{x \in \text{Jump-Sources}(L,\text{pos})}$ Check-Bounds-Helper$(x, \text{visited} \cup \{L\}, \text{true})$
6:     else if instr = Advance $N$ then
       return Check-Bounds-Helper$(\text{pos} - 1, \text{visited}, \text{false}) - N$
7:     else if instr = RestorePosition $v$ then
       return $\bigcap_{x \in \text{Store-Sources}(L,\text{pos})}$ Check-Bounds-Helper$(x, \text{visited}, \text{true})$
8:     else if instr $\in$ {Jump $L$, Return $B$, EndCall $L$} then
       return $(-1, \infty)$
9:     else
       return Check-Bounds-Helper$(\text{pos} - 1, \text{visited}, \text{false})$
10: procedure CHECK-BOUNDS-HELPER(I, pos, visited, jump)
11:     instr $\leftarrow$ I[pos]
12:     if instr = BoundsCheck $N$ $L$ then
13:         result $\leftarrow$ Check-Bounds(I, pos)
14:         if jump then
        return result $\cap (-1, N)$
15:         else
        return result $\cap (N, \infty)$
16:     else
        return Check-Bounds(I, pos)
```

Algorithm 3 Backtracing chars

```
1: procedure CHECK-CHARS(I, pos, visited)
2:     if pos $< 0$ then return $(-1, \infty)$
3:     instr $\leftarrow$ I[pos]
4:     if instr = MarkLabel $L$ then
5:         if $L \in$ visited then return $(-1, \infty)$
        return $\bigcap_{x \in \text{Jump-Sources}(L,\text{pos})}$ Check-Chars-Helper($x$, visited $\cup \{L\}$, true)
6:     else if instr = Advance $N$ then
        return Check-Chars-Helper(pos $- 1$, visited, false) $- N$
7:     else if instr = RestorePosition $v$ then
        return $\bigcap_{x \in \text{Store-Sources}(L,\text{pos})}$ Check-Chars-Helper($x$, visited, true)
8:     else if instr $\in$ {Jump $L$, Return $B$, EndCall $L$} then
        return $(-1, \infty)$
9:     else
        return Check-Bounds-Helper(pos $- 1$, visited, false)
10: procedure CHECK-CHARS-HELPER(I, pos, visited, jump)
11:     instr $\leftarrow$ I[pos]
12:     if instr = Char $C$ $NL$ then
13:         result $\leftarrow$ Check-Chars($I$, pos)
14:         if jump then
        return result $\cap (\emptyset, \{\})$
15:         else
        return result $\cap (, \emptyset)$
16:     else
        return Check-Chars($I$, pos)
```

Forwardtracing

We use forward tracing, described in Algorithm 4, to optimize the jump target of `Jump`, `BoundsCheck` and `Char` instructions. Informally, we do this by updating the jump target of these instructions to skip any instruction that we know to be unneeded because of the backtracing result.

Example 41. Consider an instruction sequence of the following shape:

$$\begin{aligned}&\texttt{Jump}f\\&...\\&\texttt{MarkLabel}\ f\\&\texttt{BoundsCheck}\ 2\ f\\&...\end{aligned}\tag{5.19}$$

Assume that backtracing has established that there are at least 3 more characters in the input string. This means that the `BoundsCheck` 2 f instruction is unneeded when jumping to f from the `Jump` f instruction. Therefore, we can update the instruction sequence to the following:

$$\begin{aligned}&\texttt{Jump}f'\\&...\\&\texttt{MarkLabel}\ f\\&\texttt{BoundsCheck}\ 2\ f\\&\texttt{MarkLabel}\ f'\\&...\end{aligned}\tag{5.20}$$

Note that we have to leave the old `MarkLabel` f in the sequence, since we do not know if there are any other jumps to f.

Our forwardtracing algorithm has two tiers. Forward-Trace-Single will follow a single path for as far as it can be skipped. Forward-Trace will then use the results of the single trace to look past jumps for which we were not able to gather enough information during backtracing to predict their outcome, by following both jumps. If both of the jumps still lead to the same location, the outcome of the current jump does not matter and we can skip it.

Optimizing jump targets

We use the forward tracing methods to determine the best jump target for `Char`, `Bounds-Check` and `Jump` instructions. We can then insert a new label at that position, and update the label of the instruction to the new label.

Algorithm 4 Forwardtracing

```
1: procedure FORWARD-TRACE(I, pos, backtracePos)
2:     result $\leftarrow$ Forward-Trace-Single($I$, pos, $backtracePos$)
3:     instr $\leftarrow$ $I$[pos]
4:     if instr = RestorePosition $v$ then
5:         if $L \in$ visited then return $(-1, \infty)$
6:         result2 $\leftarrow$ Forward-Trace-Single($I$, pos, $backtracePos$)
7:         if $I$[result2] = RestorePosition $v_2$ then return result2
       return result
8:     else if instr = Char $C$ $N$ $L$ $\vee$ instr = BoundsCheck $N$ $L$ then
9:         result1 $\leftarrow$ Forward-Trace-Single($I$, pos + 1, $backtracePos$)
10:        result2 $\leftarrow$ Forward-Trace-Single($I$, indexOf($I$, $L$), $backtracePos$)
11:        if result1 = result2 then return result1
       return result
12: procedure FORWARD-TRACE-SINGLE(I, pos, backtracePos)
13:     while true do
14:         instr $\leftarrow$ I[pos]
15:         if instr = Jump $L$ then
16:             pos $\leftarrow$ indexOf($I$, $L$)
17:         else if instr = MarkLabel $L$ then
18:             pos $\leftarrow$ pos + 1
19:         else if instr = Char $C$ $NL$ then
20:             $(C_m, C_f)$ $\leftarrow$ Check-Chars(I, pos, $\emptyset$)
21:             if $C \in C_m[N]$ then
22:                 pos $\leftarrow$ pos + 1
23:             else if $C \in C_f[N]$ then
24:                 pos $\leftarrow$ indexOf(I, $L$)
25:             else return pos
26:         else if instr = BoundsCheck $N$ $L$ then
27:             $(B_{\Leftarrow}, B_{>})$ $\leftarrow$ Check-Bounds(I, pos, $\emptyset$)
28:             if $N \Leftarrow B_{\Leftarrow}$ then
29:                 pos $\leftarrow$ pos + 1
30:             else if $N > B_{>}$ then
31:                 pos $\leftarrow$ indexOf(I, $L$)
32:             else return pos
33:         else return Check-Chars(I, pos)
```

5.2.8 Skipping or removing unneeded `RestorePositions`

If we have an instruction sequence of the following shape, where all I_x are not an `Advance` or `BeginCall` instruction:

$$\begin{aligned}
&\texttt{StorePosition}\ v\\
&I_1\\
&I_2\\
&...\\
&I_n\\
&\texttt{BoundsCheck}\ N\ L_{fail}\\
&...\\
&\texttt{MarkLabel}\ L_{fail}\\
&\texttt{RestorePosition}\ v
\end{aligned} \tag{5.21}$$

We can optimize this sequence to:

$$\begin{aligned}
&\texttt{StorePosition}\ v\\
&I_1\\
&I_2\\
&...\\
&I_n\\
&\texttt{BoundsCheck}\ N\ L'_{fail}\\
&...\\
&\texttt{MarkLabel}\ L_{fail}\\
&\texttt{RestorePosition}\ v\\
&\texttt{MarkLabel}\ L'_{fail}
\end{aligned} \tag{5.22}$$

We can take this one step further and perform abstract interpretation [4] on the entire instruction sequence, to infer the offsets stored in all variables.

Example 42. The `RestorePosition` in the following instruction sequence is unneeded:

$$\begin{aligned}
&\texttt{StorePosition}\ v\\
&\texttt{Advance}\ 3\\
&\texttt{BoundsCheck}\ N\ L'_{fail}\\
&\texttt{RestorePosition}\ v\\
&\texttt{MarkLabel}\ L'_{fail}
\end{aligned} \tag{5.23}$$

Using abstract interpretation can infer that when arriving at the `RestorePosition` instruction, v will be equal to $i-3$. We can now replace the `RestorePosition` instruction with `Advance -3`.

In general we can replace all `RestorePosition` instructions, where we have inferred that the offset is always a fixed number, by an `Advance` instruction.

5.2.9 Merging `StorePositions`

Definition 43. We can merge two `StorePosition` instructions, `StorePosition` v_A and `StorePosition` v_B when for each position where we find `StorePosition` v_A, the other instruction `StorePosition` v_B can be found directly before or after it.

Definition 44. We merge `StorePosition` v_A and `StorePosition` v_B by removing all `StorePosition` v_B instructions, and substituting all `RestorePosition` v_B instructions with `RestorePosition` v_A

Merging two variables into one reduces the number of `StorePosition` needed, and also reduces the stack frame size, which reduces memory usage and increases performance.

5.2.10 Adding fast-paths to loops

`BoundsChecks` can become very costly when executing them inside a loop (i.e. a repeat). When matching patterns on text files, we attempt to match the pattern on each input position in the string, so most of the time we will not be at the end of the file. This means we don't need the `BoundsChecks` in the common case. We can solve this problem by adding a separate fast and slow path for the loop. Consider the following structure:

$$\begin{array}{l} \texttt{MarkLabel } L_{loop} \\ I_1 \\ I_2 \\ \ldots \\ I_n \\ \texttt{Jump } L_{loop} \end{array} \tag{5.24}$$

Here, I_1, I_2, I_n are instructions, and L_{loop} is a label. Assume we know heuristically that the loop body $(I_1, I_2, ..., I_n)$ contains many `BoundsChecks`. We can then optimize the instruction sequence to the following:

$$\begin{array}{l} \texttt{MarkLabel } L_{fast} \\ \texttt{BoundsCheck } K\ L_{slow} \\ I_1 \\ I_2 \\ \ldots \\ I_n \\ \texttt{Jump } L_{fast} \\ \texttt{MarkLabel } L_{slow} \\ I'_1 \\ I'_2 \\ \ldots \\ I'_n \\ \texttt{Jump } L_{slow} \end{array} \tag{5.25}$$

We choose K to be some large number, for example the largest bound checked in the loop body. We generate the duplicated instruction sequence $I'_1, I'_2, ..., I'_n$ by replacing each label that is marked inside the loop body with a fresh label. This ensures that we will not have duplicate `MarkLabel` L instructions, and that we do not jump back to the fast path.

Other optimizations will be able to further optimize the fast path, by removing unneeded `BoundsCheck`s, or delaying them for longer. The slow path will only be invoked for the last K characters in the input string.

Chapter 6

Implementation

Our implementation provides both a library and a `grep`-like tool named `peg-match`. Both are available on GitHub, under the LGPL v3.0 [1].

6.1 peg-match

`peg-match` is a command-line tool that provides functionality similar to the popular unix-tool `grep`:

Example 45. The following command will print all words starting with 'th' in the file 'input.txt':

```
peg-match "'th' [a-z]+" input.txt
```

Having a tool that is just as good as an existing tool will not convince users of the existing tool to switch, since there is no direct benefit associated with switching. To make the switch worthwhile, `peg-match` needs to also have functionality that cannot be found in `grep`. The PEG extensions described in Section 3.1 were aimed at making PEGs easier to use in pattern-matching use cases. We have attempted to do the same thing for the tool itself, by adding features not found in tools like `grep` that make using it easier.

6.1.1 The `peg-match` standard library

Since we're no longer bound to the limitations of regular expressions, we can add a standard library of commonly used patterns and parameterized non-terminals. Having a standard library available means that hard-to-remember patterns or constructs can be stored under an easy-to-remember name.

Example 46. As discussed before, matching all words ending in 'ing' is more complicated due to the limited backtracking of PEGs. Therefore, `peg-match` provides a parameterized non-terminal to make this process as easy as it would be when using regular expressions. The following command will match all words ending in 'ing':

```
peg-match 'String::Until<\([a-z]), \("ing")>' input.txt
```

This uses the extended grammar described in Section 3.1.

Currently, the following modules for common tasks are included with `peg-match`:

[1] https://github.com/Jos635/SharpPeg

1. The `Char::` namespace provides sets of commonly used character classes, which aims to replace the backslash-followed-by-a-letter commonly used in regular expression matching libraries. For example, \w in a regular expression is available as `Char::Letter` in `peg-match`.

2. The `String::` namespace provides mostly parameterized non-terminals for common string structures, for example:

   (a) `String::Until<Repeat, Ending>` - A word consisting of `Repeats`, ending in `Ending`

   (b) `String::Join<Separator, Item>` - One or more `Items`, with `Separator` in-between the items.

   (c) `String::Quoted<Repeat, QuoteChar>` - Zero or more `Repeats` surrounded by `QuoteChar`. `QuoteChar` may also occur in the string, if it is preceded by a backslash.

3. The `Phone` namespace provides a grammar that can be used to match common phone number notation formats.

4. The `Net::Email` namespace can be used to match e-mail addresses.

5. The `Net::Ip` namespace can be used to match any IP address, which includes IPv4, IPv6 and future IP formats, currently named "IPvFuture".

6. The `Net::Ipv4` namespace can be used to match IPv4 addresses, as well as the CIDR notation.

7. The `Net::Ipv6` namespace can be used to match IPv6 addresses, as well as the CIDR notation.

8. The `Net::IpvFuture` namespace can be used to match any IPvFuture address.

9. The `Net::Rfc1035` namespace contains an implementation of the domain name format descibed in rfc1035 [13].

10. The `Net::Rfc1235` namespace contains an implementation of the updated domain name format descibed in rfc1123 [3].

11. The `Net::Uri` namespace implements the entire URI grammar described in rfc3986 [2]. It can be used to match URIs, web addresses or parts of URIs.

12. The `File::Csv` namespace contains a CSV grammar adapted from rfc4180 [15]. It contains a parameterized non-terminal that accepts a custom field separator. Fields escaped with double quotes are also supported.

13. The `DateTime::Rfc3339` namespace contains a full grammar adapted from rfc3339 [10], which is a subset of the ISO 8601 date, time, interval and period notation.

14. The `DateTime::Informal` namespace contains a few grammars that can be helpful for matching informal date and time notation.

6.1.2 Structured output

Traditionally, tools like `grep` have been limited to only printing text as output. `peg-match` can optionally output structured JSON instead. This can be useful when using `peg-match` as part of a script, or for example when attempting to extract a certain column from a CSV file.

Example 47. Using the `File::Csv::CsvFile` non-terminal from the `File::Csv::` namespace, we can parse an entire CSV file at once. The CSV data can be output as JSON, which makes it much easier to manipulate. For example, the output could be piped into the tool `jq`, which can manipulate JSON files.

Without the structured output, we would have needed to write a small program that imports a CSV library, parses the file using the library, and outputs our desired format. In practice, this works as follows:

```
$> peg-match -j 'File::Csv::CsvFile' data.csv
{
  "records": [
    [
      "id",
      "name",
      "value"
    ],
    [
      "1",
      "Daniel",
      "60"
    ],
    [
      "2",
      "Mark",
      "20"
    ]
  ]
}
```

6.2 The Library

6.2.1 Overview

The library is split into 5 namespaces:

1. `Operators` - Classes needed to represent a PEG.
2. `Common` - Classes that are shared between multiple namespaces.
3. `Compilation` - Classes that are needed for the PEG compiler.
4. `Runner` - Classes that are needed to match a compiled PEG on an input string.
5. `SelfParser` - Classes that implement a PEG that can parse itself.

To get started quickly, the `PatternCompiler` class contains the boilerplate code needed for the most common use-case: compiling a pattern and building a runner. To match a PEG on an input string, all that is needed is the following:

```
var runner = PatternCompiler.Default.Compile(somePeg);
var result = runner.Run(inputString);
```

The `PatternCompiler` is a combination of the main three parts in the compilation pipeline of the library, the compiler, the optimizer and the jitter. The compiler is responsible for transforming a PEG, defined using the classes in `Operators` to a compiled program C (see Section 3.2.1). The optimizer performs any number of transformations on the compiled program C to make it faster or more efficient. The jitter is responsible for the last-minute conversion to a format that can be executed. This might be as simple as initializing an instance of an interpreter class, or it might be compilation to another language. For example, the `Runner.ILRunner.ILJitter` class will translate the instructions from our parsing machine to the .NET Common Intermediate Language (CIL).

All three components can be extended separately. For example, a new compiler could be written that directly translates regular expressions to the instruction set of our parsing machine, reusing the existing jitter. Similarly a new jitter could be written that, for example, skips the CIL generation (described in Section 6.2.6) and directly outputs x86 assembly, while re-using the existing compilers. Lastly, optimizations can be extended at two points. An individual optimization can be added by adding a class that extends `Optimizations.BaseOptimization` to the list of optimizations. If the `Optimizations.Default.BaseOptimization` is too restricting, a class that implements `IOptimizer` can be added, which is only required to receive a program C and return a new, optimized C'.

6.2.2 Captures

When defining languages formally, we are only interested in whether an input string matches the PEG or regular expression. However, in the real world we often want to know more than just a boolean result indicating whether the string matched. Many regular expression libraries re-use the grouping parentheses as a 'capture'.

Example 48. When matching the regular expression `ab(c+)d` to an input string, 'abcccccd', the regular expression library will not only return that the expression matched the input string, but it can also return the c+ part of the regular expression separately.

When using a regular expression library to search and replace certain parts of an input text, the captures can also be used to copy certain parts of the input to the replacement. This is often denoted using '$N' or '\N', which refers to the Nth capture in the expression.

Our implementation has optional support for captures. Instead of re-using the grouping operator, we are using a separate operator, which looks like this: $e_1 \ \{ \ e_c \ \} \ e_2$, where e_1, e_2 are PEG expressions and e_c is the PEG expression that will be captured.

Captures are implemented using a new instruction, `Capture` N v K, where N is an offset, v a variable, and K a capture key. This instruction adds an entry $(K, V[v], i)$ to the capture list, where $V[v]$ is the value of the variable, and i is the current position in the input string. The capture key $K \in \mathbb{N}$ can be used to identify the capture afterwards, and has no meaning in and of itself.

According to our formal model of the machine, the instruction has no effect on the PEG matching process because our model does not take captures into account. Therefore, it made little sense to include the `Capture` instruction as part of the formal definition of the machine.

6.2.3 Representing PEGs

PEGs are represented using the classes inside the `Operators` namespace. Each non-abstract class represents a single operator. Only the essential operators are implemented. Syntactic sugar is available in the form of static methods in the `Operator` class. The following operators are defined:

1. `Any`, the any operator (.)
2. `CharacterClass`, a terminal character, extended to optionally match a set of characters.
3. `Empty`, the empty string (ε)
4. `Not`, the any operator (.)
5. `Pattern`, a non-terminal (A_i)
6. `PrioritizedChoice`, the prioritized choice operator (e_1/e_2)
7. `Sequence`, the concatenation operator (e_1e_2)
8. `ZeroOrMore`, the repeat operator ($e*$)
9. `CaptureGroup`, an operator which will add a capture entry for everything that is matched inside (e)

These operators all inherit from the abstract `Operator` class, and if the operators are not atomic they inherit from the `SingleChildOperator` (`Not`, `ZeroOrMore`, `CaptureGroup`) or `MultiChildOperator` (`Sequence`, `PrioritizedChoice`) as well.

Example 49. Consider the expression `'a' / (!('bc'+))`. We represent this as follows:

```
new PrioritizedChoice(new CharacterClass('a'), new Not(Operator.OneOrMore(
    CharacterClass.String("bc"))))
```

Due to the verbosity, these declarations are hard to read. We have implemented implicit conversion from characters and strings to their respective operators, so that the above expression can also be declared like this:

```
new PrioritizedChoice('a', new Not(Operator.OneOrMore("bc")))}
```

However, this is still very verbose. Therefore, we have also implemented a PEG parser that can parse a PEG from a string input, which we describe in Section 6.2.8.

6.2.4 Compilation

Compilation of the PEGs is handled in the `Compilation.Compiler` class. The compiler only takes a start non-terminal ('pattern'). We start by building `PatternInfo` for the start non-terminals, to discover all the non-terminals we will need to compile. We also keep track of whether non-terminals are recursive and how big non-terminals are. We use this info to inline extremely small non-terminals, to avoid the method invocation overhead. Currently the compiler will inline non-terminals of 16 operators or less that do not contain any non-terminals.

Compilation itself is very straight-forward, and implemented in about 200 lines of code. We take a different approach for reducing the number of advances than described in Section 5.1.2. Instead of pre-calculating the prefix size (which would take a recursive iteration over all child nodes every time), we generate a `BoundsCheck` and keep track of its position, so we can modify that instruction afterwards to patch the offset.

6.2.5 Optimization

An optimizer can be defined by implementing the `Optimizations.IOptimizer` interface. By default, the optimizations are handled by the `Optimizations.DefaultOptimizer` class. This class can be instantiated with a list of optimizations, which inherit from `Optimizations.Default.BaseOptimization`. When optimizing it will repeatedly run all optimizations sequentially, until the instruction sequence reaches a 'stable' point where no optimizations are changing it anymore.

6.2.6 Code generation and execution

A code generator can be implemented by adding a class that implements `IJitter`. The `IJitter` interface requires a `Compile` method to be implemented, which can translate the `CompiledPeg` to an `IRunner`. The `IRunner` interface defines an interface for matching a string. The jitter can either return an instantiation of an existing class, or dynamically generate a new class and return an instantiation of that class.

CIL generation

The conversion from our instruction set to the Common Intermediate Language (CIL, formerly called Microsoft Intermediate Language) is done by the `ILJitter` class. This class translates instructions one-by-one to equivalent CIL opcodes.

The implementation translates each non-terminal to a separate method, and adds one additional method, `char* RunInternal(char*)`, which invokes the start non-terminal. We use unsafe pointer arithmetic to avoid array bounds checks. This makes the methods incompatible with the `IRunner` interface. Therefore, we have added a class, `BaseJitted-Runner`, which implements the `IRunner` interface, and sets up fields that are used by the dynamically generated code. The class that's generated dynamically inherits from the `BaseJittedRunner`, so that we can avoid generating boilerplate code for interfacing with `IRunner` at runtime.

Interpreter

The `InterpreterRunner` class implements a simple interpreter for our parsing machine. Instructions are directly fetched from the instruction list, and then executed using one big `switch` statement. Positions of the `MarkLabel` instructions are pre-calculated when the class is instantiated, so that jumping to a label consists of just 2 array lookups.

The implementation of the intepreter is not optimized for speed. Our parsing machine is mostly designed to be compiled to machine code, which does not need an interpreter to be executed. Therefore, we do not expect the interpreter to be used in general. Instead, its intention is to be as reliable as possible so it can be used as a debugging tool for the parsing machine. There are often limited to no debugging symbols available for dynamically compiled code, like the code generated by the `ILJitter`, which makes debugging the parsing machine difficult. Since the interpreter does not generate any code dynamically, debugging is much easier.

6.2.7 Captures

The `IRunner` can, besides returning whether a match was successful, also output a list of all captures using the `Runner.Capture` class. This class contains fields that mark the beginning and the end position of the captured string, as well as the associated capture key. The class also contains a field `CaptureCloseIndex`, which indicates how far in the matching process the capture was added. This can be useful when determining the capture order of two nested captures that matched the same part of the string.

Example 50. We'll use {`name:expression`} to indicate a capture group.

Consider the expression {`outer:`{`inner:'abc'`}}. If mached on the input string "abc", we'll receive two captures: outer and inner, both from position 0 to position 2 in the input string. Without looking at the expression, we cannot determine which capture group was captured first. However, when looking at the `CaptureCloseIndex`, the `CaptureCloseIndex` of outer will be greater than the `CaptureCloseIndex` of inner, so we know that inner must have matched before outer.

6.2.8 The self-parser

The `SelfParser.PegGrammar` class provides an extensible PEG grammar definition. The class itself only implements the grammar as defined by Ford [7]. All our extensions are defined separately in the `peg-match` tool.

The grammar contains capture operators with distinct capture keys for each operator or construct in the PEG grammar. We then match our grammar to the input string, which will return a list of all captures. Using these captures, we build the PEG expression in a bottom-up fashion.

To extend the parser, a class that inherits from the parser can be implemented. This class can then modify any of the existing non-terminals, which are all stored in class fields. The class can also hook into the method that builds an expression from the captures by overriding it. This allows for virtually unlimited extensibility.

Chapter 7

Related Work and Conclusions

Our own work has mainly been focussed on the parsing machine and the PEG extensions. We use the same definitions for `match` as used by Ierusalimschy and Medeiros. We also use similar notation when describing state transtitions, states, instructions and the PEG compilation.

Due to the very different instruction design of our parsing machine, most optimizations were designed from scratch. We do implement character classes (Section 5.1.1), an optimization also implemented by Ierusalimschy and Medeiros. The PEG extensions and the implementation are our own work.

7.1 Regular expressions

7.1.1 Benchmarks

Table 7.1 shows a comparision of different regex runtimes and SharpPEG, our implementation. The numbers listed are the best time (in ms) out of 50 runs. Each run consists of finding all occurrences of the pattern in all works of Mark Twain, which is available in the public domain[1]. The benchmarks were run on a PC running Windows 10, build 1703, with 16 GB of 1600Mhz DDR3 RAM and an Intel Core i7-4770K, clocked at 3.5Ghz. This benchmark has been adapted from a regular expressions benchmark [14], which is an updated version of a benchmark that was used to show the performance improvements in the PCRE JIT compiler.

The following regex libraries were benchmarked:

1. PCRE2 [2]
2. tre [3]
3. Oniguruma [4]
4. re2 [5]
5. Hyperscan (hscan) [6]

[1]http://www.gutenberg.org/ebooks/3200
[2]http://www.pcre.org/
[3]https://github.com/laurikari/tre
[4]https://github.com/kkos/oniguruma
[5]https://github.com/kkos/oniguruma
[6]https://github.com/01org/hyperscan

6. Rust regex crate (rust_regex) [7]

We benchmarked the following regular expressions:

1. B1: `Twain`
2. B2: `[a-z]shing`
3. B3: `Huck[a-zA-Z]+|Saw[a-zA-Z]+`
4. B4: `Tom|Sawyer|Huckleberry|Finn`
5. B5: `.{0,2}(Tom|Sawyer|Huckleberry|Finn)`
6. B6: `.{2,4}(Tom|Sawyer|Huckleberry|Finn)`
7. B7: `Tom.{10,25}river|river.{10,25}Tom`
8. B8: `[a-zA-Z]+ing`
9. B9: `([A-Za-z]awyer|[A-Za-z]inn)\s`

	SharpPEG	pcre2	pcre2-dfa	pcre-jit	re2	onig	tre	hscan	rust_regex
B1	9	3	9	12	2	13	195	1	2
B2	16	332	539	12	68	12	277	3	5
B3	13	16	17	2	27	31	340	2	4
B4	13	20	21	19	29	35	623	2	33
B5	35	3464	2531	233	34	64	1612	2	32
B6	43	3440	3035	265	34	65	2437	3	32
B7	20	49	62	12	36	62	362	2	23
B8	105	786	1212	54	82	613	359	12	13
B9	30	713	769	27	66	136	653	4	33

Table 7.1: Performance comparison of regular expression libraries and SharpPEG

The regular expressions in the table were translated to PEGs before running them in our solution, using the conversion algorithm of [12]. We have manually checked the correctness of these translations, and included them in Appendix A.

7.1.2 Benchmarking issues

There are a number of possible issues with the benchmarks. Most importantly, most of the other PEG libraries operate directly on the raw UTF-8 formatted string data. This did not make sense for our implementation, as strings in C# are always arrays of UTF-16 data. While using UTF-16 requires processing twice as much memory, UTF-8 contains continuation bytes which means that advancing one character is more complex than incrementing a pointer by 1.

[7]https://doc.rust-lang.org/regex/regex/index.html

Another issue is the fact that we are comparing regular expression libraries to a PEG library. While this is necessary to establish a rough comparison between our implementation, which is aimed to be a replacement for regular expressions, a direct translation from regular expression to PEG might not always yield the best PEG.

One last issue is that some of the regular expression libraries seem to be using tricks that defeat the point of the benchmarks. For example, the minimal time needed to iterate over every character in the input string seems to be around 5-8ms. As can be seen in Table 7.1, the Hyperscan library is able to match most expressions in just 1-2ms. Hyperscan is using many special string searching algorithms that happen to be applicable to all our benchmarks. The point can be made that this defeats the goals of the benchmarks, as we are comparing general-purpose pattern matching libaries to a large collection of highly specific string searching algorithms.

7.2 LPeg

Our new parsing machine and implementation improve upon the work done by Ierusalimschy and Medeiros [11].

1. Our Parsing Machine is suitable for JIT compilation, which eliminates all interpreter overhead and brings performance on par with other regular expression matching libraries
2. Our PEG extensions make PEGs easier to use for pattern matching. In particular, we added the following:
   (a) Parameterized non-terminals solve the absence of native operators like the 'a ending with b' operator, which can be implemented in regular expressions using `a+ b`
   (b) Fixed repeat operators allow for conciser definitions of repetitive patterns
   (c) Namespaces add proper support for PEG libraries, allowing code to be re-used
3. Our smaller instruction set reduces the work required to implement the parsing machine

7.2.1 Benchmarks

While the LPeg implementation was benchmarked by the authors [8], they unfortunately did not provide the code they used to benchmark their implementation. Therefore, instead of trying to replicate their benchmarking setup we implemented the above benchmark for LPeg too. The benchmarking results are listed in Table 7.2.

Note that our interpreter is not aimed to be as fast as possible, as described in Section 6.2.6.

Unfortunately benchmark B7 did not function properly, and returned incorrect results. We have not been able to determine whether this is the result of an error on our end, or a bug in LPeg.

7.2.2 Instruction design

The LPeg Parsing Machine requires at least 8 instructions, but to reach full performance 19 instructions are required. Our implementation uses only 9 instructions, and does not

	SharpPEG (UJ)	SharpPEG (OI)	SharpPEG (OJ)	LPeg
B1	99	404	9	114
B2	138	401	16	204
B3	102	553	13	115
B4	119	1143	13	113
B5	209	2823	35	739
B6	213	3136	43	725
B7	106	597	20	ERR (10401)
B8	193	2123	105	959
B9	171	628	30	302

Table 7.2: Performance comparison SharpPEG and LPeg. UJ = unoptimized, jitted, OI = optimized, interpreted, OJ = optimized, jitted

require more instructions to increase performance. Instead, performance is increased by altering the sequence of instructions, removing unneeded instructions and re-ordering existing instructions.

While the difference in the number of instructions might seem to indicate that one solution is better than the other, this is not the case. The difference mostly highlights the different design goals of the libraries: LPeg is implemented as an interpreter, which introduces a fixed overhead to each instruction that's executed, so having many complex instructions can be faster than having fewer smaller instructions. Our implementation on the other hand, is meant to be JIT-compiled to machine code, which does not have any overhead for executing an instruction. Therefore, having fewer instructions is beneficial, as it reduces the number of possible instructions the optimizations have to deal with.

7.3 PACKRAT parsing

PACKRAT parsing [6], which uses memoization, is needed for guaranteed linear parsing of any language. However, we claim that for the most common pattern matching expressions we do not need memoization to achieve linear run time. Since our machine is built to be a replacement for regular expressions, most expressions will be simple and often use only a single non-terminal, which renders memoization useless.

In our benchmark, there is only one pattern that could benefit from memoization, namely: `[a-zA-Z]+ing`. Since this pattern relies on the backtracking behaviour of regular expressions, we need to translate it to a more complex pattern in PEGs. Our tool `peg-match` contains a parameterized non-terminal to simplify this process, where we are translating as follows:

```
Until<Repeat, End> <- (!(!(!End) !(Repeat (!End Repeat)* End)) Repeat)* End
```

We could also translate this to a recursive solution:

```
Until<Repeat, End> <- Repeat Until<Repeat, End> / End
```

To determine whether PACKRAT could help improve run-time performance, we have implemented memoization in our implementation, which can be toggled by setting a flag in the `ILJitter`. Benchmarking the two different versions shows that this does not improve performance, as can be seen in Table 7.3.

Benchmark	Non-recursive	Recursive (not memoized)	Recursive (memoized)
`[a-zA-Z]+ing`	105	228	310

Table 7.3: Memoization versus no memoization versus a non-recursive alternative

These results are most likely the result of two factors:

1. Method invocation is relatively costly. A new stack frame needs to be allocated, and registers must be temporarily stored on the stack. Upon returning, a stack frame needs to be removed and any stored registers need to be loaded back in. This in and of itself is relatively costly.

2. When there exists a non-recursive solution for a particular expression, the additional memory look-up in the recursive variant caused by the memoization is just as slow as the overhead caused by the non-recursive solution.

Bibliography

[1] Atwood, J. (2005). Regex use vs. regex abuse. [accessed 22-November-2017]. https://blog.codinghorror.com/regex-use-vs-regex-abuse/.

[2] Berners-Lee, T., Fielding, R., and Masinter, L. (2005). Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 (Internet Standard). Updated by RFCs 6874, 7320. https://www.rfc-editor.org/rfc/rfc3986.txt.

[3] Braden, R. (1989). Requirements for Internet Hosts - Application and Support. RFC 1123 (Internet Standard). Updated by RFCs 1349, 2181, 5321, 5966, 7766. https://www.rfc-editor.org/rfc/rfc1123.txt.

[4] Cousot, P. and Cousot, R. (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA. ACM.

[5] Crocker, D. and Overell, P. (2008). Augmented BNF for Syntax Specifications: ABNF. RFC 5234 (Internet Standard). Updated by RFC 7405. https://www.rfc-editor.org/rfc/rfc5234.txt.

[6] Ford, B. (2002). Packrat parsing:: Simple, powerful, lazy, linear time, functional pearl. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 36–47, New York, NY, USA. ACM.

[7] Ford, B. (2004). Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 111–122, New York, NY, USA. ACM.

[8] Ierusalimschy, R. (2009). A text pattern-matching tool based on parsing expression grammars. *Softw. Pract. Exper.*, 39(3):221–258.

[9] Kleene, S. C. (1956). Representation of events in nerve nets and finite automata. In Shannon, C. and McCarthy, J., editors, *Automata Studies*, pages 3–41. Princeton University Press, Princeton, NJ.

[10] Klyne, G. and Newman, C. (2002). Date and Time on the Internet: Timestamps. RFC 3339 (Proposed Standard). https://www.rfc-editor.org/rfc/rfc3339.txt.

[11] Medeiros, S. and Ierusalimschy, R. (2008). A parsing machine for pegs. In *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, pages 2:1–2:12, New York, NY, USA. ACM.

[12] Medeiros, S., Mascarenhas, F., and Ierusalimschy, R. (2014). From regexes to parsing expression grammars. *Sci. Comput. Program.*, 93:3–18.

[13] Mockapetris, P. (1987). Domain names - implementation and specification. RFC 1035 (Internet Standard). Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2673, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966, 6604, 7766. https://www.rfc-editor.org/rfc/rfc1035.txt.

[14] Schmidt, D. (2017). A comparison of regex engines. [accessed 11-December-2017]. https://rust-leipzig.github.io/regex/2017/03/28/comparison-of-regex-engines/.

[15] Shafranovich, Y. (2005). Common Format and MIME Type for Comma-Separated Values (CSV) Files. RFC 4180 (Informational). Updated by RFC 7111. https://www.rfc-editor.org/rfc/rfc4180.txt.

Appendix A

Converted regular expressions

The benchmarks used an experimental implementation of the regex conversion method intruduced in a paper by Medeiros et al. [12]. However, the implementation of the conversion is still very experimental. As such, we include the translated and de-sugared PEGs used in the benchmarks here, so that correctness can be verified manually.

We have replaced all occurences of '`!([\n-\r]) .`' with $\boxed{\text{s}}$ to improve readability. These $\boxed{\text{s}}$'s are a side-effect of the meaning of '.' (the any operator). Because of historical reasons, in virtually all regex-matching libraries a '.' matches any character except newline characters, instead of just matching any character.

Regex	PEG
Twain	'Twain'
[a-z]shing	[a-z] 'shing'
Huck[a-zA-Z]+|Saw[a-zA-Z]+	('H' ('u' ('c' ('k' ([a-zA-Z] [a-zA-Z]*))))) / ('S' ('a' ('w' ([a-zA-Z] [a-zA-Z]*))))
Tom|Sawyer|Huckleberry|Finn	'Tom' / 'Sawyer' / 'Huckleberry' / 'Finn'
.{0,2}(Tom|Sawyer|Huckleberry|Finn)	e ((($\boxed{s}$ $\boxed{s}$) ('Tom' / 'Sawyer' / 'Huckleberry' / 'Finn')) / ($\boxed{s}$ ('Tom' / 'Sawyer' / 'Huckleberry' / 'Finn')) / 'Tom' / 'Sawyer' / 'Huckleberry' / 'Finn')
.{2,4}(Tom|Sawyer|Huckleberry|Finn)	($\boxed{s}$ $\boxed{s}$) ((($\boxed{s}$ $\boxed{s}$) ('Tom' / 'Sawyer' / 'Huckleberry' / 'Finn')) / ($\boxed{s}$ ('Tom' / 'Sawyer' / 'Huckleberry' / 'Finn')) / 'Tom' / 'Sawyer' / 'Huckleberry' / 'Finn')
Tom.{10,25}river|river.{10,25}Tom	('T' ('o' ('m' (($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ $\boxed{s}$))))))))) ((($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ $\boxed{s}$)))))))))))))) 'river') / (($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ $\boxed{s}$))))))))))))) 'river') / (($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ $\boxed{s}$)))))))))))) 'river') / (($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ $\boxed{s}$))))))))))) 'river') / (($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ $\boxed{s}$)))))))))) 'river') / (($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ $\boxed{s}$))))))))) 'river') / (($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ $\boxed{s}$)))))))) 'river') / (($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ $\boxed{s}$))))))) 'river') / (($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ $\boxed{s}$)))))) 'river') / (($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ $\boxed{s}$))))) 'river') / (($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ $\boxed{s}$)))) 'river') / (($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ $\boxed{s}$))) 'river') / (($\boxed{s}$ ($\boxed{s}$ $\boxed{s}$)) 'river') / (($\boxed{s}$ $\boxed{s}$) 'river') / ($\boxed{s}$ 'river') / 'river'))))) / ('r' ('i' ('v' ('e' ('r' (($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ $\boxed{s}$))))))))) ((($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ $\boxed{s}$)))))))))))))) 'Tom') / (($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ $\boxed{s}$))))))))))))) 'Tom') / (($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ $\boxed{s}$)))))))))))) 'Tom') / (($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ $\boxed{s}$))))))))))) 'Tom') / (($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ $\boxed{s}$)))))))))) 'Tom') / (($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ $\boxed{s}$))))))))) 'Tom') / (($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ $\boxed{s}$)))))))) 'Tom') / (($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ $\boxed{s}$))))))) 'Tom') / (($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ $\boxed{s}$)))))) 'Tom') / (($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ $\boxed{s}$))))) 'Tom') / (($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ $\boxed{s}$)))) 'Tom') / (($\boxed{s}$ ($\boxed{s}$ ($\boxed{s}$ $\boxed{s}$))) 'Tom') / (($\boxed{s}$ ($\boxed{s}$ $\boxed{s}$)) 'Tom') / (($\boxed{s}$ $\boxed{s}$) 'Tom') / ($\boxed{s}$ 'Tom') / 'Tom')))))))
[a-zA-Z]+ing	[a-zA-Z] (((!((!(!'ing')) (!([a-zA-Z] (((!'ing') [a-zA-Z])* 'ing'))))) [a-zA-Z])* 'ing')
([A-Za-z]awyer|[A-Za-z]inn)\s	([A-Za-z] ('a' ('w' ('y' ('e' ('r' [\u0009-\u000A\u000C-\u000D])))))) / ([A-Za-z] ('i' ('n' ('n' [\u0009-\u000A\u000C-\u000D]))))