An RPython JIT for Parsing Expression Grammars

Bachelor's Thesis

by

Stefan Karl Troost

born in

Nettetal

submitted to

Professorship for Software Engineering and Programming Languages
Prof. Dr. Michael Leuschel
Heinrich-Heine-University Düsseldorf

Supervisor:
Dr. Carl Friedrich Bolz

Abstract

PEGs in general and it's Lua implementation LPeg in particular have been proposed as more expressible alternatives to regular expressions. However, the implementation of LPeg as a bytecode based interpreter written in C can be limiting for some applications. In this thesis, the LPeg interpreter was re-implemented in the RPython language of the PyPy project. The RPython language then automatically generates a JIT compiler for LPeg patterns running on this interpreter. With the help of this JIT the performance of string matching using LPeg is greatly improved and a number of limitations of the C implementation are lifted.

Acknowledgments

A lot of people supported me during my work on this thesis to whom I wish to express my gratitude.

First and foremost i wish to thank Prof. Dr. M. Leuschel for accepting my bachelor thesis application and supporting my request for a wisdom teeth related extension of my deadline. In the same breath i thank Dr. Carl Friedrich Bolz for burning through hours of his life to supervise me and Dr. John Witulski for teaching me how compilers work, and introducing me to Dr. Carl Friedrich Bolz.

Of course I wish to thank my family, Karl-Rainer & Helene Troost for creating me and Alexander Johannes Troost for proof-reading and being my favorite brother.

Finally, I wish to thank Silke Schroers for keeping me sane, Robin Rawiel for suggesting in 2011 that i should learn python, and Simon Krücken for proof-reading.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Text pattern matching is a task demanded in a wide variety of fields including but not limited to data validation, data scraping, parsing and the production of syntax highlighting. The most common way to approach this task is by using regular expressions to define a pattern that can be searched by a pattern matching engine like grep, AWK, sed or PCRE. However, regular expressions come with a plethora of limitations, such as being hard to read and consequently hard to maintain, having varying syntax standards and being unable to recognize more complex patterns, for example the problem of balanced parentheses [Rei11, chapter 1.15].

To adress these limitations, parsing expression grammars (PEGs) have been developed as a viable alternative to regular expressions. Unlike regular expressions, PEGs resemble context free grammars in terms of syntax and expression power, yet they avoid the issue of ambiguity by introducing an ordered choice operator [For04]. LPeg exists as an implementation of PEGs, which compiles high level patterns into a bytecode-based language that is then executed using a C-based interpreter [Ier09].

Since similar implementations of regular expressions often use just-in-time compilers to improve performance [noac], it appears reasonable to do the same for LPeg. The RPython-toolchain was developed by researchers of the PyPy team and allows a bytecode interpreter written in a statically typed subset of python called RPython to be converted to a tracing JIT through the addition of few so called jit-hints [BCFR09]. By re-implementing the LPeg interpreter in RPython a JIT was implemented that interprets LPeg's bytecode. To further improve performance, certain optimizations that are specific to the interaction between RPython's tracing JIT and LPeg's bytecode were introduced. These optimizations combined with the

tracing JIT significantly boost performance given sufficiently large filesizes.

The remainder of this thesis is structured as follows.

2: Background The RPython toolchain, parsing expression grammars, LPeg and the virtual machine (VM) used to interpret LPeg's bytecode are explained in detail.

3: The JitPeg Interpreter The implementation details of the interpreter are presented in this chapter.

4: Optimizing JitPeg Optimizations of the RPython-JIT in regards to runtime are listed and elaborated.

5: Results Various benchmarks demonstrating the effectiveness of the tracing JIT and the optimizations are displayed. Finally, conclusions are drawn.

Chapter 2

Background

2.1 The RPython Toolchain

The PyPy project [noaf] is an ongoing international research effort, with the goal to develop a re-implementation of the Python programming language that is more performant relative to the standard implementation, CPython [RP06].

As part of that research project the PyPy team has developed the RPython toolchain, which allows for code that is written in a static subset of Python, called RPython, to be translated to C-code which is in turn compilable [AACM07].

Additionally, the RPython toolchain includes a just-in-time compiler generator [Bol14], which allows for the user to add certain JIT hints [BCF+11] to an interpreter written in the RPython language, which is then translated to a tracing JIT, significantly boosting performance.

2.1.1 JITs vs. Interpreters

Both JITs and interpreters serve the same purpose: to allow the execution of programming code that is not native to the hardware. In the case of the LPeg interpreter, this is performed by parsing the high-level language into an intermediate representation (bytecode), and executing this bytecode directly through interpretation. A JIT does the same thing, but with the added ability to compile parts of the program that are frequently executed to machine code

at runtime. In the case of a tracing JIT, it decides to do so by detecting (tracing) frequently iterated loops of the program. Therefore, a tracing JIT operates on the assumption that a program spends most of it's runtime in loops [BCFR09].

Since tracing a program's execution and compiling parts of it during execution is associated with a cost in computing time, one would expect a JIT to initially perform slower than an equivalent interpreter. Nevertheless, replacing parts of a program with equivalent machine code yields significant benefits in terms of performance. We would expect a JIT to have significantly better performance than an equivalent interpreter given enough time of a given program is spent in loops, so that the time cost of tracing a program's execution and generating machine code on the fly is outweighed by the time benefit of executing machine code as opposed to having to interpret the corresponding bytecode.

2.2 Parsing Expression Grammars

Parsing Expression Grammars (PEGs) [For04] are a type of formal grammar similar to context-free grammars (CFGs), meaning that a PEG is a set of production rules describing words in a formal language. The difference between PEGs and CFGs is that PEGs are unambiguous. This is achieved by redefining the ambiguous choice operator of CFGs (usually noted as `|`) as an `ordered` choice operator. In practice this means that if a rule in a PEG presents a choice, a PEG parser prioritizes the leftmost choice. Practical uses for PEGs include parsing and pattern-searching. In comparison to regular expressions, PEGs stand out as being able to be parsed in linear time, being strictly more powerful than regular expressions, as well as being arguably more readable.

2.3 LPeg

LPeg [Ier09] is an implementation of PEGs written in C to be used in the Lua programming language. A crucial detail of this implementation is that it parses high level function calls, translates them to bytecode and interprets that bytecode. Therefore, we can try to improve LPeg by replacing the C-interpreter with an RPython-JIT. To achieve this, a modified version

of LPeg is used to parse PEGs and pass the generated Intermediate Representation, the LPeg bytecode, to the JitPeg interpreter.

2.3.1 LPeg Patterns

The LPeg interpreter executes bytecodes created by parsing a string of commands using the LPeg library. The optimized RPython-JIT supports a subset of the LPeg library, with some of the more advanced or obscure features being cut. The table 2.1 shows the basic operations used to create patterns in LPeg.

Table 2.1: Supported Operations for lpeg patterns [noab]

Operator	Description
`lpeg.P(string)`	Matches string literally
`lpeg.P(n)`	Matches exactly n characters
`lpeg.P(-n)`	Matches at most n characters
`lpeg.S(string)`	Matches any character in string (Set)
`lpeg.R("xy")`	Matches any character between (inclusive) x and y (Range)
`pattern^n`	Matches at least n repetitions of pattern
`pattern^-n`	Matches at most n repetitions of pattern
`pattern1 * pattern2`	Matches pattern1 followed by pattern2
`pattern1 + pattern2`	Matches pattern1 or pattern2 (ordered choice)
`pattern1 - pattern2`	Matches pattern1 if pattern2 does not match
`-pattern`	Equivalent to ("" - pattern)

To extract semantic information from a pattern, captures are needed. A large set of capture operations that LPeg offers are not supported, most notably `lpeg.Cf` and the `/`-operator, which allow for a user-defined function to be applied to all captures of a pattern. The following table shows the operations supported for capture creation.

Operation	What it produces
`lpeg.C(pattern)`	the match for the pattern plus all captures made by pattern
`lpeg.Cp()`	the current position (matches the empty string)

Any LPeg pattern can be used in a rule with the syntax `>rule< = >pattern<`. The `lpeg.V(>rulename<)` operator is used to reference a rule. Figure 2.1 shows the use of grammar rules, captures and patterns to create a pattern that finds URLs in a given string. These patterns are translated into bytecode by the LPeg parser, which can be executed in the JitPeg VM. The full list of supported LPeg bytecode can be found in section 2.4.1.

Figure 2.1: LPeg search pattern for URLs

```
lpeg.P{                                                                    1
"S";                                                                       2
S = lpeg.V("URL") + lpeg.P(1) * lpeg.V("S"),                               3

URL = lpeg.Cp()*lpeg.C(lpeg.P("http") * lpeg.V("urlchar")^3),              5

urlchar = lpeg.R("az","AZ","09") + lpeg.S("-._~:/?#@!$&*+,;=")}^0          7
```

2.4 The VM

The state of the VM at any point in time is defined by the following variables:

- The program counter: The program counter (`PC`) is a whole number representing the index of the next instruction to be executed.
- The string index: The string index (`index`) is a whole number representing the current position in the inputstring
- The stack: The stack is a data structure that may get filled during bytecode execution with two kinds of entries: return addresses, which are used for function calls, and choice points, which are used for backtracking.
- The capture stack: The capture stack is a stack of capture objects, which are used to extract semantic meaning from a given inputstring.
- The fail boolean: A boolean value indicating that the VM has failed to match a pattern, which triggers backtracking.

These variables are manipulated through the execution of bytecode which allows the VM to find the corresponding high level pattern.

2.4.1 Supported LPeg Bytecodes

The supported LPeg Bytecode consists of 4 Categories:

Character Matching Bytecode

These bytecodes advance `PC` and `index` under certain conditions, otherwise they may set VM to fail state.

`any`: Checks if there is any characters left in the inputstring. If it succeeds it advances the `index` and `PC` by 1, if not the bytecode fails, causing the VM to attempt to backtrack.

`char c`: Checks if there is a character left in the inputstring, and if that character is equal to `c`. If it succeeds it advances the `index` and `PC` by 1, if not the bytecode fails.

`set X`: Checks if there is a character left in the inputstring, and if the current character is in the set `X`. If it succeeds it advances the `index` and `PC` by 1, if not the bytecode fails. A set is the union of a number of character ranges of at least size 1.

`span X`: While the current character is in the set X, the `index` is advanced by 1. Then advances `PC`. Has no defined fail case.

Program Flow Bytecode

These bytecodes may set the `PC` to arbitrary values.

`testany n`: This is a lookahead bytecode. If there is any character left in the inputstring it advances the `PC` but not the `index`. Otherwise it jumps to `n`.

`testchar c n`: This is a lookahead bytecode. If there is any character left in the inputstring and that character is equal to `c` it advances the `PC` but not the `index`. Otherwise it jumps to `n`.

`testset X n`: This is a lookahead bytecode. If there is any character left in the inputstring and that character is in the set `X`, it advances the `PC` but not the `index`. Otherwise it jumps to `n`.

`call n`: Puts a return address (the current `PC` + 1) on the stack and sets the `PC` to `n`. Has no defined failure case.

`ret`: Opposite of call. Removes the top value of the stack (if the string of bytecodes is valid this will always be a return address) and sets the `PC` to the removed value. Has no defined failure case.

`jmp n`: sets `PC` to `n`

Bytecodes that support backtracking

These bytecodes create and consume choice points that determine the VMs behavior in case it is set into the fail state.

`choice n`: Puts a choice point on the stack. Has no defined failure case.

`commit n`: Removes the top value of the stack (if the string of bytecodes is valid this will always be a choice point) and jumps to `n`. Has no defined failure case.

`partial commit n`: Modifies the top of the stack (if the string of bytecodes is valid this will always be a choice point). Sets the choicepoint's `index` and `capture` values to the current `index` and `capture` values, then jumps to `n`. Has no defined failure case.

`fail`: Always fails, causing the VM to backtrack.

`failtwice`: removes the top of the stack (if the string of bytecodes is valid this will always be a choice point), then fails, causing the VM to backtrack.

`end`: Terminates the VM's execution, returning captures.

Capture Bytecode

These bytecodes capture semantic information of the input and return it at the end of bytecode execution. The supported bytecode excludes a lot of the capture capabilities provided by LPeg.

`Fullcapture Position:` Pushes a positioncapture object with the current `index` value to the capture stack.

`Fullcapture Simple n:` Pushes a simplecapture object with the current `index` value and `size=n` to the capture stack.

`Opencapture Simple:` Pushes an open simplecapture object with the current `index` value and undetermined size to the capture stack.

`Closecapture:` Sets the top element of the capturestack to full and sets it's `size` value using the difference between the VM's current `index` and the `index` stored in the capture object.

2.4.2 The Stack

The stack is a data structure used by the VM to keep track of return addresses and choice points.

`return addresses` consist of a whole number intended to be written to the `PC` after a function call has returned. They are created by the `call` bytecode and removed by the `ret` bytecode or by backtracking.

`choice points` are used by the VM to restore itself to a previous state in case of backtracking. A choice point consists of values to overwrite the current `PC` and string `index`, as well as the capture stack at a previous state. Choice points are created by the `choice` bytecode, potentially modified by the `partial commit` bytecode and removed by the `commit` and `failtwice` bytecode or by backtracking.

2.4.3 The Fail State and Backtracking

If a bytecode has failed to match a character (`any`, `char`, `set`) or the `fail` or `failtwice` bytecodes were called, the VM is put into the fail state. In this state the VM attempts backtrack by removing entries from the stack until a choice point is found, which is then used to restore the VM to a non-failed state. This is achieved by overwriting the VM's values for the `PC`, `index` and captures with the ones specified by the choice point. This backtracking procedure

may fail if the stack is empty or filled solely with return addresses. In this case, the VM halts and any successful captures are returned.

2.4.4 Captures

The capture object is used to extract semantic meaning from a given inputstring. A capture object holds several values, depending on it's kind. Captures are created by the `opencapture` and `fullcapture` bytecode, potentially modified by the `closecapture` bytecode and removed by backtracking. After initialisation, capture objects are stored in a seperate stack structure. This seperate stack is called the `capture stack`. There are two supported kinds of capture, called simplecapture and positioncapture.

Simplecapture

A simplecapture object captures and returns a substring of the inputstring. To do so, a simplecapture object holds three values: a boolean `status` indicating if the capture is open or full, an integer `index` indicating at which position in the inputstring the desired substring is located, and an integer `size` representing the length of the substring.

There are two ways to create a simplecapture object:

`fullcapture simple (size=n)` creates a full simple capture object with an `index` equal to the VMs current `index` and a `size` equal to `n`.

`opencapture simple` creates an open simple capture object with an `index` equal to the VMs current `index`. The open attribute indicates that the capture's `size` value is indetermined

`closecapture` asserts that the current top element of the capture stack is an open simple capture object and sets it's `status` to full and it's `size` to the difference between the VM's current `index` and the capture's stored `index`.

This distinction between open and full captures is necessary, since full captures can only be used to capture patterns of fixed length.

Positioncapture

A position capture object captures and returns the VM's `index` at the time it was created. To create a positioncapture object, the `fullcapture position` bytecode is used. Open position captures do not exist, since position captures have no `size` value.

2.5 Examples

2.5.1 Backtracking example

The pattern that is used in the following example has the following syntax: `lpeg.P('ab') + lpeg.P('ad')`. Semantically, this means a pattern that accepts either exactly the string 'ab' or the string 'ad'. One valid translation of this pattern into bytecode is shown in figure 2.2

Figure 2.2: Bytecode for the pattern `lpeg.P('ab') + lpeg.P('ad')`

```
choice L1
char 'a'
char 'b'
commit
end
L1:char 'a'
char 'd'
end
```

By examining the execution of this string of bytecodes with the inputstring 'ad', one can see how the VM uses choice points to backtrack:

`choice L1`: A choice point is put on the stack with `index` = 0 and `PC` = L1.

`char 'a'`: The VM compares the current character in the inputstring to 'a'. It succeeds, so the `PC` and `index` are advanced by 1.

`char 'b'`: The VM compares the current character in the inputstring ('d') to 'b'. It fails, so the VM attempts to backtrack

The VM removes the choice point from the stack, and sets `index` to 0 and `PC` to L1.

`L1: char 'a'`: Since the VM's `index` has been restored to 0, this comparison succeeds, so the `PC` and `index` are advanced by 1.

`char 'd'`: The comparison succeeds, so `PC` and `index` are advanced by 1.

`end`: The VM halts.

2.5.2 Capture example

The following pattern is used to highlight the use of multiple kinds of capture objects in a pattern: `lpeg.C(lpeg.R('09')^1) * lpeg.Cp()`. Semantically, this pattern accepts a sequence of digits, using a simple capture to return the accepted digits, and a position capture to return the number of digits. Since the bytecode for this pattern does not include choice points, calls or jumps, it is executed in the order it is written. The following is an illustration of that program flow for the inputstring `"3141"`:

`opencapture simple`: An open simple capture object with `index` = 0 is created.

`set[('0'-'9')]`: The VM checks if the current character ('3') is between 0 and 9. It succeeds, so `PC` and `index` are incremented. This `set` bytecode is necessary to ensure that the inputstring hold at least 1 character.

`span[('0'-'9')]`: As long as the current character is between '0' and '9', the `index` is incremented. Thus, the remaining inputstring ("141") is consumed.

`closecapture`: The capture object on top of the capture stack is closed, and it's `size` value is set to 4.

`fullcapture position`: A positioncapture object with `index` = 4 is created and put on the capture stack.

`end`: The VM halts and returns the capture objects.

After the capture objects are decoded by the interpreter, the following output is produced:

```
3141

Position:4
```

Chapter 3

The JitPeg Interpreter

The JitPeg interpreter is a bytecode interpreter, meaning it implements a virtual machine capable of executing the bytecode. A core function of this virtual machine is the dispatch loop, which defines the behavior for all bytecodes and serves as the main-loop of the VM.

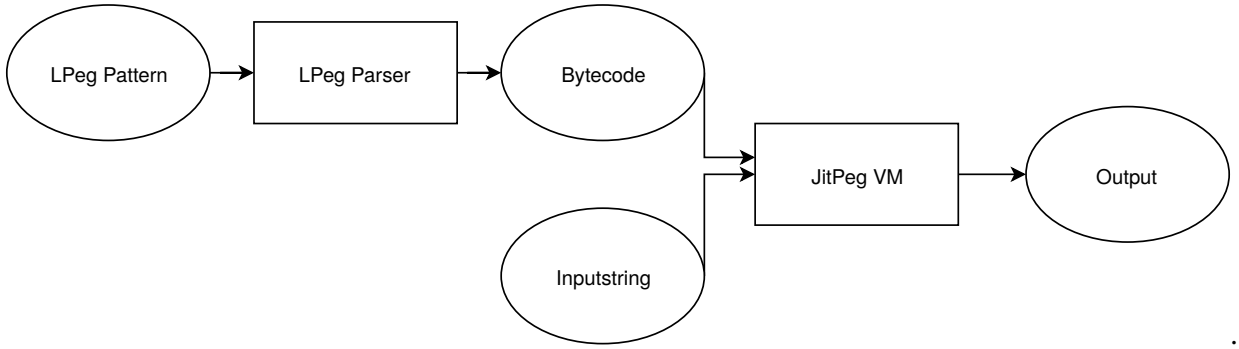
3.1 High level overview

The JitPeg interpreter works by using the LPeg Parser to parse an LPeg pattern into bytecode, and by executing that bytecode using an interpreter written in RPython, which includes a JIT. See figure 3.1 for an illustration of this process. The translation from a high level LPeg pattern into bytecode is achieved by using a debug feature of LPeg, which emits bytecode to the console via stdout. A simple parser takes that bytecode as a string as input and produces a Python-list of instruction objects as output, which is the final representation of the bytecode used by the JitPeg VM.

3.2 VM implementation

In this section, we will take a closer look at the RPython implementation of the VM described in section 2.4 by examining segments of the dispatch loop defining specific bytecodes, and by outlining the implementation of the VM's data structures.

Figure 3.1: Flow chart describing the outline of the JitPeg project



.

3.2.1 Bytecode examples

The bytecodes described in section 2.4.1 were implemented in an RPython-interpreter. By adding jit-hints, we were able to generate an efficient JIT. The following code examples show some implementations of bytecodes.

```
...
        elif instruction.name == "any":
            if index >= len(inputstring):
                fail = True
            else:
                pc += 1
                index += 1
...
```

The code for the any-bytecode is relatively straight-forward. It either advances the PC and index or sets the VM into the fail state, depending on whether the end of the inputstring has been reached or not.

```
...
        if instruction.name == "char":
            if index >= len(inputstring):
                fail = True
            elif instruction.character == inputstring[index]:
                pc += 1
                index += 1
            else:
                fail = True
...
```

The char-bytecode also looks as one would expect[1]. If the VM's string index is out of range or the character comparison fails, the VM is put into the fail state, otherwise the PC and index are advanced by 1. As you can see, the character the current inputstring is compared to is stored in the instruction object.

```
...
        elif instruction.name == "jmp":
            pc = instruction.goto
...
```

The jmp-bytecode includes a goto value which is used to set the PC value for the next instruction

```
...
        elif instruction.name == "choice":
            pc += 1
            choice_points = choice_points.push_choice_point(
                instruction.goto, index, captures)
...
```

As we can see here, the choice-bytecode puts a choice point onto the stack that may be backtracked to if the VM is in the fail-state. This choice point consists of a PC to jump to which is included with the choice-bytecode. It also holds the current index and captures values at the time the choice point was created.

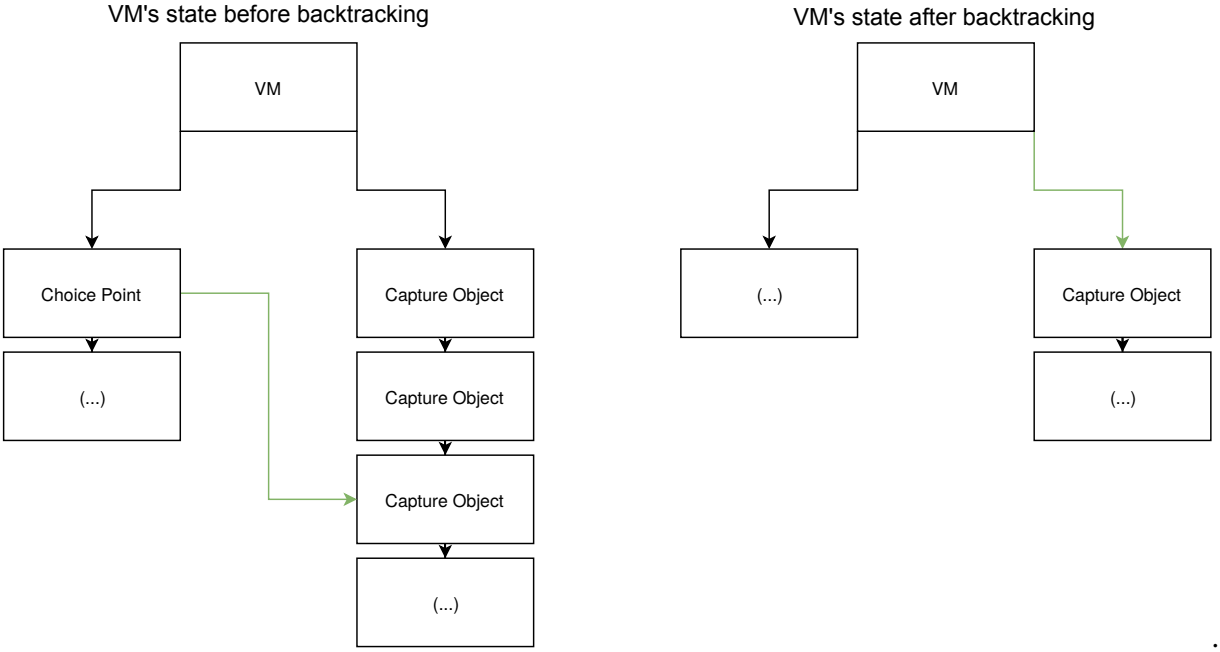
3.2.2 Capture Stack

The capture stack is implemented as a single linked list, meaning that each capture object holds a reference to the previous capture object in the stack. The advantage of this becomes clear if one considers the VM's behavior during backtracking: a choice point is removed from the stack to restore the VM's PC, index and capture stack. This storing and restoring of the capture stack works by writing a reference to the top of the capture stack into a given choice point, then during backtracking, setting the VM's top of the capture stack to

[1] The code example for the char-bytecode has been simplified for clarity, since the actual implementation includes a jit-optimization that allows the VM to execute multiple successive char-bytecodes at once.

the stored reference. Figure 3.2 shows an illustration of that process. This implementation proves to be efficient since restoring the `capture stack` is accomplished simply by moving a pointer.

Figure 3.2: Illustration of the backtracking process in relation to the capture stack



3.2.3 Backtracking Stack

The backtracking stack is built similarly to the `capture stack`, but is heavily optimized. As such, the stackentry objects of choice points and return addresses are merged into a single object using nested stacks and memoization techniques to improve performance. These optimizations are elaborated on in more detail in section 4.6.

Chapter 4

Optimizing JitPeg

4.1 Applying the JIT to the Interpreter

4.1.1 Jit Driver

The `JitDriver` object provided by the RPython-toolchain is what allows the JIT component to be added to the RPython interpreter. During initialisation of a jit driver, variables are classified into "green" and "red" variables. Green variables are those that are relevant to the generation of machine code, whereas red ones are not [BCFR09]. Figure 4.1 shows the source code for the main dispatch loop's JIT driver. The `PC` and `instructionlist` variables are obviously relevant for the generation of machine code. The `fail` boolean is relevant since the VM's behavior is independent of the `PC` and `instructionlist` if it is backtracking. The `flags` variable defines which JIT optimizations are active, which determines the implementation of some bytecodes.

Figure 4.1: Source Code for the Jit Driver (slightly simplified)

```
driver = jit.JitDriver(reds=["index", "inputstring",                    1
                             "choice_points", "captures"],              2
                       greens=["pc", "fail", "instructionlist",         3
                               "flags"])                                4
```

Figure 4.2: Source code for `jit_merge_point` and `can_enter_jit` hints (slightly simplified)

```
    while True:                                                              1
        instruction = instructionlist[jit.promote(pc)]                       2
        if instruction.isjumptarget and flags.jumptargets:                   3
            driver.can_enter_jit()                                           4
        driver.jit_merge_point()                                             5
```

4.1.2 Jit Hints

Two basic hints to add to a JIT are `jit_merge_point` and `can_enter_jit`. `jit_merge_point` is a hint that is necessary for the JIT to function. It indicates the beginning of the dispatch loop, allowing the JIT to return program control to the interpreter [noad].

The `can_enter_jit` hint marks the beginning of a potential loop. This allows the JIT to only search for beginnings of loops in bytecodes that fulfill the necessary criterium of being able to be jumped to. JitPeg finds those potential jumptargets by analyzing the bytecode during the parsing process and setting a boolean `isjumptarget`. Figure 4.2 shows the implementation of the conditional JIT hint `can_enter_jit` and the necessary hint `jit_merge_point`.

4.2 The CharRange object

A common task of the VM is to check if the current character is part of a set of consecutive characters (i.e. small letters a-z, capital letters A-Z, digits 0-9). A typical bytecode to fulfill this task is `set [('a'-'z')]`. The naive way to implement this functionality is to create a Python-list of characters specified in the `set` bytecode and use the built in `in`-keyword to check if the current character is in the set. This approach is potentially inefficient, since the runtime of that check is limited by the number of characters in the set. For sets of consecutive characters, JitPeg uses a more efficient approach: The `CharRange` object holds a minimum and a maximum value. It includes a method `is_match` that checks if the given parameter's ASCII value is between the minimum and maximum value. This way, the maximum number of checks performed is constant .

4.3 Branch-free check for set membership

As hinted at in section 4.2, set membership of a certain character is a frequently demanded task of the VM. To accomplish this, the VM must, in the general case, compare the desired character to others in a set. In the context of the tracing JIT, this creates the following problem: the naive approach would be to use the logical `or`-operator to perform such a comparison, which evaluates each statement in order and aborts (short-circuits) if a statement in the sequence evaluates to true. This short-circuiting, when traced by a JIT, creates code that includes many branches in the form of guards. Given that branch misprediction is a significant contributor to runtime [ESE06], a more sensible approach would be to use the bitwise `|`-operator, which evaluates all statements in a sequence and performs a bitwise or-operation on them. Figure 4.3 shows the branch-free code generated by this implementation, which can be executed extremely efficiently by modern pipelined out-of-order CPUs [HP11].

Figure 4.3: Trace for bytecode `set [("a"-"z","A"-"Z","0"-"9")]`

```
i8 = strgetitem(p1, i0)
i10 = int_le(97, i8)
i12 = int_le(i8, 122)
i13 = int_and(i10, i12)
i15 = int_le(65, i8)
i17 = int_le(i8, 90)
i18 = int_and(i15, i17)
i19 = int_or(i13, i18)
i21 = int_le(48, i8)
i23 = int_le(i8, 57)
i24 = int_and(i21, i23)
i25 = int_or(i19, i24)
guard_true(i25, descr=<Guard0x7f1f511d7a50>) [i4, i0, p3, p2, p1]
```

4.4 Generating Machine Code for the span Bytecode

Given the implementation specification of the `span` bytecode shown in 2.4.1, the implementation naturally leads to a while-loop. This poses a problem for the JIT, since the intuitive approach of implementing the `span` bytecode with a while loop would result in the JIT in-

Figure 4.4: Implementation of the span-optimization (simplified)

```
spanloopdriver = jit.JitDriver(reds=["index", "inputstring"],                    1
                               greens=["instruction"])                           2

def spanloop(inputstring, index, instruction):  # span optimization              4
    while(index < len(inputstring)                                               5
          and instruction.incharlist(inputstring[index])):                       6
        index += 1                                                               7
        spanloopdriver.jit_merge_point()                                         8
    return index                                                                 9
```

terpreting this loop without generating code, which defeats the purpose of adding a JIT. This is the case because entering a nested while loop from the bytecode dispatch loop leaves the scope of the main JIT driver. This problem is solved by moving this while loop to a separate method, adding a separate JIT driver and specifying a `jit_merge_point` inside that nested while loop. Figure 4.4 shows the implementation. By introducing a new jit driver with the specific purpose of optimizing the span-loop, the instructions inside the span-loop are traced and corresponding machine code can be generated.

4.5 Optimizing Search Patterns

The motivation behind this optimization is structured as follows: Search patterns in LPeg typically have a very recognizable structure of `<pattern> + lpeg.P(1) * lpeg.V(1)`, which is also seen in the search pattern for URLs shown in figure 2.1. This structure attempts to match a pattern and, if unsuccessful, moves one place further in the string and tries again. Figure 4.5 shows the translation of a search pattern into bytecode.

Figure 4.5: Patternsearching bytecode

```
L1: testchar c L2
(code for pattern)
L2: any
jmp L1
```

This code attempts to match the first character of the desired pattern and, if unsuccessful,

jumps to L2, where a single character is consumed, at which point program flow jumps back to L1. The execution of this bytecode can be optimized by recognizing this bytecode structure intended to search for a character in a string and replacing it with python's built in method `str.find`. Since `str.find` is a built-in method that is included in the RPython-architecture, it is already efficient. Figure 4.6 shows both the function recognizing the described search pattern as well as the function used to replace the naive bytecode execution.

Figure 4.6: Source code for finding a search pattern on bytecode level

```
def testchar_check_optimize(instructionlist, pc):
    myself = instructionlist[pc]
    if instructionlist[myself.goto].name == "any":
        nextnextinstr = instructionlist[myself.goto+1]
        if (nextnextinstr.name == "jmp"
           and nextnextinstr.goto == myself.label):
            return True
    return False
[...]
def testchar_optimize(inputstring, index, char):
    assert index >= 0
    return inputstring.find(char, index)
```

4.5.1 Testset

The same search pattern structure described in 4.5 can also occur with a pattern with an undetermined first character. In this case, the initial `testchar` bytecode is replaced by a `testset` bytecode. Since Python in general and RPython in particular do not include a built-in method similar to `str.find` to find a member of a set in a string, a separate method is needed. The requirements for this method are to advance the `index` value of the VM as long as the current character is not part of a set. This happens to be the exact opposite behavior of the span-bytecode explained in section 2.4.1, meaning that the implementation of this optimized testset method shown in figure 4.7 uses the same principles as the optimized span loop described in section 4.4 to improve performance.

Figure 4.7: Source code for testset bytecode optimization (simplified)

```
testsetdriver = jit.JitDriver(reds=["index", "inputstring"],                    1
                              greens=["instruction"])                           2

def testset_optimize(inputstring, index, instruction):                          4
    assert index >= 0                                                           5
    while index < len(inputstring):                                             6
        if instruction.incharlist(inputstring[index]):                          7
            return index                                                        8
        index += 1                                                              9
        testsetdriver.jit_merge_point()                                         10
    return -1                                                                   11
```

4.6 Stack Representation

Due to the way the stack used for backtracking is accessed by the VM, a traditional linked-list stack implementation can be modified to improve performance by merging the returnaddress and choice point object, segmenting the stack and introducing memoization.

The stack is segmented by abusing the fact that the stack consists of alternating choice points with zero or more return addresses between them. By defining a stackentry as a choice-point containing a substack of return addresses, the VM has direct access to the topmost choice point, allowing it to skip over return addresses during backtracking as described in section 2.4.3.

Furthermore, by introducing memoization into the initialisation process of the nested return-naddress stack objects, the VM saves the time cost associated with re-initializing previously created objects by caching them. Memoization is applicable in this case, since the maximum number of distinct stacks of return addresses is determined by the created bytecode and thus the number of cached return address tuples cannot grow indefinitely.

Chapter 5

Results

In order to evaluate the advantages and shortcomings of adding and optimizing a tracing JIT to a pattern matching VM, benchmarks were run to compare different iterations of JitPeg to LPeg as well as Grep, a well-known and widely-used command-line utility included with many Unix-like operating systems. An otherwise idle Intel Core i5-2430M CPU with 3072 KiB of cache and 8 GiB of RAM, running with 2.40GHz was used. The machine was running Ubuntu 14.04 LTS and Lua 5.2.3. GNU grep 2.16 was used as a point of comparison for search patterns, whereas python 2.7.6 was used to compare JitPeg's performance to python's built-in module `json` using the `load`-method. The benchmarks were run 100 times in a new process each. Full runtime of the called process was measured, including starting the process. The plots shown in this section include various implementations of JitPeg to illustrate the effectiveness of the tracing JIT as well as the added optimizations. The boxes are defined to contain the medium 50% of data, whereas the whiskers contain the datapoints that fall less than 1.5 times the length of the box away from the box. Remaining points are outliers. The implementations include a version with the JIT turned off (`jitpeg_nojit`), a version with the JIT turned on and the optimization-flags turned off (`jitpeg_noopt`), and a version with the JIT and all optimizations enabled (`jitpeg_fullopt`).

5.1 Searching URLs

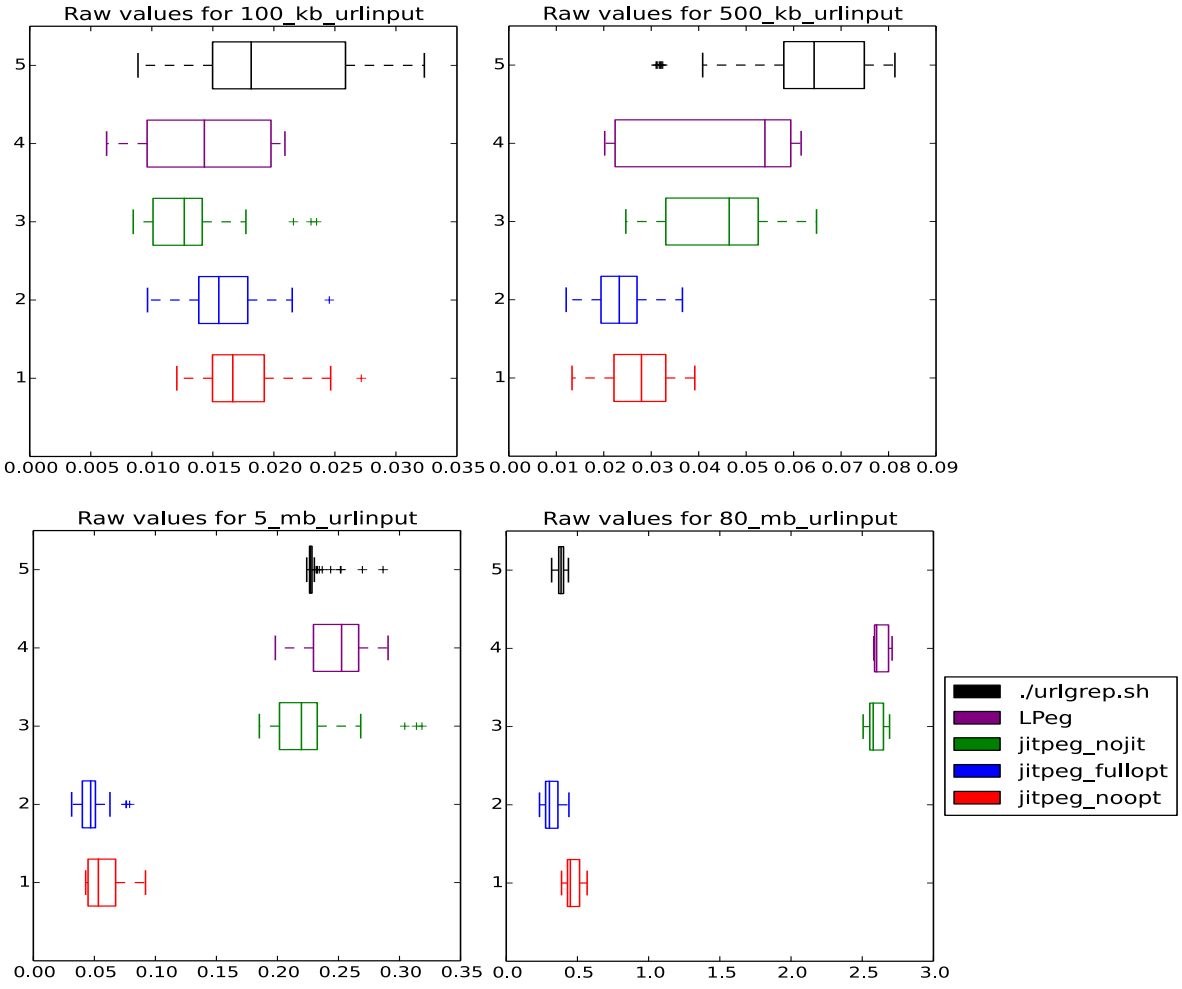
In order to test a practical search pattern, a pattern to find URLs was used. The actual pattern shown in figure 2.1 roughly approximates the actual definition of a valid URL but works well

enough to yield meaningful results in benchmarking. The regular expression used by grep is equivalent to the LPeg pattern and shown in figure 5.1. The plots shown in figure 5.2 show that at low inputsizes enabling a JIT may actually be a small disadvantage compared to the equivalent interpreter. This can be explained by the fact that tracing bytecode execution to detect loops is a task that requires computing resources and aims to save time eventually. The plots also show that at sufficiently high inputsizes the optimizations provide a tangible benefit on top of the tracing JIT.

Figure 5.1: Grep call with a regular expression describing URLs

```
#!/bin/sh
grep -oE 'http[-a-zA-Z0-9._~:/?#@!$&*+,;=]{3,}' $1
```

Figure 5.2: Plots for searching files of varying sizes for URLs



5.2 Parsing JSON

A pattern that recognizes JSON was used to test JitPeg's parsing capabilities. The implementation seen in figure 5.4 closely resembles the official specifications on the JSON website [noaa]. Since the grammar that describes JSON can not be expressed by regular expressions, the grep script was replaced with a python script using the `json`-module as a point of comparison. The plots shown in figure 5.3 confirm the observations previously seen in figure 5.2. A notable difference can be seen for the 80 megabyte JSON file, which LPeg was unable to parse. This is due to a limitation of LPeg's C-interpreter which requires it's stack to be of a fixed maximum size. After manually increasing this limit from 400 to 40000, LPeg was still unable to parse large JSON files. This raises the question if LPeg was intended to be used for parsing, since, if left at default settings, LPeg was unable to parse a JSON file of 100 kb in filesize. Compared to python's `json`-module, JitPeg performs surprisingly well at high filesizes given that the `json`-module is optimized for the sole purpose of parsing JSON.

Figure 5.3: Plots for parsing JSON files of varying size

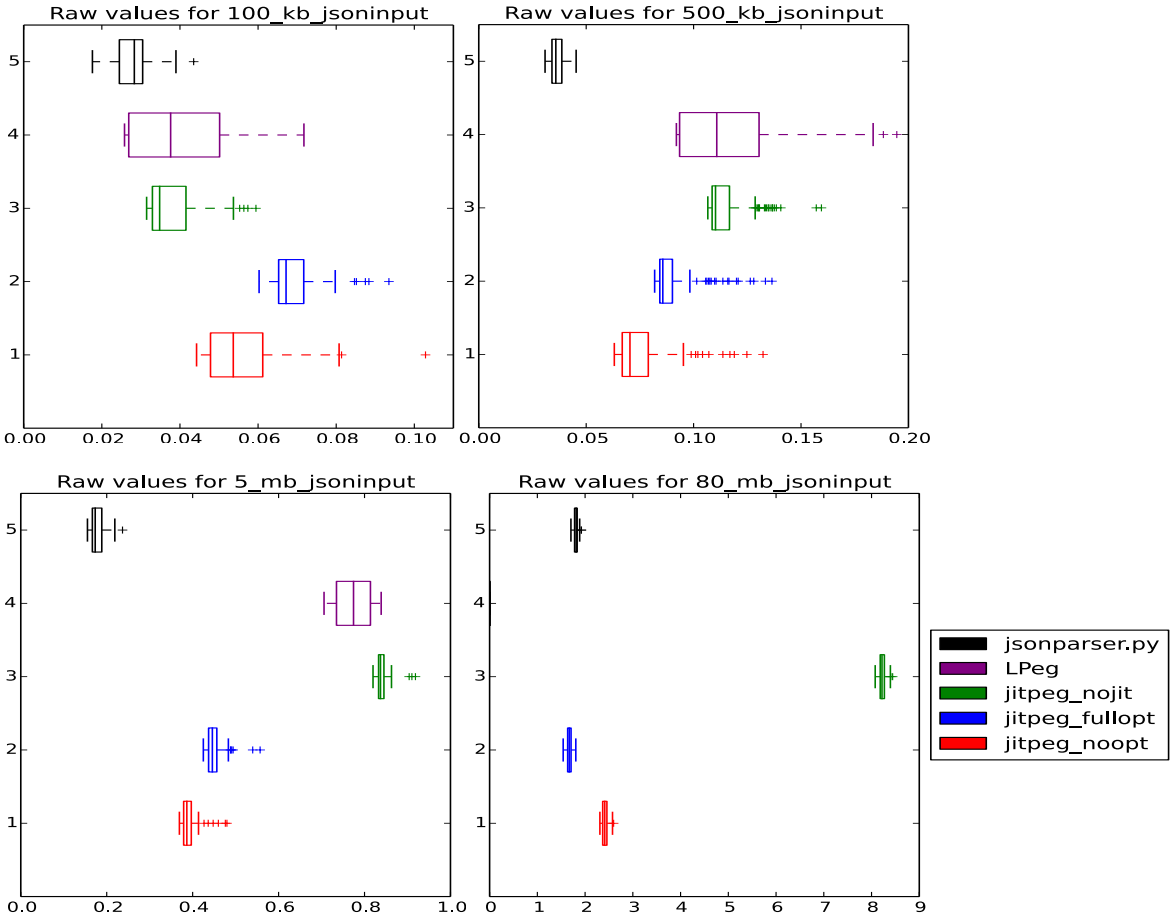


Figure 5.4: LPeg pattern for JSON

```
lpeg.P{
"JSON";
JSON = lpeg.V("VALUE"),

VALUE = lpeg.Cp()*lpeg.V("OBJECT")*lpeg.Cp()
       +lpeg.Cp()*lpeg.V("ARRAY")*lpeg.Cp()
       +lpeg.V("STRING")+lpeg.V("NUMBER")+lpeg.P("true")+lpeg.P("false")+lpeg.P("null"),

NUMBER = lpeg.V("EXP") + lpeg.V("FLOAT") + lpeg.V("INTEGER"),
INTEGER = lpeg.V("ONENINE")*lpeg.V("DIGITS")
         +lpeg.V("DIGIT")
         +lpeg.P"-"*lpeg.V("DIGIT")
         +lpeg.P"-"*lpeg.V("ONENINE")*lpeg.V("DIGITS"),
ONENINE = lpeg.R("19"),
DIGIT = lpeg.P"0" + lpeg.V("ONENINE"),
DIGITS = lpeg.V("DIGIT")^1,

FLOAT = lpeg.V("INTEGER")*lpeg.P(".")*lpeg.V("DIGITS"),
EXP = lpeg.V("FLOAT")*lpeg.P"E"*lpeg.V("SIGN")*lpeg.V("DIGITS")
     +lpeg.V("FLOAT")*lpeg.P"e"*lpeg.V("SIGN")*lpeg.V("DIGITS"),
SIGN = lpeg.P"+"+lpeg.P"-"+lpeg.P"",

ARRAY = lpeg.P"["*lpeg.V("ELEMENTS")*lpeg.P"]" + lpeg.P"["*lpeg.V("WS")*lpeg.P"]",
ELEMENTS = lpeg.V("ELEMENT") * lpeg.Cp()*lpeg.P"," * lpeg.V("ELEMENTS")
          +lpeg.V("ELEMENT"),
ELEMENT = lpeg.V("WS")*lpeg.V("VALUE")*lpeg.V("WS"),

STRING = lpeg.P'"'*lpeg.V("CHARACTERS")*lpeg.P'"',
CHARACTERS = lpeg.V("CHARACTER")^0,
CHARACTER = (lpeg.R('\032\255'))-lpeg.S([[\"]])
           +lpeg.V("ESCAPED"),
ESCAPED = lpeg.P([[\]]) * lpeg.S([["\/bnrt]])
         +lpeg.P([[\u]]) * lpeg.V("HEX")* lpeg.V("HEX")
         *lpeg.V("HEX")* lpeg.V("HEX"),
HEX = lpeg.V("DIGIT") + lpeg.R("af","AF"),

OBJECT = lpeg.P'{'*lpeg.V("MEMBERS")*lpeg.P'}'
         +lpeg.P'{'*lpeg.V("WS")*lpeg.P'}',
MEMBERS = lpeg.V("MEMBER")*lpeg.Cp()*lpeg.P',*lpeg.V("MEMBERS")
          +lpeg.V("MEMBER"),
MEMBER = lpeg.V("WS")*lpeg.V("STRING")*lpeg.V("WS")
         *lpeg.Cp()*lpeg.P":"*lpeg.V("ELEMENT"),

WS = lpeg.S('\32\13\10\9')^0
}
```

5.3 Conclusion

The goal of this thesis has been to improve the performance of LPeg by replacing the C-interpreter with an RPython-JIT. As was shown in the benchmarks, this goal has been reached. The underwhelming performance of the RPython-JIT at low total runtimes is explained by the time the JIT needs to warm up, roughly estimated to be at around 0.2 seconds [noae]. Improving the JIT by adding specific optimizations has also proven to be an effective performance upgrade at sufficiently high runtimes.

Chapter 6

Related Work

Compiling regular expressions to machine code has a long history, probably starting with Ken Thompson, who compiled regular expressions to IBM 7094 machine code [Tho68].

JIT-compiling regular expressions is a fairly standard technique that is used in Chrome [noah] and PyPy itself [BW08, chapter 19.5]

However, really efficient implementations do a lot more advanced techniques, such as employing SIMD instructions of modern CPUs [WHC+19].

There have been previous attempts at using RPython to generate a JIT compiler for a very simple description of regular languages, with good speedups [Bt10]. But it only matches regular expressions, not more advanced string patterns. JitPegs approach of using a linked list as stack of choice points is very similar to how choice points are represented as a stack of failure continuations in Pyrolog [BLS10]

Rosie [noag] is a pattern language similar to LPeg, which includes a library of patterns and focuses on maintainability, speed and extensibility.

Bibliography

[AACM07] ANCONA, Davide; ANCONA, Massimo; CUNI, Antonio; MATSAKIS, Nicholas D.: RPython: A Step Towards Reconciling Dynamically and Statically Typed OO Languages. In: *Proceedings of the 2007 Symposium on Dynamic Languages*. New York, NY, USA : ACM, 2007 (DLS '07). ISBN 978–1–59593–868–8, 53–64. . event-place: Montreal, Quebec, Canada

[BCF+11] BOLZ, Carl F.; CUNI, Antonio; FIJA\LKOWSKI, Maciej; LEUSCHEL, Michael; PEDRONI, Samuele; RIGO, Armin: Runtime Feedback in a Meta-tracing JIT for Efficient Dynamic Languages. In: *Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. New York, NY, USA : ACM, 2011 (ICOOOLPS '11). ISBN 978–1–4503–0894–6, 9:1–9:8. . event-place: Lancaster, United Kingdom

[BCFR09] BOLZ, Carl F.; CUNI, Antonio; FIJALKOWSKI, Maciej; RIGO, Armin: Tracing the Meta-level: PyPy's Tracing JIT Compiler. In: *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. New York, NY, USA : ACM, 2009 (ICOOOLPS '09). ISBN 978–1–60558–541–3, 18–25. . event-place: Genova, Italy

[BLS10] BOLZ, Carl F.; LEUSCHEL, Michael; SCHNEIDER, David: Towards a Jitting VM for Prolog Execution. In: *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*. New York, NY, USA : ACM, 2010 (PPDP '10). ISBN 978–1–4503–0132–9, 99–108. . event-place: Hagenberg, Austria

[Bol14] BOLZ, Carl F.: *Meta-Tracing Just-in-Time Compilation for RPython*. Düssel-

dorf, Heinrich-Heine-Universität Düsseldorf, Diss., 2014

[Bt10] BOLZ-TEREICK, Carl F.: *PyPy Status Blog: An Efficient and Elegant Regular Expression Matcher in Python*. https://morepypy.blogspot.com/2010/05/efficient-and-elegant-regular.html. Version: Mai 2010.

[BW08] BROWN, Amy; WILSON, Greg: *The Architecture Of Open Source Applications, Volume Ii*. Mountain View : lulu.com, 2008. ISBN 978–1–105–57181–7

[ESE06] EYERMAN, Stijn; SMITH, J.E.; EECKHOUT, Lieven: Characterizing the Branch Miss Prediction Penalty, 2006. ISBN 978–1–4244–0186–4, S. 48–58.

[For04] FORD, Bryan: Parsing Expression Grammars: A Recognition-based Syntactic Foundation. In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA : ACM, 2004 (POPL '04). ISBN 978–1–58113–729–3, 111–122. . event-place: Venice, Italy

[HP11] HENNESSY, John L.; PATTERSON, David A.: *Computer Architecture: A Quantitative Approach*. 5 edition. San Francisco, CA : Morgan Kaufmann, 2011. ISBN 978–81–7867–266–3

[Ier09] IERUSALIMSCHY, Roberto: A text pattern-matching tool based on Parsing Expression Grammars. In: *Software: Practice and Experience* 39 (2009), Nr. 3, 221–258. http://dx.doi.org/10.1002/spe.892. DOI 10.1002/spe.892. ISSN 1097–024X.

[noaa] *JSON*. https://json.org/.

[noab] *LPeg - Parsing Expression Grammars For Lua*. http://www.inf.puc-rio.br/~roberto/lpeg/.

[noac] *PCRE Performance Project*. https://zherczeg.github.io/sljit/pcre.html.

[noad] *PyJitPl5 — RPython Documentation*. https://rpython.readthedocs.io/en/latest/jit/pyjitpl5.html.

[noae] *PyPy - Performance*. https://www.pypy.org/performance.html#insider-s-point-of-view.

[noaf] *PyPy - Welcome to PyPy*. https://pypy.org/.

[noag] *Rosie Pattern Language: About Rosie Pattern Language (RPL)*. https://rosie-lang.org/about/.

[noah] *Speeding up V8 regular expressions · V8*. https://v8.dev/blog/speeding-up-regular-expressions.

[Rei11] REIS, Anthony J. D.: *Compiler Construction Using Java, JavaCC, and Yacc*. 1 edition. Hoboken, N.J : Wiley-IEEE Computer Society Pr, 2011. ISBN 978–0–470–94959–7

[RP06] RIGO, Armin; PEDRONI, Samuele: PyPy's Approach to Virtual Machine Construction. In: *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*. New York, NY, USA : ACM, 2006 (OOPSLA '06). ISBN 978–1–59593–491–8, 944–953. . event-place: Portland, Oregon, USA

[Tho68] THOMPSON, Ken: Programming Techniques: Regular Expression Search Algorithm. In: *Commun. ACM* 11 (1968), Juni, Nr. 6, 419–422. http://dx.doi.org/10.1145/363347.363387. DOI 10.1145/363347.363387. ISSN 0001–0782.

[WHC+19] WANG, Xiang; HONG, Yang; CHANG, Harry; PARK, KyoungSoo; LANGDALE, Geoff; HU, Jiayu; ZHU, Heqing: Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs, 2019, 631–648.

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 29.04.2019 Stefan Karl Troost

Diese CD enthält:

- eine *pdf*-Version der vorliegenden Bachelorarbeit
- die LaTeX- und Grafik-Quelldateien der vorliegenden Bachelorarbeit samt aller verwendeten Skripte
- die Quelldateien der im Rahmen der Bachelorarbeit erstellten Software
- den zur Auswertung verwendeten Datensatz
- die Websites der verwendeten Internetquellen