Harvard University

Senior Thesis

Incremental PEG Parsing

Author:
Zachary Yedidia

Advisor:
Professor Stephen Chong

A thesis submitted in partial fulfillment of the requirements for the degree of Bachelor of Arts

in the

Department of Computer Science

March 26, 2021

Abstract

Code analysis software in a text editor or IDE must repeatedly parse source code whenever an edit occurs. In many cases, a user's edit will affect only the parse of nearby characters, meaning a full-document reparse is unnecessary and inefficient. Incremental parsing algorithms support quick reparsing after common-case edits by remembering parse state and only parsing the parts of the document that have changed.

This thesis builds on previous work in incremental parsing for parsing expression grammars (PEGs). We develop new methods for incremental parsing that enable reparsing in logarithmic time in the common case for a wide variety of grammar types. These methods are implemented in a practical library called GPeg that supports efficient dynamic incremental parsers and a language-agnostic parser format via a parsing machine. Finally, we use this library to study the performance and usability of our parsing methods for the cases of pattern matching, full grammar parsing, and syntax highlighting.

Acknowledgements

I would like to give a huge thank you to my advisor Professor Stephen Chong for invaluable advice and encouragement, and being the best advisor I could have asked for. Working on this thesis has been a major highlight of my senior year, and I'm so thankful for all the support I received throughout this exceptional year. Warm thanks to Professor Nada Amin for generously offering to read my thesis. Finally, I'm deeply grateful to my mom, dad, and brother for their unending love and support since the very beginning.

Contents

Abstract **i**

Acknowledgements **ii**

1 Introduction **1**

2 Background **3**
2.1 Parsing Expression Grammars 3
2.2 Parsing Machines 5
2.3 Incremental PEG Parsing 6

3 A PEG Parsing Machine **9**
3.1 Basic Parsing Machine 9
3.2 Pattern Compiler 11
3.3 Optimizations 14
3.4 Additional Features 18
3.5 Implementation as a Bytecode Virtual Machine 20

4 Incremental Parsing **21**
4.1 Captures 21
4.2 Memoization 26
4.3 Memoization Table Implementation 29
4.4 Tree Memoization 32

5 Evaluation **38**
5.1 Language Parsing 38
5.2 Case Study: Syntax Highlighting 41
5.3 Pattern Matching 47
5.4 Encoding 49

6 Related Work **50**
6.1 Incremental Parsing 50
6.2 PEG Machines 50

7 Conclusion **52**
7.1 Future Work 52

References **53**

A Parsing Machine Specification **55**
A.1 Semantics 55
A.2 Encoding 60

B The GPeg Library **63**

Chapter 1

Introduction

Automated tooling for managing and analyzing source code is very important to developer productivity, and fundamentally relies on parsing. Text editors and integrated development environments (IDEs) tend to include source code analysis tooling so that programmers can see feedback immediately as they edit code. The problem of parsing therefore becomes more difficult because it is not efficient to parse the document from scratch after every edit. In many cases, it is not necessary either, as an edit only changes a small subset of the overall parse tree. Incremental parsing algorithms provide methods for reparsing only the necessary subsets of the document after an edit, while still providing the same resulting parse tree.

Source code analysis in a text editor can require either "heavy" or "light" parsing support. Heavy support indicates that the full abstract syntax tree for the language being parsed is required. For example, the first stage of compilation would require heavy parsing. In a text editor, this may be used for linting or notifications of syntactic or semantic errors. Light parsing support can be used when the analysis does not need as much language information.[1] For example, syntax highlighting generally only needs information about what kinds of tokens exist and how they are matched (keywords, strings, comments, integers, etc.). Code folding only needs to parse the document to find fold points, and does not need information about the order of operations in arithmetic expressions. Both applications of parsing are important to a text editor, but have slightly different requirements for the underlying parsing engine. Analysis routines that require heavy parsing are generally precompiled or even provided outside of the editor as a language server. Supporting many languages is difficult because each grammar requires a lot of effort to build. Light parsing tools on the other hand are generally provided directly by the editor for many languages, and there is an expectation that the editor can be extended with support for additional languages at runtime. As a result, static parser generation is not well-suited to text editor support. Additionally, the parse trees generated by light grammars tend to be very linear, with almost no tree-like structure, because these grammars operate more like searches.

This thesis presents an algorithm and implementation for incremental parsing that supports both heavy and light applications well, and is additionally easily usable for applications that do not require any incremental parsing. The algorithm builds on prior work in incremental PEG parsing, but introduces a new data structure allowing incremental reparsing to occur in logarithmic time in the common case. An implementation prototype is provided in the form of the GPeg library. It is our hope that GPeg can be easily used for incremental parsing in text editors, and we have plans to integrate it into the Micro text editor [34] to replace the current syntax highlighting engine.

Our contributions include:

[1] The terms "island grammars" and "micro-grammars" have also been used to describe this idea.

- A PEG parsing machine augmented to support memoization.
- A new interval tree data structure for the memoization table that improves the time complexity for memoization entry invalidation after an edit.
- A new memoization strategy that improves the time complexity for reparsing after applying edits.

The implementation can be found on GitHub at github.com/zyedidia/gpeg.
The thesis is organized as follows:

- Chapter 2 provides the background that this work directly builds on, including parsing expression grammars (PEGs), the LPeg pattern matching library and virtual machine, and previous incremental packrat parsing algorithms.
- Chapter 3 describes the basic PEG parsing machine implemented in GPeg, which is a reimplementation of the LPeg parsing machine with minor modifications. We describe the instruction set, a bytecode format, and an optimizing compiler for transforming grammars into bytecode.
- Chapter 4 describes the design and implementation of the new incremental parsing algorithm. First, new parsing machine instructions are presented for supporting traditional packrat parsing. Next, the implementation is described, which uses a new data structure for the memoization table. Finally, we present a new memoization strategy for efficiently handling flat grammar structures.
- Chapter 5 performs evaluation, testing the parsing engine in three use-cases: non-incremental parsing and pattern matching, incremental parsing for "heavy" workloads (a Java/JSON parser) and incremental parsing for "light" workloads (a syntax highlighter).
- Chapter 6 describes related work including incremental parsing algorithms for context-free grammars, and existing projects for incremental parsing.
- Appendix A provides the full specification of the parsing machine.
- Appendix B provides information for using the GPeg library.

Chapter 2

Background

2.1 Parsing Expression Grammars

2.1.1 Motivation

The formalism of parsing expression grammars (PEGs) was introduced by Bryan Ford in 2004 for building string recognizers [7]. The primary motivation is that while context free grammars (CFGs) were initially invented for generating languages, with an emphasis on natural language, they are now widely used for recognizing programming languages – a purpose for which they were not built nor designed. In particular, CFGs have two major problems for parsing programming languages:

1. Ambiguity: CFGs use non-deterministic choice for alternation, which can result in multiple correct parse results. For parsing natural languages this makes sense because natural languages inherently have ambiguity and resolving this ambiguity is not necessarily the job of the parser. On the other hand, programming languages are designed to be unambiguous, and if CFGs are used to define the language, special care must be taken to avoid ambiguity, or different parser implementations may return different parse results.
2. Lexing versus parsing: most programming language specifications are split into two parts: a CFG to define the hierarchy of the language, and a set of regular expressions to define the terminals of the CFG. Neither regular grammars nor context-free grammars are suitable on their own to fully define the language because CFGs cannot easily express common lexical idioms, and regular languages cannot express recursion.

These two problems result in complexity. There is complexity in the grammar definition to avoid ambiguity, and complexity in the parser implementations (GLR vs. LR vs. LL vs. LALR etc.) for how to deal with ambiguity. Programmers need to choose which parsing algorithm to use, and implement a separate lexer as well. The split between lexing and parsing can also result in actual problems with the resulting language, such as the C++ syntax for nested template arguments, which requires a space between angle brackets because the lexer would interpret the brackets as a right-shift token.

```
vector<vector<float> > matrix;
```

PEGs solve both of these problems with an unambiguous choice operator (solving problem 1), and predicates which allow unlimited lookahead and a unified language definition (solving problem 2).

```
doc           <- JSON !.
JSON          <- S_ (Number / Object / Array / String / True / False /
                   Null) S_
Object        <- '{' (String ':' JSON (',' String ':' JSON)* / S_) '}'
Array         <- '[' (JSON (',' JSON)* / S_) ']'
StringBody    <- Escape? ((!["\\\00-\37] .)+ Escape*)*
String        <- S_ '"' StringBody '"' S_
Escape        <- '\\' (["{|\\bfnrt] / UnicodeEscape)
UnicodeEscape <- 'u' [0-9A-Fa-f][0-9A-Fa-f][0-9A-Fa-f][0-9A-Fa-f]
Number        <- Minus? IntPart FractPart? ExpPart?
Minus         <- '-'
IntPart       <- '0' / [1-9][0-9]*
FractPart     <- '.' [0-9]+
ExpPart       <- [eE] [+\-]? [0-9]+
True          <- 'true'
False         <- 'false'
Null          <- 'null'
S_            <- [\11-\15\40]*
```

FIGURE 2.1: The JSON language described as a PEG.

2.1.2 Definition

This section presents an informal description of PEGs and the syntax used commonly to describe them, and particularly the parts that are different from context-free grammars. Figure 2.1 shows a complete PEG for the JSON language. Many constructs should be recognizable to those familiar with regular expressions and context-free grammars:

- `'...'` matches the literal in quotes (single or double quotes are acceptable).
- `[...]` matches and of the characters in brackets (including character ranges).
- `p*` greedily matches zero or more occurrences of `p`.
- `p+` greedily matches one or more occurrences of `p`.
- `a b` matches `a` followed by `b`.

A grammar is list of patterns called non-terminals, each associated with a name. The syntax `name <- pattern` is used to create a non-terminal, and the pattern may refer to other non-terminals (which can result in recursion).

The difference between PEGs and CFGs arises in the choice operator, `'/'`, and predicates such as the not predicate operator, `'!'`.

Choice Operator

The choice operator in PEGs is represented using the / character, primarily to differentiate it from the notation of | commonly used in CFGs. While in a CFG alternation does not specify

any order, this is not the case with PEGs. In a PEG the patterns in the alternation must be attempted in the order which they appear. Testing the next choice is allowed only if the previous choices failed to match. This results in unambiguous parses.

The "dangling else" problem is a classic example of ambiguity in CFGs. Solving the problem requires either using a meta-rule outside the CFG formalism, or restructuring the CFG which results in severe obfuscation. With PEGs, the correct behavior is easily expressed thanks to the unambiguous choice operator:

```
Statement <- IF Cond THEN Statement ELSE Statement
           / IF Cond THEN Statement
           / ...
```

One consequence of prioritized choice is that left recursion has no meaning in a PEG. Consider a left recursive rule `a <- a / ...`. Prioritized choice will cause the parser to always attempt to match `a` before trying alternatives. This causes infinite recursion and makes the pattern impossible to use in parsing.

Predicates

PEGs use *predicates* to express lookahead. The two predicate operators are `!` and `&`. The expression `!p` fails if `p` matches starting at the current location and succeeds otherwise. More specifically, `!p` attempts to match the pattern `p`, then backtracks to the original point where it began the match attempt while preserving only the knowledge of whether `p` matched or not. In both cases (success or failure) it does not consume any input. The expression `&p` is the converse of `!p` and succeeds if `p` matches and fails otherwise – this is equivalent to `!!p`.

One example of the not predicate is in the JSON grammar (figure 2.1) in the `doc` non-terminal. The pattern `!.` succeeds if it is not possible to accept another character, meaning that the end of the document has been reached. This ensures that the grammar will only accept an input if it is entirely matched by the `JSON` non-terminal.

In general, predicates can match patterns that have arbitrary length. In order to support predicates, a PEG parser must allow unlimited lookahead.

2.2 Parsing Machines

The concept of a parsing machine was first introduced by Knuth [17], and can be applied to parsing PEGs. Each pattern is compiled into a program for the machine. Programs can then be run by a virtual implementation of the machine to parse programs.

LPeg [15] is a library for Lua implemented in C which provides a parsing machine for matching PEGs. It is widely used in Lua as a replacement for regular expression matching, the purpose it was primarily built for. Indeed PEGs can be a good alternative to regular expressions because regular expression implementations often provide so many additional features that the formal theory of a regular language is no longer relevant to the implementation. PEGs provide a formalism that is more powerful than regular expressions, something that is clearly desired by users of regular expressions because so many features such as lookahead/lookbehind have been added to regex implementations.

LPeg uses a parsing machine [21] [14] to enable efficient dynamic parsing and avoid the packrat parsing algorithm (described next in section 2.3.1). LPeg chooses to avoid packrat

parsing so that it can efficiently parse large amounts of flat data, a workload that could result in unacceptably high memory usage in a packrat parser.

These are both key goals for a parser that may be used in a text editor (e.g., for syntax highlighting). In this thesis, we present a PEG parsing machine that is built on top of the same ideas as the LPeg machine (most core instructions and optimizations are the same as those described by [14]). Chapter 3 describes the parsing machine that we use, which is very similar to the LPeg machine, but in the subsequent chapters we augment the machine to support incremental parsing.

2.3 Incremental PEG Parsing

Incremental PEG parsing algorithms have been presented in the past. One of the most simple and effective methods is *incremental packrat parsing*, introduced by Dubroy et al. [4]. This thesis builds on incremental packrat parsing by presenting new data structures and parsing strategies, and implementing these algorithms in a PEG parsing machine.

2.3.1 Packrat Parsing

In 2002, Ford presented packrat parsing as a method for efficiently parsing PEGs [6]. Packrat parsing is commonly used for parsing PEGs because it guarantees linear time parses even though PEGs support unlimited lookahead. It works by keeping a memoization table – a data structure which allows the parser to remember the results of attempting to parse a certain pattern starting at a certain location. If the parser ever tries to reparse the same pattern at that location, it can first check the memoization table for an entry and if there is one simply use the information in the entry instead of parsing the input. Parsing time is linear because work is never duplicated and the grammar only has a fixed number of patterns to try before it fails.

The memoization table is a key-value store where the key is a pair (id, pos) which corresponds to a pattern[1] (uniquely identified by *id*) starting at a given location *pos* in the input. The value is a memoization entry which stores all the information produced by parsing from the pair (id, pos):

1. The length of the match (or a special value $\perp$ if the pattern did not match).

2. The parse tree generated by matching by the pattern (only if the pattern matched).

Patterns may be marked for memoization. The parser attempts to match such a pattern at a certain position, it will either succeed or fail. In both cases, the parser will then insert an entry into the memoization table at (id, pos), logging the result. Packrat parsing algorithms usually memoize every non-terminal in the grammar. However, it is possible and even desirable to use a different memoization strategy because the standard strategy has significant memory overhead

[1] Most packrat parsers only apply memoization to non-terminals.

2.3.2 Incremental Packrat Parsing

Packrat parsing is appealing for the case of incremental parsing because it keeps a memoization table of partial results. Incremental packrat parsing is a simple and effective algorithm that takes advantage of this.

As explained by Dubroy [4], a packrat parser can be modeled by a function:

$$\textsc{Parse} : (G, s) \to R$$

where G is a grammar, s is an input string, and R is the parse result (possibly a parse tree, or indication of success/failure).

An incremental packrat parser can be modeled similarly by a function:

$$\textsc{Parse} : (G, s, M) \to (M', R)$$

where M and M' are memoization tables. When M is empty, the incremental packrat parser is the same as the packrat parser, except it exposes the resulting memoization table to the user. If M is not empty, then the parser will execute faster because it will be able to skip entries that were filled before the parse even began.

After an initial parse, when an edit to the input string occurs, parts of the memoization table become invalid. Evicting these entries results in a valid memoization table which can then be used as an input for reparsing. Dubroy introduces a function for this:

$$\textsc{ApplyEdit} : (s, M, e) \to (s', M')$$

where e is an edit consisting of two parts: an interval $[e_{start}, e_{end})$, specifying the part of the document that is removed, and a string of bytes e_{text} which is then inserted at e_{start}.[2]

Combining the incremental parse function and edit application produces an incremental parsing algorithm, shown in algorithm 1.

Algorithm 1 Incremental Parse

1: $M \leftarrow$ a new memoization table
2: $s \leftarrow$ the initial input string
3: $G \leftarrow$ the grammar
4: **for** each edit operation e **do**
5: $\quad s, M \leftarrow \textsc{ApplyEdit}(s, M, e)$
6: $\quad M \leftarrow \textsc{Parse}(G, s, M)$

Applying the edit consists of evicting all newly invalidated memoization entries, and making sure the start positions of all entries are properly shifted according to the edit (deletion and/or insertion).

The incremental packrat parsing algorithm can thus be summarized as the following three steps that must be performed when an edit is made:

1. Determine all memoization entries that are invalidated by the edit and evict them from the memoization table (performed by ApplyEdit).

[2] Insertion and deletion are special cases where $e_{start} = e_{end}$, and $|e_{text}| = 0$ respectively.

2. Shift the start position of all memoization entries that start after the edit (performed by ApplyEdit).

3. Reparse the document from the start using the modified memoization table (performed by Parse).

A memoization entry is invalidated by an edit if any of the characters examined to succeed or fail the match are changed by the edit. Thus, in a memoization entry we must not only track how many characters were matched, but also how many characters were examined to make the much. Our memoization entry now stores:

1. The length of the match (or $\perp$ if the pattern did not match).

2. The number of characters examined to succeed/fail the match.

3. The parse tree generated by matching by the pattern (only if the pattern matched).

Since PEGs support unlimited lookahead, the number of examined characters may be much larger than the length of the match. A memoization entry at position p with n_e characters examined to make the match is invalidated by an edit over the interval $[e_{start}, e_{end})$ if that interval overlaps with $[p, p + n_e)$.

The incremental packrat parsing algorithm is effective for improving the performance of reparses, but previous implementations do not improve the asymptotic complexity of reparsing compared to the initial parse for any types of edits. This is because previous implementations use a traditional memoization table structure (an array or hashtable), which results in linear complexity for steps 1 and 2. Additionally, traditional memoization strategies (memoize every non-terminal) result in linear time for step 3 and high memory usage.

Using non-traditional memoization table data structures and a packrat parsing strategy tailored for reparsing, we can achieve logarithmic performance for reparsing in the common case.[3] The methods we propose for this are described in chapter 4.

[3]Since an edit can completely destroy the parse tree (e.g., opening a multiline comment at the top of the document), worst case complexity is still linear.

Chapter 3

A PEG Parsing Machine

This chapter describes the design and implementation of a basic PEG parsing machine. This parsing machine is heavily based on the LPeg parsing machine [14] [21]. Indeed many of the instructions are the same and most of the same optimizations are performed. In the next chapter more differences between the two approaches emerge as we will augment the PEG machine introduced here with additional support for captures, memoization, and incremental parsing.

3.1 Basic Parsing Machine

We will now introduce a stack-based virtual machine abstraction that can be used to apply a PEG to text. We will discuss how to compile a PEG into a program that can be run on the virtual machine, and how it can be used to determine if some text is accepted by the grammar. Note that the machine described in this chapter only determines if text is accepted by the grammar – it does not construct an AST or perform memoization. In the next chapter we will discuss how to augment the machine to support AST construction and memoization primarily for the purpose of incremental parsing.

The basic PEG machine is represented as a tuple of three values:

$$\langle ip, sp, S \rangle \in \mathbb{N}_{\perp} \times \mathbb{N} \times \textbf{Stack}$$

- The instruction pointer ip represents the address of the next instruction to execute. It also may have a special value $\perp$, which indicates that the machine is in a failure state and must run a recovery routine to return to a valid state, or if that isn't possible, fail the match.

- The subject pointer sp represents the index in the input string I of the next byte to be examined by the machine.

- The stack S is a list of entries, where S_1 is the top entry and $S_{|S|}$ is the bottom entry. In the basic parsing machine, there are two types of entries:

  1. Return entries, which store an instruction pointer to return to: $(ip_r)_{ret}$.
  2. Backtrack entries, which store both an instruction pointer and subject pointer to return to: $(ip_b, sp_b)_{bt}$.

The POP function takes as input a stack and returns the stack with the top entry removed, and separately also returns the top entry.

```
1: procedure POP(S)
2:     e ← S_1
3:     S ← S_{2...|S|}
4:     return S, e
```

The PEG machine supports a number of instructions that manipulate the machine state. Many instructions also use labels, which are simply pointers to instructions.

- `Char` b: advances ip and consumes one byte from the subject if it matches `B` and goes to the fail state otherwise.

```
1: if I[sp] = b then
2:     ip ← ip + 1
3:     sp ← sp + 1
4: else
5:     ip ← ⊥
```

- `Jump` l: sets ip to l.

```
1: ip ← l
```

- `Choice` l: pushes a backtrack entry storing l and sp so that the parser can return to this position in the document later and parse a different pattern (stored at l).

```
1: S ← (l, sp)_bt :: S
```

- `Call` l: pushes the next ip to the stack as a return address and jumps to l. Calls will be used to implement non-terminals.

```
1: S ← (ip + 1)_ret :: S
2: ip ← l
```

- `Commit` l: pops the top entry off the stack and jumps to l. This allows the machine to commit to a state and discard a backtrack entry.

```
1: S, _ ← POP(S)
2: ip ← l
```

- `Return`: pops a return address from the stack and jumps to it.

```
1: S, (ip_r)_ret ← POP(S)
2: ip ← ip_r
```

- `Fail`: sets ip to the fail state: $\perp$.

```
1: ip ← ⊥
```

- `End`: ends matching and accepts the subject.
- `EndFail`: ends matching and fails the subject.

In addition to the previous instructions, we have two instructions that aren't strictly necessary, but are useful.

- `Set` X: advances ip and consumes one byte from the subject if it is contained by the character set X, and goes to the fail state otherwise.

```
1: if I[sp] ∈ X then
2:     ip ← ip + 1
3:     sp ← sp + 1
4: else
5:     ip ← ⊥
```

- `Any` n: advances ip and consumes n bytes from the subject if possible and fails otherwise. This instruction can only fail by reaching the end of the subject.

```
1: if sp + n ≤ |I| then
2:     ip ← ip + 1
3:     sp ← sp + n
4: else
5:     ip ← ⊥
```

When the machine enters the failure state (when $ip = \perp$), stack entries are repeatedly popped. If a backtrack entry is popped, the machine backtracks to the state stored in the entry and can resume normal operation. If the stack is completely emptied, the match is declared a failure and is terminated.

1: **while** $|S| > 0$ **do**
2: $S, e \leftarrow \text{POP}(S)$
3: **if** $(ip_1, sp_1)_{bt} := e$ **then**
4: $ip \leftarrow ip_1$
5: $sp \leftarrow sp_1$

These instructions allow matching with unlimited lookahead via backtrack entries, loops via jumps, and functions via calls and returns.

3.2 Pattern Compiler

Given the basic instruction set of the PEG machine, we can implement each PEG operation as a small program. Let's first consider one of the simplest possible programs: one that only accepts the string `abc`.

```
Char 'a'
Char 'b'
Char 'c'
End
```

We match each character with the input and consume it if successful. If the match ever fails, the machine enters the fail state, and with no backtrack entry on the stack it will fail the match.

Note that a wildcard (the pattern '`.`') can be matched with `Any 1`, and arbitrary character sets can be matched with the `Set` instruction.

If we want to encode choice, for example to `abc / xyz` we can use the stack for backtracking:

```
    Choice L1
    Char 'a'
    Char 'b'
    Char 'c'
    Commit L2
L1: Char 'x'
    Char 'y'
    Char 'z'
L2: End
```

The `Choice` instruction at the start ensures that if we fail while trying to parse `abc`, we will backtrack and jump to `L1`, which will begin matching `xyz`. If we reach the commit, this means we have successfully parsed `abc`, and can delete the backtrack entry and jump to the rest of the program (which in this case is simply the `End` instruction to signal a successful match).

By using the commit instruction to jump over the second pattern in the alternation while simultaneously popping the backtrack entry off the stack, we encode the exact semantics for alternation in this case. This extends to the general case of `p1 / p2`. If `p1` and `p2` compile to <p1> and <p2>, then `p1 / p2` compiles to

```
    Choice L1
    <p1>
    Commit L2
L1: <p2>
L2: ...
```

Another core operation in PEGs is the repetition operator, written `p*`, which greedily matches `p`. We can encode this in our machine with a loop, which tries to match `p` in each iteration and exits the loop when it fails.

```
L1: Choice L2
    <p>
    Commit L1
L2: ...
```

The final interesting operator is the negative lookahead operator, written `!p`, which succeeds only if `p` does not match at the current position. It does not consume any input.

```
    Choice L2
    <p>
    Commit L1
L1: Fail
L2: ...
```

We first save the state of the machine with a `Choice` instruction. If matching `p` fails, we backtrack and continue with the rest of the program (a success, because `p` did not match). If we do in fact match `p`, we won't backtrack so we can remove the stack entry with a `Commit` and instead fail the overall pattern.

These core operations, along with a few others that are similar are shown in figure 3.1a. So far we have not been concerned with the efficiency of these operations. After discussing how to compile grammars we will examine ways to optimize the implementations of these operations.

3.2.1 Grammars

A grammar is a set of patterns each associated with a name. Each pattern/name is called a *non-terminal*. Non-terminals may refer to other non-terminals, in a possibly recursive manner.

To compile a grammar, we compile every non-terminal to a program that is marked by a specific label. Referring to a non-terminal compiles into a call instruction that jumps to the non-terminal's label. For example the expression `B <- '(' S ')'` would compile to

```
B:  Char '('
    Call S
    Char ')'
```

The program will be undefined if `S` does not exist, so another pass should be done to ensure all `Call` instructions have valid destinations.

For a complete example, consider the following grammar which matches parenthesized expressions:

Pattern	Compilation Result
’abc’	Char ’a’ Char ’b’ Char ’c’
.	Any 1
[...]	Set [...]
p1 p2	<p1> <p2>
p1 / p2	Choice L1 <p1> Commit L2 L1: <p2> L2: ...
p*	L1: Choice L2 <p> Commit L1 L2: ...
p+	<p> L1: Choice L2 <p> Commit L1 L2: ...
p?	Choice L1 <p> Commit L1 L1: ...
!p	Choice L2 <p> Commit L1 L1: Fail L2: ...

(A) Basic pattern compiler.

Pattern	Compilation Result
[...]*	Span [...]
!p	Choice L1 <p> FailTwice L1: ...
&p	Choice L1 <p> BackCommit L2 L1: Fail L2: ...
p*	Choice L2 L1: <p> PartialCommit L1 L2: ...
p+	<p> Choice L2 L1: <p> PartialCommit L1 L2: ...
[...]?	TestSetNoChoice [...]

(B) Optimizations.

FIGURE 3.1: Basic pattern compiler plus additional optimizations. All optimizations shown here are presented by LPeg.

```
S <- B / [^()]+
B <- '(' S ')'
```

Given that the starting non-terminal is S, this would compile to

```
    Call S
    End
S:  Choice L1
    Call B
    Commit L2
L1: Set {'\x00'..'\'','*'..'\u00ff'}
    Span {'\x00'..'\'','*'..'\u00ff'}
L2: Return
B:  Char '('
    Call S
    Char ')'
    Return
```

3.3 Optimizations

There are a variety of optimizations that can be made to the basic compiler, including traditional techniques like inlining and tail-call optimization, and more specialized approaches involving the introduction of new VM instructions.

3.3.1 Special-purpose instructions

The basic compiler performs a lot of stack manipulation. We can introduce three new instructions for the special cases of predicates and repetition.

- `PartialCommit` l: updates the backtrack entry on the top of the stack to the current subject position and jumps to l. This effectively performs a commit and choice in one instruction.

  1: $S, (ip_0, sp_0)_{bt} \leftarrow \textsc{Pop}(S)$
  2: $S \leftarrow (ip_0, sp)_{bt} :: S$
  3: $ip \leftarrow l$

- `BackCommit` l: pops the top backtrack entry off the stack and updates sp to its subject position, then jumps to l.

  1: $S, (ip_0, sp_0)_{bt} \leftarrow \textsc{Pop}(S)$
  2: $sp \leftarrow sp_0$
  3: $ip \leftarrow l$

- `FailTwice`: pops the top entry off the stack and sets ip to $\perp$.

  1: $S, _ \leftarrow \textsc{Pop}(S)$
  2: $ip \leftarrow \perp$

- `Span` X: consumes input and advances sp as long as the input matches the character set X. This instruction never fails, but might not consume any input if there is no match.

  1: **if** $I[sp] \in X$ **then**
  2: $\quad sp \leftarrow sp + 1$
  3: **else**
  4: $\quad ip \leftarrow ip + 1$

Using these instructions we can optimize the compilation of `p*`, `&p`, `!p`, and `[...]*` (where `[...]` is a character set), as shown in figure 3.1b.

3.3.2 Control flow optimization

A particularly important optimization for grammars is tail-call optimization. The compiler may use a `Jump` instruction instead of a `Call` if the `Call` is immediately followed by a `Return`. As a result, grammars which perform searches compile into loops.

For example, the grammar `X <- 'foo' / .  X` matches an input string if it contains the string "foo". This grammar compiles to:

```
    Call X
    End
X:  Choice L1
    Char 'f'
    Char 'o'
    Char 'o'
    Commit L2
L1: Any 1
    Jump X
L2: Return
```

In addition, if the compiler sees that there is no label marking the `Return` instruction, the `Return` can be removed entirely because it cannot be reached.

Another control flow optimization is the jump replacement optimization. None of the following instructions read the value of *ip*: `PartialCommit`, `BackCommit`, `Commit`, `Jump`, `Return`, `Fail`, `FailTwice`, and `End`. Therefore if any of these instructions are the target of a `Jump`, then the `Jump` can be replaced with the target instruction directly. In a sense, this optimization copies instructions that do not read *ip* to more convenient locations.

3.3.3 Head-fail optimization

A *head fail* is a failure that occurs on a pattern's first check. Head fails are very common: for example when searching for '`foo`', almost all fails will be head fails while matching '`f`'. Not only are head fails common, but they are also costly. Typically a head fail will involve a `Choice` instruction followed by a `Char` instruction that fails. The `Choice` instruction pushes the state onto the stack, and the failure causes the state to be immediately restored.

For optimizing head fails, we introduce some new instructions:

- `TestChar` b l: checks if b matches at the current sp. If so, pushes a backtrack entry on the stack, and advances sp. If not, jumps to l.

```
1: if I[sp] = b then
2:     S ← (l, sp) :: S
3:     sp ← sp + 1
4:     ip ← ip + 1
5: else
6:     ip ← l
```

- `TestSet` X l: checks if X matches at the current sp. If so, pushes a backtrack entry on the stack, and advances sp. If not, jumps to l.

```
1: if I[sp] ∈ X then
2:     S ← (l, sp) :: S
3:     sp ← sp + 1
4:     ip ← ip + 1
5: else
6:     ip ← l
```

- `TestAny` n l: checks if there are at least n characters remaining. If so, pushes a backtrack entry on the stack, and advances sp by n. If not, jumps to l.

```
1: if sp + n ≤ |I| then
2:     S ← (l, sp) :: S
3:     sp ← sp + 1
4:     ip ← ip + 1
5: else
6:     ip ← l
```

- `TestCharNoChoice` b l: checks if b matches at the current sp. If so, advances sp. If not, jumps to l.

```
1: if I[sp] = b then
2:     sp ← sp + 1
3:     ip ← ip + 1
4: else
5:     ip ← l
```

- `TestSetNoChoice` X l: checks if X matches at the current sp. If so, advances sp. If not, jumps to l.

```
1: if I[sp] ∈ X then
2:     sp ← sp + 1
3:     ip ← ip + 1
4: else
5:     ip ← l
```

Our head-fail instructions are slightly different from those described by Ierusalimschy for LPeg. LPeg does not perform a stack push in the success case of `TestChar` or `TestSet`, and instead uses a special-purpose `Choice` instruction for this. We find that it is cleaner and more efficient to include the stack push in the test instruction. Our `*NoChoice` instructions are equivalent to LPeg's test instructions.

Using these new instructions, we can replace common constructs such as

```
Choice L1
Char 'f'
```

with the single instruction `TestChar 'f' L1`. This will be much more efficient for a head-fail because there won't be any stack manipulation.

The `*NoChoice` versions of the `Test` instructions are more rarely useful, particularly for certain situations where the backtrack entry is not necessary. One such situation is optional character ranges: `[...]?`. The program should check if the set matches, and if it does not, it should directly jump to the next pattern. The `TestSetNoChoice` instruction has these exact semantics. Other optimizations also make use of these instructions, discussed in section 3.3.4.

Taking the example from before of `X <- 'foo' / . X`, which searches for "foo", we can see that the critical section that matches the "f" will be costly. However, with head-fail optimization, this becomes

```
    Call X
    End
X:  TestChar 'f' L1
    Char 'o'
    Char 'o'
    Commit L2
L1: Any 1
```

```
    Jump X
L2: Return
```

The critical section matching the initial "f" is now more efficient as it only manipulates the stack if it succeeds. When the "f" fails to match (which will be the case for the vast majority of attempts), no stack entry is pushed.

3.3.4 Common idioms

Charset alternation Operations with character sets can often be optimized. Alternation with sets performs set union. For instance, `[a-z] / [0-9]` can be optimized to join the two sets: `[a-z0-9]`. We can similarly merge single characters into character sets.

Charset not predicate The not predicate is also commonly used to represent set minus. For instance, `![a-z] .` represents `[^a-z]`, so we can also optimize this case by collapsing uses of the not predicate like this one into `Set` instructions.

Disjoint alternation When the two operands of an alternation begin with disjoint characters/sets, we know that if the first check of the first operand succeeds, the second operand will always fail (even if the first operand fails later in the pattern). This means we do not need the backtrack entry on the stack if the first check succeeds (we don't need to backtrack to check the second operand, because we know it will fail). For this case, we can use the `TestCharNoChoice` and `TestSetNoChoice` operators.

Dedicated search Since the search operation is somewhat common and often a bottleneck, GPeg provides a dedicated operator for it, and under certain conditions can provide optimization by restructuring the pattern used for searching. The usual idiom for searching for a pattern `p` is `S <- p / . S`. In this formulation we try to match `p` at every character in the document, which can be inefficient. Instead of attempting to match at every character, we can try to only match when the first character is known to be correct by consuming as many incorrect characters as possible before starting matching. Let `x` be the first character of pattern `p`. The following grammar performs a more efficient search for `p`:

```
S <- p / . [^x]* S
```

If `p` fails then it consumes one character and then consumes characters until the next `x` is found before trying to match `p` again. Thus we only attempt to match `p` when we find know the first test will succeed. Consumption of non-`x` characters is handled efficiently by the `Span` instruction. This technique is described by Ierusalimschy [14], and GPeg performs it automatically when the search operator is used (though `p` must begin with either a `Set` or `Char` instruction because `x` must be statically known).

3.3.5 Inlining

Inlining is a traditional and very effective compiler optimization which eliminates function calls by directly placing function bodies at the call-sites. Not only does inlining reduce stack pressure by eliminating call/return instructions, but it also enables other optimizations by making the function body available to the optimizer at the call site. Inlining typically results in larger code sizes, but faster runtime. The GPeg compiler only performs inlining for functions that do not use the `Call` instruction (though inlining `Call` instructions repeatedly can make other functions available for inlining). Thus mutually recursive or recursive functions

do not have inlining applied to any level. Additionally if a function has a size above a certain threshold, the function will not be inlined.

Figure 3.2 shows a visualization of the function calls between non-terminals. The colors display the compiler's inlining analysis based on the size of each non-terminal.

For an example of the effectiveness of inlining, let's examine how the JSON grammar defines numbers:

```
Number        <- Minus? IntPart FractPart? ExpPart?
Minus         <- '-'
IntPart       <- '0' / [1-9][0-9]*
FractPart     <- '.' [0-9]+
ExpPart       <- [eE] [+\-]? [0-9]+
```

With inlining combined with all the previously discussed optimizations, the `Number` non-terminal compiles to an efficient parsing program:

```
    TestCharNoChoice '-' L1
L1: TestCharNoChoice '0' L2
    Jump L3
L2: Set {'1'..'9'}
    Span {'0'..'9'}
L3: TestChar '.' L4
    Set {'0'..'9'}
    Span {'0'..'9'}
    Commit L4
L4: TestSet {'E','e'} L5
    TestSetNoChoice {'+','-'} L6
L6: Set {'0'..'9'}
    Span {'0'..'9'}
    Commit L5
L5: ...
```

Inlining is a key optimization that enables other optimizations and removes function call overhead.

3.4 Additional Features

The PEG parsing machine can also support some additional features that make it more practical for certain use-cases.

3.4.1 Error Recovery

Error messages and recovery are provided via a single instruction. Automatic error messages/recovery is not provided – the grammar creator has full control of messages, but also all the responsibility for the implementation.

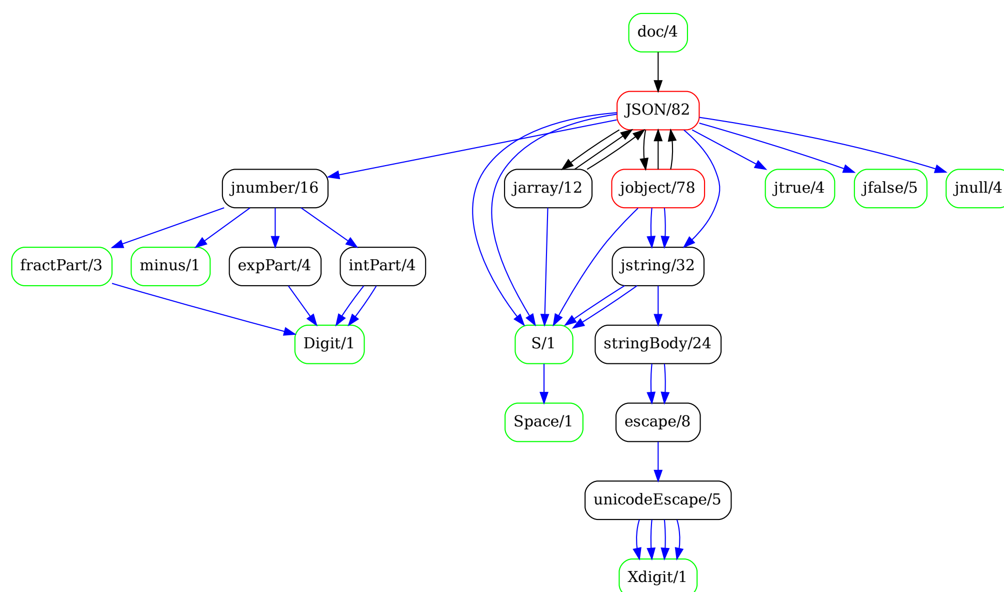


FIGURE 3.2: Function call dependencies in the JSON grammar. Blue arrow: inlined function call; black arrow: normal function call; green box: small function; red box: function that is too large to inline. The number in each box indicates the function's size (number of instructions).

- `Error` M l: records an error at the current position, with the message M. The parsing machine then jumps to l. The RECORDERROR function is implementation-specific but may print the error, or record it to a log.

```
1: RECORDERROR(M)
2: ip ← l
```

The idea is that once the parser completes, a list of errors can be returned. Since `Error` jumps to a specified label, it can jump to a program that performs a recovery routine and then jumps to a known state. This style of error recovery is presented and studied in [23] and [22].

For example, suppose we are writing a parser for a list of comma-separated identifiers. If the parser finds an invalid character while parsing an identifier, it can jump to a pattern that consumes characters until a comma is found, at which point the parser has returned to a known state and can jump back to the top-level instruction. In general, error recovery works by consuming characters until the parser knows it has returned to a valid state. Automating the creation of recovery patterns is a possible future direction.

If the grammar creator simply wants to terminate execution with a message when an error is found, `Error` can jump to an instruction `EndFail` that ends the program in failure.

Errors can be combined with prioritized choice and predicates to control the state that the error recovery routine runs in.

3.5 Implementation as a Bytecode Virtual Machine

Now that we have a specification for the PEG parsing machine, there are many ways it could be implemented. We have chosen to implement it as a bytecode virtual machine for two primary reasons: making a VM is much simpler than a JIT but is still efficient enough for most cases, and with a bytecode format compiled PEG programs can be compactly serialized and saved to disk or transferred to another program.

A text editor may provide support for parsing many languages via these grammars, and thanks to serialization the default set of grammars can be precompiled and serialized to save on startup time and storage space.

Having an explicit specification for the bytecode is useful because it means the parser is not tied to a single programming language. LPeg has no bytecode specification, so it is not simple to load LPeg programs to be run from a non-Lua environment. The hope with GPeg's specification is that it would be possible to implement libraries in different languages that can execute from the same bytecode format.

The encoding scheme is described fully in appendix A.2.

Chapter 4

Incremental Parsing

Currently the basic parsing machine can only determine whether or not the input string is accepted or not by the grammar. To make the parser more practical, we would also it to be possible to save parts of the input in a possibly hierarchical structure (an abstract syntax tree) for later processing. *Captures* serve to store this information for parts of the grammar that the user selects and are returned when parsing completes.

First, we add support for captures so that the parsing machine can return an abstract syntax tree rather than just an indication of whether the input string matched or not. We will use the same instructions as LPeg for specifying captures, but the implementation for actually performing the captures will be quite different because LPeg uses a delayed resolution of captures which is incompatible with memoization.

Second, to support incremental parsing, we add support for memoization. This will involve adding new instructions to the parsing machine (which will be similar to those for captures), and adding the new memoization table structure. At this point we will have a packrat parsing machine which can be used for incremental parsing. We describe a new structure for the memoization table, which uses an interval tree to make edit application more efficient than with a traditional packrat parsing memoization table (usually a flat array).

Since our overall goal for memoization is to support incremental parsing, we will also add some new instructions to modify the classic packrat parsing memoization strategy to increase its incremental parsing performance. We call this new strategy "tree memoization."

4.1 Captures

Captures allow the parser to construct an abstract syntax tree from a grammar. A pattern p may be marked for capturing with a specific ID which means that when the pattern is matched the text, and ID will be stored in a capture object. A pattern p marked for capturing may contain other patterns marked for capturing within it. In such a case, the inner captures will be stored as children of p's capture. In sum, a capture is a structure with three fields (note that the definition is recursive, because a capture's children are themselves captures):

$$(id, content, children) \in \mathbb{N} \times \textbf{Content} \times \langle \textbf{Capture} \rangle$$

The content of the capture may take different forms depending on the workload. A simple choice would be to store the captured text as a string (only for nodes that do not have children), but this loses information about where the captured occurred. Another choice would be to store the starting position and length, but this makes the capture more difficult

to relocate if an edit occurs.[1] We assume the content can be created from the start and end position of the capture using a function $content(sp_{start}, sp_{end})$.

If a pattern marked for capturing fails to match, its capture should not be recorded.

In our extended PEG syntax, a pattern may be marked for capturing by writing `{ p }`. This generates a unique capture ID for the pattern.

4.1.1 Stack Modification

We will introduce two new instructions, `CaptureBegin ID` and `CaptureEnd` which will be formally specified in the next section (4.1.2). For now, we can note that marking a program <p> to be captured with `ID` will involve wrapping it with those instructions:

```
CaptureBegin ID
<p>
CaptureEnd
```

Implementing captures will require a new type of stack entry, a capture entry $(id, pos)_{cap}$. The capture entry records the ID of the capture and its start position. The `CaptureBegin` instruction will be able to push a new entry of this type to the stack. When `CaptureEnd` runs, we create the capture. The question then is where do we store the capture? If a higher-level pattern in the grammar fails then the capture should be deleted, so we cannot store it to a global list of valid captures. In addition, we don't have any information about the children (captures created inside of <p>, since <p> may itself contain capture instructions).

The stack is the method for committing and tracking partial results. Thus to solve these issues, we add a list of captures to every stack entry:

- $\langle e, caps \rangle$, where e is a stack entry such as $(ip)_{ret}$, $(ip, sp)_{bt}$, $(id, pos)_{cap}$, and $caps$ is a list of capture structures. In our notation, we use e^{caps} to refer to the captures of a stack entry.

In addition, we keep a global list of "top-level" captures. Note that this updates total our machine state to

$$\langle ip, sp, S, C \rangle \in \mathbb{N}_\perp \times \mathbb{N} \times \textbf{Stack} \times \langle \textbf{Capture} \rangle$$

where C is the top-level list of captures.

When the parser creates a capture, it appends it to the list of captures of the entry at the top of the stack, or if the stack is empty it appends it to the top-level capture list.

If an entry is popped from the stack in the failure state, the entry's captures are discarded. Conversely if an entry is popped by a commit instruction (`Commit`, `PartialCommit`, or `BackCommit`), this is asserting that the current parser state is valid. Thus we would like to "propagate" the captures from the committed entry to the new top of the stack. Propagation specifically means appending the popped entry's captures to the captures of the new stack top. If the stack is empty after popping, the captures are appended to the top-level list. The two pop implementations are shown in figure 4.1.

[1] GPeg uses second method, but this has a significant memory overhead discussed in chapter 5. In the future GPeg may allow the user to select what kind of capture content information it needs.

```
1: procedure POP(S)
2:     e ← S_1
3:     S ← S_{2...|S|}
4:     return S, e
```

(A) Popping when $ip = \perp$ does not propagate any captures.

```
1: procedure POPANDPROP(S)
2:     e ← S_1
3:     if |S| > 1 then
4:         S_2^caps ← S_2^caps :: e^caps
5:     else
6:         C ← C :: e^caps
7:     S ← S_{2...|S|}
8:     return S, e
```

(B) Popping during a commit appends the popped entry's captures to the new top of the stack.

FIGURE 4.1: Stack popping behavior is modified to accodomate captures.

```
1: S, _ ← POPANDPROP(S)
2: ip ← l
```

(A) New semantics for `Commit` l.

```
1: S, (ip_0, sp_0)_bt ← POPANDPROP(S)
2: S ← ⟨(ip_0, sp)_bt, ⟨⟩⟩ :: S
3: ip ← l
```

(B) New semantics for `PartialCommit` l.

FIGURE 4.2: New semantics to propagate captures during commits.

4.1.2 Instructions

To implement captures, we introduce two new instructions:

- `CaptureBegin` *id*: starts a capture registered for *id* by pushing (id, sp) to the stack.

```
1: S ← (id, sp)_cap :: S
```

- `CaptureEnd`: pops a capture entry e off the stack storing $(id, sp_c)_{cap}$ (note that the capture list is not propagated for this pop). Creates a new capture object $(id, content(sp_c, sp), e^{caps})$, and this capture is appended to the capture list of the new stack top or the top-level capture list if the stack is now empty.

```
1: S, e ← POP(S)
2: (id, sp_c)_cap ← e
3: c ← (id, content(sp_c, sp), e^caps)
4: if |S| ≠ 0 then
5:     S_1^caps ← S_1^caps :: c
6: else
7:     C ← C :: c
```

We also must change commit instructions to use POPANDPROP (see appendix A for the complete instruction set semantics with all updates). The `PartialCommit` instruction must also be modified to clear the captures of the backtrack entry it is partially committing. The new semantics for `Commit` and `PartialCommit` are shown in figure 4.2.

As mentioned earlier, to capture a pattern p with *id* in our parsing machine, we can surround the output from compiling p with the `CaptureBegin` and `CaptureEnd` instructions:

```
Num: CaptureBegin 0
     Set  [1-9]
     Span [0-9]
     CaptureEnd
     Return
```

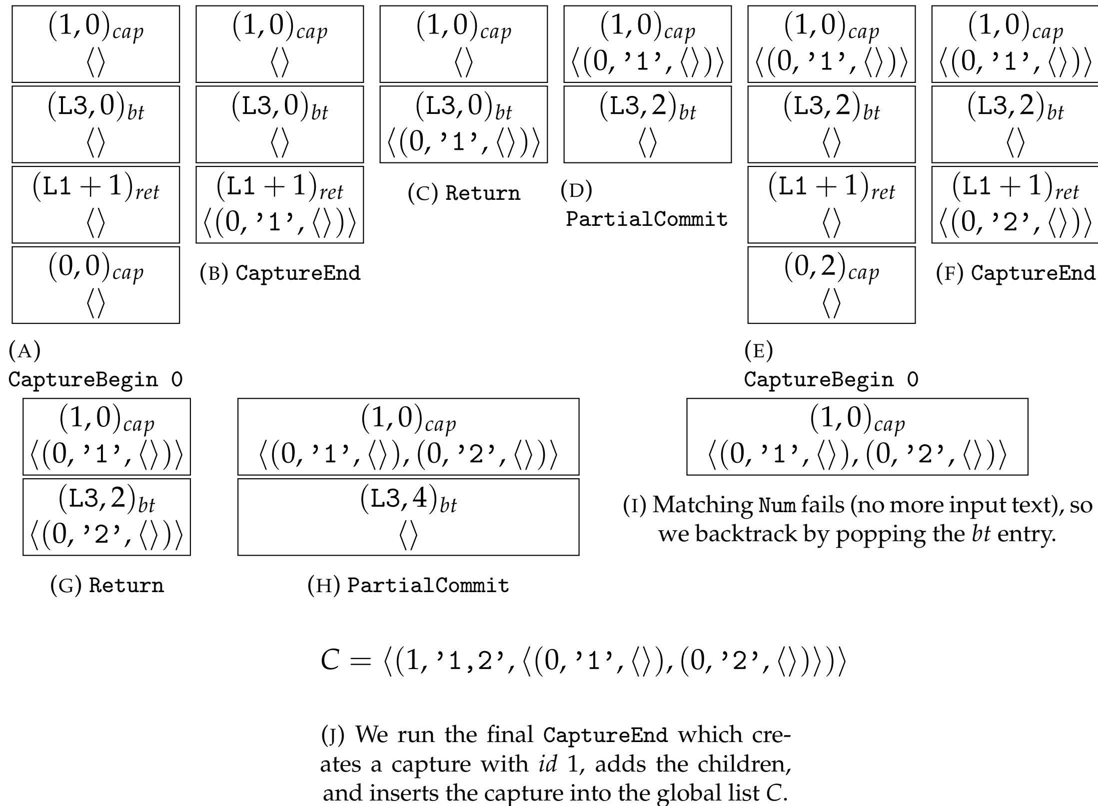


FIGURE 4.3: Stack state while matching `1,2` with the example number matching program. Each caption shows the most recently executed instruction.

This program matches and captures a number, and then returns to the caller. Composing with further captures, we can match 0 or more repetitions of a number, separated by an optional comma:

```
    CaptureBegin 1
    Choice L3
L1: Call Num
    TestCharNoChoice ',' L2
L2: PartialCommit L1
L3: CaptureEnd
```

This program will return a single capture referring to the entire input text, and with children referring to each number in the list. Each time the `PartialCommit` runs, the capture generated by calling `Num` is propagated upward to the capture entry for ID 1. Running the final `CaptureEnd` creates a capture whose children include all propagated captures (all occurrences of the numbers), and propagates it to the top level which is return to the user. Figure 4.3 shows a step-by-step sequence of stack operations using this grammar to parse `1,2`, which generates a capture tree where the root node matches the entire text and has two children, each matching the numbers `1` and `2` respectively.

For a more complex example, we can build a capture tree from an arithmetic grammar, where the order of operations is encoded into the grammar. In our extended PEG notation,

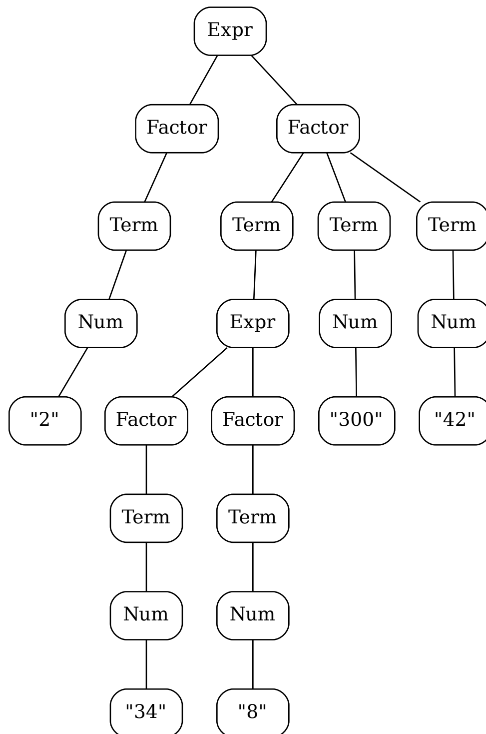


FIGURE 4.4: Capture tree generated by parsing `2+(34-8)/300*42` using the arithmetic grammar.

`{ p }` is used to capture patterns (each distinct use of `{ ... }` is assigned a unique ID when compiled).

```
Expr   <- { Factor ([+\-] Factor)* }
Factor <- { Term ([*/] Term)* }
Term   <- { Number / '(' Expr ')' }
Number <- { [0-9]+ }
```

Since this grammar uses a lot of nested captures, get a large tree structure for parsing arithmetic grammars. An example resulting capture tree is shown in figure 4.4.

4.1.3 Optimizations

Inserting a `CaptureBegin` instruction before a pattern can prevent certain optimizations from taking place such as head-fail optimization. In many cases, the `CaptureBegin` is placed between the `Choice` and `Char` instructions, preventing the program from being optimized to a `TestChar`. Additionally, in some cases the size of the pattern to capture can be known statically, meaning that pushing and popping a stack entry to track the starting location is unnecessary. To solve these two problems, we add two new capture instructions:

- `CaptureLate` n id: This instruction is the same as `CaptureBegin`, except that it marks the start location of the capture stack entry as $sp - n$.

```
1: S ← (id, sp − n)_cap :: S
```

- `CaptureFull` n id: Creates a new capture immediately where the content is the previous n bytes of the subject string. This instruction is equivalent to `CaptureLate` n id; `CaptureEnd`.

1: $c \leftarrow (id, content(sp - n, sp), e^{caps})$
2: **if** $|S| \neq 0$ **then**
3: $\quad S_1^{caps} \leftarrow S_1^{caps} :: c$
4: **else**
5: $\quad C \leftarrow C :: c$

With the new `CaptureLate` instruction, we can optimize our program to match and capture numbers:

```
Num: Set [1-9]
     CaptureLate 1 0
     Span [0-9]
     CaptureEnd
     Return
```

Now if this function is ever inlined to a location immediately after a `Choice`, a head-fail optimization can be applied.

In general, the `CaptureLate` instruction can be applied by "pushing" a `CaptureBegin` instruction down through instructions that consume a fixed number of bytes, such as `Char`, `Set`, `Any`. If a `CaptureLate` instruction is pushed so far that it is immediately before the `CaptureEnd`, the two instructions can be merged by the optimizer into a single `CaptureFull` instruction.

4.1.4 Comparison with LPeg

This method of creating captures is significantly different from LPeg. LPeg uses similar instructions, but does not perform stack propagation like GPeg. Instead LPeg marks certain locations when a capture occurs, and returns after the parse has been completed to build capture objects. This does not work well with incremental parsing because it does not allow captures to be accessed during the parse, which is necessary for storing/loading them as partial results to/from memoization entries.

4.2 Memoization

Implementing memoization in the parsing machine is similar to captures, except there is no need for propagation since backtracking cannot invalidate a memoization entry once it has been completed (the memoization table may store entries for patterns that are no longer a part of the final match). Since we implement memoization in a similar fashion to captures, the decision of which patterns to memoize is made by the program (and therefore either the grammar writer, or the compiler).

Like for captures, we introduce a new kind of stack entry: a memoization entry. It is exactly the same as a capture entry, a memoization ID and position: $(id, pos)_{memo}$.

We also have to add a memoization table to the machine state. The memoization table maps keys of the form (id, pos) to entries. Recall from section 2.3.2 describing incremental packrat parsing that a memoization entry must store the following information:

- The length of the match (or $\perp$ if the pattern did not match).
- The number of characters examined to succeed/fail the match.
- The parse tree generated by matching by the pattern (only if the pattern matched).

In the parsing machine, a memoization entry is

$$(id, sp, len, exam, caps) \in \mathbb{N} \times \mathbb{N} \times \mathbb{N}_\perp \times \mathbb{N} \times \langle \textbf{Capture} \rangle.$$

Tracking the number of examined characters requires adding an additional register in the parsing machine to keep track of the number of characters examined so far. For this purpose, we add a new register, *ep*, which tracks the furthest location ever read (or examined) in the subject string. The *ep* register is almost the same as *sp*, except it can only be increased (for example, it is not updated by backtracking). This is a conservative estimate of the number of examined bytes for a given pattern, and may be larger than the actual number.

Our machine state is now

$$\langle ip, sp, S, C, M, ep \rangle \in \mathbb{N}_\perp \times \mathbb{N} \times \textbf{Stack} \times \langle \textbf{Capture} \rangle \times \textbf{MemoTable} \times \mathbb{N}$$

where M is the memoization table, and *ep* is the examined pointer (the furthest examined byte in the input string).

To avoid large space usage in the memoization table, we do not perform memoization if the entry's number of examined bytes is below a certain threshold. Generally, the threshold is set to 512, but this can be tuned.

4.2.1 Instructions

Similar to captures, the two basic instructions we introduce are:

- `MemoOpen` l id: checks if there is a memoization entry $(id, sp_1, len, exam, caps)$ in the table corresponding to (id, sp). If so, jumps to l, and advances sp by len (except if len is -1, which causes the machine to go to the failure state). If there is no corresponding memoization entry, a memoization stack entry $(sp, ID)_{memo}$ is pushed.

1: **if** $e \leftarrow M[(id, sp)]$ **then**
2: $\quad (id, sp_1, len, exam, caps) \leftarrow e$
3: $\quad$ **if** $len \neq \bot$ **then**
4: $\quad\quad sp \leftarrow sp + len$
5: $\quad\quad ep \leftarrow max(sp, ep)$
6: $\quad\quad S_1^{caps} \leftarrow S_1^{caps} :: caps$
7: $\quad\quad ip \leftarrow l$
8: $\quad$ **else**
9: $\quad\quad ip \leftarrow \bot$
10: **else**
11: $\quad S \leftarrow (id, sp)_{memo} :: S$

- `MemoClose`: pops a memoization stack entry e of the form $(id, sp_1)_{memo}$. Inserts a new memoization entry into the table created using the following information:
  - The start position sp_1.
  - The length $sp - sp_1$.
  - The number of examined characters $ep - sp_1$.
  - The captures created while parsing the pattern that was memoized, stored in e^{caps}.

1: $S, e \leftarrow \text{PopAndProp}(S)$
2: $(id, sp_1) \leftarrow e$
3: $m \leftarrow (id, sp_1, sp - sp_1, ep - sp_1, e^{caps})$
4: $M \leftarrow M[(id, sp_1) \mapsto m]$

If a memoization entry is popped during failure, we can run the same routine as for `MemoClose`, but use a length of -1 to indicate that the pattern did not match.

As an example, let's use a simple arithmetic grammar with memoization, where the syntax `{{ p }}` marks `p` for memoization.

```
Expr   <- Factor ([+\-] Factor)*
Factor <- Term ([*/] Term)*
Term   <- Number / '(' Expr ')'
Number <- {{ [0-9]+ }}
```

Compiling this grammar results in the following program (note that `Number` was inlined into `Term`):

```
        Call Expr
        Jump L8
Expr:   Call Factor
        Choice L2
L1:     Set {'+','-'}
        Call Factor
        PartialCommit L1
L2:     Return
Factor: Call Term
        Choice L4
```

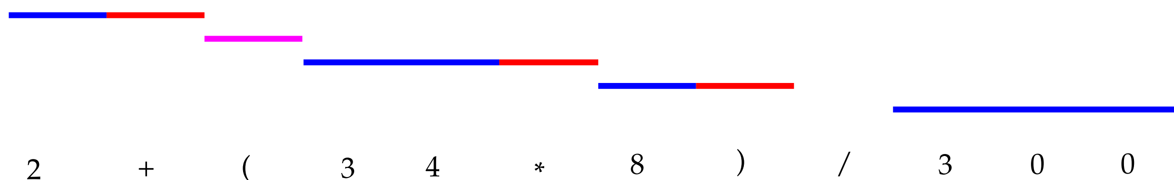


FIGURE 4.5: Memoization entries generated by the example arithmetic grammar when parsing `2+(34*8)/300` (this grammar only memoizes numbers). Each level indicates a new memoization entry for `Num`. The length of the entry is shown in blue. The number of examined characters past the end of the length are shown in red. Examined characters for matches that failed are shown in magenta.

```
L3:     Set {'*','/'}
        Call Term
        PartialCommit L3
L4:     Return
Term:   Choice L6
        MemoOpen L5 0
        Set {'0'..'9'}
        Span {'0'..'9'}
        MemoClose
L5:     Commit L7
L6:     Char '('
        Call Expr
        Char ')'
L7:     Return
L8:     End
```

The resulting memoization table has a structure as shown in figure 4.5.

4.3 Memoization Table Implementation

The memoization table implementation is critical to the runtime performance of incremental parsing. In particular, recall from section 2.3.2 on incremental packrat parsing that the first two steps of the incremental packrat parsing algorithm manipulate the memoization table:

1. Determine all memoization entries that are invalidated by the edit and evict them from the memoization table. Entries are invalidated if their examined interval overlaps with the edit interval.

2. Shift the start position of all memoization entries that start after that edit. The entries are shifted by the difference between the size of the inserted text and deleted text (the overall change in text size).

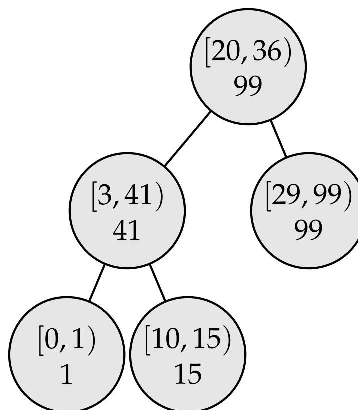


FIGURE 4.6: Interval tree storing five intervals. Each node stores an interval and the maximum endpoint in its subtree (including itself).

Existing implementations of incremental packrat parsing use a traditional memoization table data structure: either an array or a hashmap. While both structures provide constant-time access while parsing, they require linear time in the size of the file for both steps, assuming the number of memoization entries is proportional to the size of the file. This quickly becomes inefficient for incremental parsing.

Luckily, the interval tree is an existing data structure that directly solves step 1. With some augmentations, it can also make step 2 much more efficient.

4.3.1 Interval Tree

An interval tree is an augmented binary search tree that stores a set of n intervals. It can perform the following operations:

- Insert a new interval: $O(\log n)$.
- Delete an interval: $O(\log n)$.
- Query for all intervals that overlap with a specified interval: $O(\log n + m)$, where m is the number of overlapping intervals (size of the result).

The interval tree provides an ideal solution to step one because it allows us to efficiently determine all memoization entries that overlap with the edit interval, and therefore need to be evicted. Since the interval tree is an augmented binary search tree, it can be interpreted as a key-value store. In our case, the key is the pair (id, pos), and the value is the memoization entry associated with that key – for the purposes of the interval tree in particular, the value is the memoization entry's interval $[e_{pos}, e_{pos} + e_{examined})$. When sorting keys, the position is the primary key and the ID is used to arbitrary break ties (this can occur when two patterns can be parsed from the same starting location).

Implementing an interval tree involves augmenting a binary search tree. Each node in the tree corresponds to an interval, and that node is sorted based on the start position of the interval. In addition, each node in the tree stores the maximum position of any interval in its children. This allows the overlap search to skip entire subtrees, resulting in logarithmic time queries. An example interval tree is shown in figure 4.6.

The QUERY procedure for interval trees, shown in algorithm 2, recursively finds all intervals in the tree that overlap with the query interval. Each node makes sure to avoid subtrees if it can guarantee the interval will not overlap with any intervals in those subtrees. These guarantees can be made because the maximum (and minimum, via the start position) endpoints of every subtree are known.

Algorithm 2 Interval-Overlaps

1: **procedure** QUERY(n, $[l, h)$, r) ▷ Adds the intervals in the tree with root n that overlap with the interval $[l, h)$ to the list r in sorted order.
2: **if** $l \geq n_{max}$ **then**
3: **return** r
4: $r \leftarrow$ QUERY(n_{left}, $[l, h)$, r)
5: **if** OVERLAPS($[l, h)$, $n_{interval}$) **then**
6: $r \leftarrow r :: n_{interval}$
7: **if** $h \leq n_{start}$ **then**
8: **return** r
9: $r \leftarrow$ QUERY(n_{right}, $[l, h)$, r)
10: **return** r
11: **procedure** OVERLAPS($[l_1, h_1)$, $[l_2, h_2)$) ▷ Returns true iff $[l_1, h_1)$ overlaps with $[l_2, h_2)$
12: **return** $l_1 < h_2 \wedge h_1 > l_2$

The GPeg implementation builds the interval tree using an AVL tree as the underlying binary search tree. This ensures that the tree remains balanced.

4.3.2 Lazy Shifts

Step 2 still poses a problem for the interval tree. When new text is inserted into the document, all the intervals that refer to ranges after that text must be shifted by the amount of text inserted. The vanilla interval tree does not support and mechanism for efficiently shifting intervals in the tree. The normal method would be to iterate through all intervals that come after the edit and shift them, but this is at worst linear time and must be performed for every edit.

Our solution is to perform shifts lazily. The tree stores a log of shifts that have yet to be applied, and a timestamp for each shift. Creating a new shift adds it to the log and increases the shift time (the value used for creating timestamps). When a query is made in the tree, all shifts are applied to each visited node as necessary. Since the number of shifts is proportional to the number of edits so far, the worst case runtime of a query becomes proportional to the number of edits rather than the size of the file (an improvement for most workloads). Additionally, since shifts are applied once to nodes, repeated queries in the same region of the interval tree will not require much (or possibly any) shift application, making repeated edits to the same portion of the document fast.

Improving on the performance further must resolve the problem of "garbage collection" of old shifts: when do we know a shift has been applied to the entire tree and can therefore be removed from the log? The easiest solution is to periodically (e.g., every n edits) apply all

shifts to all nodes in the tree. This is the behavior of GPeg's current interval tree implementation. This leads to a considerable spike in latency for the singular edit when shift collection occurs.

Another strategy is to use a full lazy approach, similar to existing lazy graph processing algorithms [1] [5]. In this approach we keep a log of shifts for every node. When a shift is applied to a node, it then adds the shift to those of its children for which the shift is relevant (if a child refers to an interval starting before the shift location it cannot be affected by the shift). This formulation has the nice property that shifts are lazily pushed through the tree and as a result shifts that have been applied to top-level nodes are automatically collected from the shift history of those nodes (no need for separate garbage collection). There may also be opportunities for coalescing shifts together as they get pushed down the tree. Though this method has not been implemented, it seems like a promising approach for improving reparse time for documents that undergo many edits.

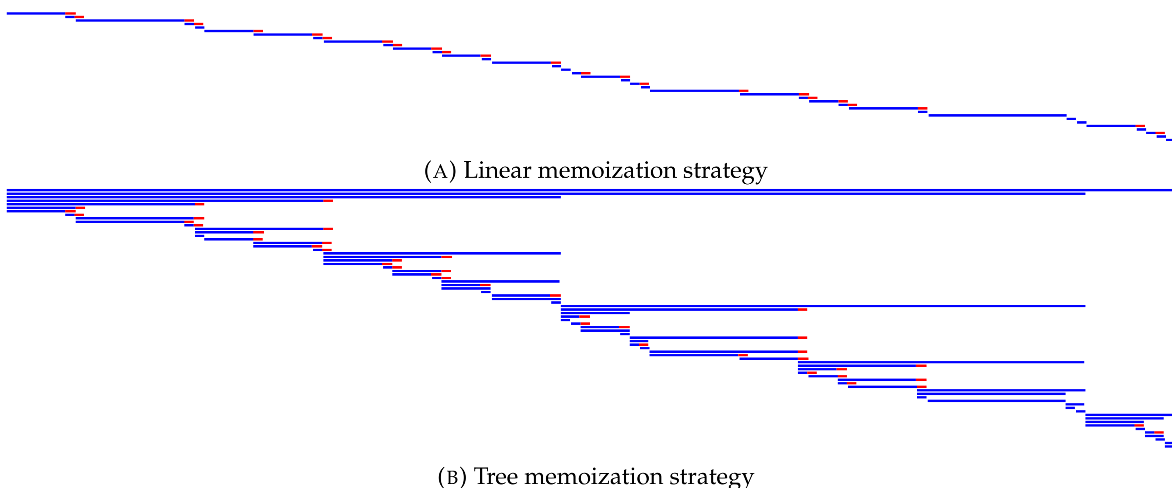
4.3.3 Interval Rope Design

Lazy shifting is easy to apply to an interval tree because the interval tree is sorted by absolute file positions. A more radical approach would be to build a tree where each interval is easily relocatable. Unfortunately, implementing the interval query operation becomes more difficult as a result. A good data structure to take inspiration from is the rope, which is traditionally used for supporting string insertion and removal in logarithmic time. In a rope, each node of the tree stores the size of its subtree, and leaf nodes store a chunk of text. Inserting new text involves changing the leaf node and updating the sizes of all parent nodes (this performs a number of operations that is proportional to the depth of the tree). It may be possible to use a similar structure for intervals, where the tree stores chunks of intervals (possible as interval trees), each with a starting position relative to the start of the chunk. The start of each chunk can be determined by a logarithmic-time walk through the tree. If the intervals are evenly distributed throughout the file, an overlap query should be roughly $O(\log n + \log i)$, where n is the size of the file and i is the number of intervals. This "interval rope" also seems like a promising data structure for this problem, but has not been implemented or formalized.

4.4 Tree Memoization

With a new memoization table data structure, we can invalidate and evict entries from the table much faster. The last step in the incremental packrat parsing algorithm is to reparse from the start of the document, using the memoization table to skipped unchanged parts. Source code in typical programming languages tends to exhibit a logarithmic structure in the parse tree: a class contains a set of function definitions, and each function consists of statements, and each statement consists of expressions, etc. The program structure is organized as a hierarchy where the parse tree has a depth that is logarithmic with respect to the number of parse nodes. This is great because it means that memoizing every non-terminal (the typical memoization strategy) creates a tree structure in the memoization table (see figure 4.7b). As a result, reparsing from the start will have runtime logarithmic in the size of the file.

However, many types of grammars (e.g., "light" grammars) tend to have a very linear structure, since they mostly consist of repetition of a single token non-terminal. Grammars for syntax highlighting are an example of this. More generally, the repetition operator causes



(A) Linear memoization strategy

(B) Tree memoization strategy

FIGURE 4.7: Resulting memoization table for two different strategies.

problems for the reparse stage because it leads to memoization structures that have a linear rather than logarithmic structure, as shown in figure 4.7a.

Our solution is to change the memoization strategy for the repetition operator. Repeating a memoized pattern, written `{{ p }}*`, would ordinarily be compiled to:

```
    Choice L2
L1: MemoOpen L3 ID
    <p>
    MemoClose
L3: PartialCommit L1
L2:
```

This leads to the linear behavior we want to avoid, since each memoization entry is inserted one after the other (figure 4.7a). We can solve this by forcing a tree structure in this specific case. Every time we find two memoization entries that are side by side and similarly sized (in terms of number of occurrences of `p`), we can add a new memoization entry that encompasses both of them. This strategy leads to a tree structure in the table allowing reparsing to skip much larger amounts of the text at once.

In order to implement this behavior in the virtual machine, there are two changes to make.

- Since this automatically imposed tree structure requires multiple entries that can start at the same location, the memoization table must be modified to accommodate this.
- New instructions for this behavior need to be introduced, and the compilation of memoized repetition must be special-cased to use these instructions.

4.4.1 Memoization Table Modifications

This new memoization strategy involves modifying the memoization table so that it can store multiple intervals each starting at the same position (and with the same ID). Each node in the tree now stores a set of intervals which all begin at the same location. When considering

maximum endpoints, the maximum of all intervals in a node is considered. When retrieving an entry from the memoization table using the key (id, pos), the entry with the largest interval $[start, start + examined)$ is used. This allows the parser to skip the maximum possible amount while reparsing. The nodes may be kept in a list, or a more complex structure like a priority queue. Since the number of intervals in a node should be logarithmic with respect to the size of the file, using a priority queue over a resizable array is not necessary.

Before this modification, entries can be evicted by specifying the key (id, pos). Now the key must either be more specific, or all the intervals associated with a key will be evicted (even though they might not all overlap with the edit). A better strategy could be to combine eviction and overlap querying so that only the specific overlapping intervals are removed. GPeg's current implementation does not perform this optimization, but this is low-hanging fruit for improving performance.

4.4.2 Machine Modifications

Our approach for implementing tree memoization is to build the tree on the fly as it is being parsed. Instead of using `MemoOpen` and `MemoClose` for memoization, we introduce new instructions (of the form `MemoTree*`) for this purpose. These instructions will keep information about previous memoization entries on the stack.

When two entries containing n of occurrences of `<p>` have been created in a row, we want to create a third entry that encompasses both of them, containing $2n$ occurrences of `<p>`. Therefore we need to keep track of the number of occurrences of `<p>` in each memoization entry.

A memoization entry now stores:

$$(id, sp, len, exam, caps, count) \in \mathbb{N} \times \mathbb{N} \times \mathbb{N}_{\perp} \times \mathbb{N} \times \langle \textbf{Capture} \rangle \times \mathbb{N}.$$

where the count is only used for memoization entries generated by `MemoTree*` instructions.

Similarly, we modify memoization stack entries to contain a count: $(id, pos, count)_{memo}$. In a stack entry, the count tracks the number of occurrences of `<p>` (starting at pos) that were been memoized by the time the entry was pushed.

The new instructions are formally specified below. Note that many of the instructions need to modify the top stack entry without popping it. In the pseudocode, we pop the entry and then pushed it back with modifications. In the actual implementation, it would be more efficient to perform a peek and modify the entry in place.

- `MemoTreeOpen` l id: This instruction starts begins a new memoization entry that is meant for tree memoization. It is very similar to `MemoOpen`, except it pushes a memoization stack entry both when it finds a memoization entry and when it doesn't. We want to push a stack entry even if the memoization entry already exists because we want to be able to construct higher levels of the tree. If we do not push the entry to the stack, subsequent repetitions will not know the count of the existing entry, and will not be able to perform merges to create the higher tree levels.

```
1:  if e ← M[(id, sp)] then
2:      (id, sp_1, len, exam, caps, count) ← e
3:      if len ≠ ⊥ then
4:          sp ← sp + len
5:          ep ← max(sp, ep)
6:          S ← (id, sp, count)_memo :: S
7:          S_1^caps ← S_1^caps :: caps
8:          ip ← l
9:      else
10:         ip ← ⊥
11: else
12:     S ← (id, sp)_memo :: S
```

- `MemoTreeInsert`: This instruction peeks the top entry on the stack and memoizes it. It also increases the count of the top entry on the stack. This should only be performed for entries corresponding to non-memoized sections of the text. Notice how in the full formulation given below, the `MemoTreeOpen` instruction skips `MemoTreeInsert` if a memoization entry already exists for the pattern to be matched.

```
1: S, e ← PopAndProp(S)
2: (id, pos, count)_memo ← e
3: S ← (id, pos, count + 1)_memo :: S
4: m ← (id, pos, sp − pos, ep − pos, e^caps, count + 1)
5: M ← M[(id, pos) ↦ m]
```

- `MemoTree`: This instruction creates the upper-level tree structure by scanning up the stack and coalescing entries that have the same count. It repeatedly peeks the top two entries on the stack, and if they have the same count, pops both and replaces them with a new entry with double the count. Doing this repeatedly results in a tree structure because each iteration pushes a new memoization entry that encompasses two lower entries. Note that the repetition arises because ip is only updated in the second branch of the if statement.

```
1: e_1 ← S_1
2: e_2 ← S_2
3: (id_1, pos_1, count_1)_memo ← e_1
4: (id_2, pos_2, count_2)_memo ← e_2
5: if id_1 = id_2 ∧ count_1 = count_2 then
6:     S, _ ← PopAndProp(S)
7:     S, _ ← PopAndProp(S)
8:     m ← (id_2, pos_2, sp − pos_2, ep − pos_2, ⟨⟩, 2 · count)
9:     M ← M[(id_2, pos_2) ↦ m]
10: else
11:     ip = ip + 1
```

- `MemoTreeClose` id: This instruction is a cleanup instruction that repeatedly removes memoization entries from the stack if they match id. This is just used at the end of tree memoization to clear the stack.

```
1: (id_1, pos, count)_memo ← S_1
2: if id = id_1 then
3:     S, _ ← PopAndProp(S)
4: else
5:     ip ← ip + 1
```

Now we can compile repeated memoization, `{{ p }}*`, using these instructions.

```
L1: MemoTreeOpen L3 ID
    Choice L2
    <p>
    Commit LN
LN: MemoTreeInsert
L3: MemoTree
    Jump L1
L2: MemoTreeClose
```

Since memoization entries track counts, the tree will be properly reconstructed after each edit.

Dummy captures

There is a small problem with the current implementation of tree memoization when captures are involved. Since `MemoTree` uses PopAndProp it will perform a linear amount of capture copying to perform propagation, even in cases where the overall reparse time would be logarithmic otherwise. In other words, we made the memoization entries have a tree

1: $e_1 \leftarrow S_1$
2: $e_2 \leftarrow S_2$
3: $(id_1, pos_1, count_1)_{memo} \leftarrow e_1$
4: $(id_2, pos_2, count_2)_{memo} \leftarrow e_2$
5: **if** $id_1 = id_2 \wedge count_1 = count_2$ **then**
6: $\quad S, _ \leftarrow \text{POP}(S)$
7: $\quad c \leftarrow (\mathbf{dummy}, \emptyset, e_1^{caps})$
8: $\quad e_2^{caps} \leftarrow e_2^{caps} :: c$
9: $\quad S, _ \leftarrow \text{POPANDPROP}(S)$
10: $\quad m \leftarrow (id_2, pos_2, sp - pos_2, ep - pos_2, \langle\rangle, 2 \cdot count)$
11: **else**
12: $\quad ip = ip + 1$

FIGURE 4.8: New semantics for `MemoTree` to enforce a tree structure on user captures. The special ID **dummy** is used for these captures.

structure, but if the capture hierarchy requested by the user is flat, reparse time will be linear because of the cost to create the capture result.

To solve this, we use "dummy captures" – captures which only store children. These are inserted during `MemoTree` to enforce a tree structure on the user's capture result. The dummy captures can transparently provide their children when requested, so with the correct interface the user need not be aware of their existence.

The new semantics for `MemoTree` are shown in figure 4.8. Thanks to these new instructions, we can perform incremental parsing of flat grammars and even pattern-matching grammars very efficiently.

4.4.3 Automatic Memoization

Tree memoization is built on the insight that repetition causes inefficient linear patterns in the memoization table. So far, we have assumed that the user specifies which patterns should be memoized. It seems likely that memoizing all repetitions with tree memoization provides a good baseline, something that could be done automatically to any existing grammar to make it amenable to efficient incremental parsing. This is a promising topic for future investigation.

Chapter 5

Evaluation

This chapter presents the evaluation of the techniques presented using the prototype implementation library called GPeg. Though GPeg has been designed primarily with incremental parsing in mind, it is a multi-purpose parsing engine and as such we evaluate it under multiple different workloads.

We first test GPeg for incremental and non-incremental parsing of small and large language grammars, using JSON and Java as examples. While GPeg enables incremental parsing, it should not significantly impair non-incremental parsing. This section seeks to evaluate GPeg's non-incremental performance, as well as the overhead of memoization for incremental parsing. We test incremental reparsing, but only for random single character edits. This shows the efficiency of using an interval tree for the memoization table, and tree memoization for reparsing, but does not bring in the overheads of shifts, and is therefore a best-case analysis.

Next we evaluate GPeg on a complete syntax highlighting workload to measure the overall performance of the system. We choose syntax highlighting because it is the most immediately available application of GPeg, and the workload for which we most intend to use GPeg. We show how to build a small syntax highlighting library on top of GPeg and analyze its performance in both initial and incremental parse times while comparing to some existing syntax highlighting algorithms.

Finally, since GPeg is directly inspired from LPeg, which is meant for pattern matching (as a replacement for regular expressions), we test GPeg's performance in pattern matching (i.e. tiny grammars).

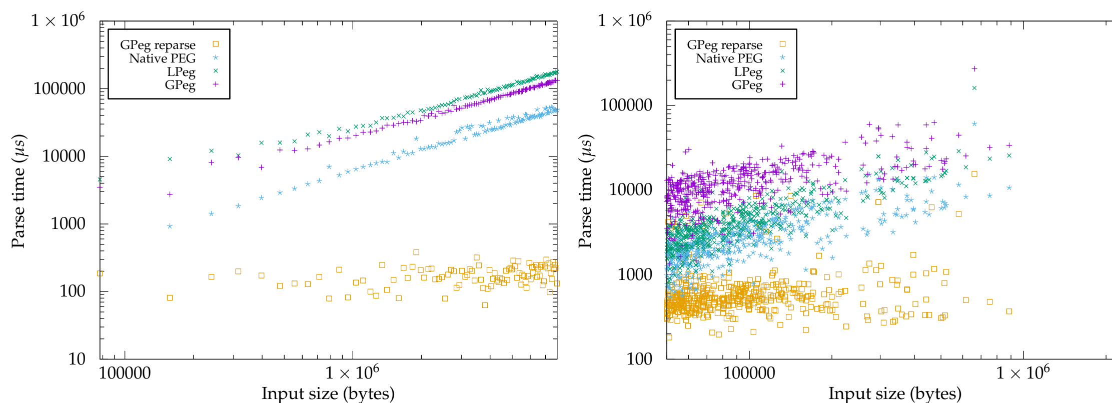
All experiments are performed on a computer running Linux 5.4 with an AMD Ryzen 5 1600. When running code in the experiments, we use the Go 1.16 compiler, the Lua 5.1 VM, and LPeg 2 (compiled from the C source code). For benchmarks involving parsing, the entire input is loaded into memory before measurement begins.

5.1 Language Parsing

In this section we test GPeg for the case of parsing from a complete language grammar. The two languages we select for testing are JSON and Java. We use LPeg as a point of comparison for another dynamic parser, and a Peg library [27] as a point of comparison for a native PEG parser (and as such, we expect this library to perform better than both GPeg and LPeg).

These experiments serve to answer the following questions:

1. How does GPeg's performance compare with other PEG parsers when all incremental parsing functionality is disabled and not needed?



(A) JSON parser performance on files ranging in size from 100KB to 8MB.

(B) Java parser performance on files ranging in size from 50KB to 2.5MB.

FIGURE 5.1: Parser performance for JSON and Java datasets.

2. How does GPeg perform for incremental parsing tasks over a significant number of edits (hundreds to thousands)?

3. What is the memory overhead of the memoization table?

4. Does tree memoization remain effective after many edits?

The following experiments have limited scope to answer these particular questions.

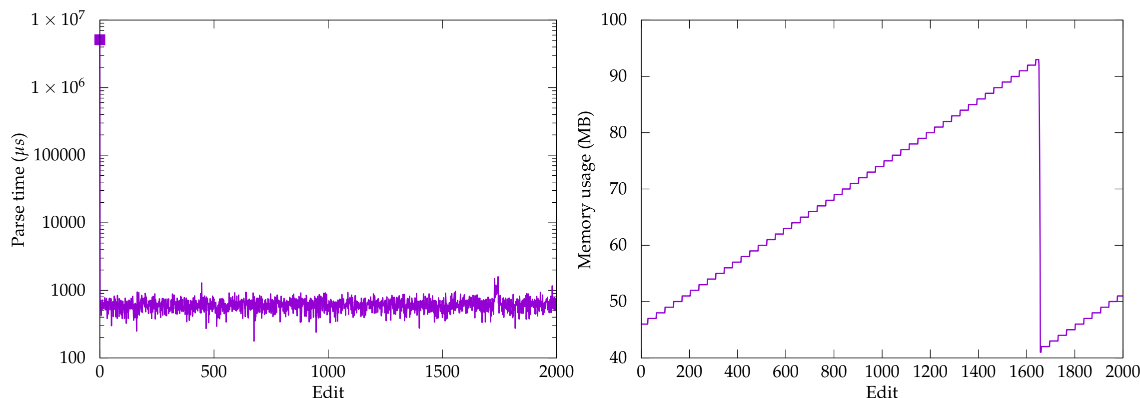
Figure 5.1 shows parse times for the three parsers in question, as well as the time for GPeg's first incremental reparse. The JSON dataset is a synthetic dataset that has been randomly generated using a representative template. The Java dataset consists of all files above 50KB in two large Java 1.7 projects: OpenJDK7 [16] and the Ceylon compiler [8]. Note that in all cases, only matching is performed (no AST is generated). Pure matching performance avoids measuring capture generation costs, and avoids imposing a certain structure of AST to capture since different applications may choose different structures.

We find that GPeg and LPeg have comparable performance, with GPeg performing slightly better for JSON and LPeg performing slightly better for Java. While the full parse times scale linearly in the size of the file, GPeg's incremental parse time scales logarithmicly in the size of the file.

The next experiment, shown in figure 5.2, performs random, single character edits to a 100MB JSON file. This displays the power of the logarithmic complexity that tree memoization and the interval tree provide. Reparses are roughly four orders of magnitude faster than the initial parse, and well within an editor's window redraw target of 10-20ms. Parser memory usage is good as well, considering the file is 100MB. The memoization threshold prevents the table from storing many small entries, which is usually the cause of high memory usage in packrat parsing.

We perform a similar experiment for Java, shown in figure 5.3. We use a 250KB Java file, one of the largest real-world Java examples we could find.[1] Performance and memory usage are promising, though the benefits of incremental parsing are not as pronounced.

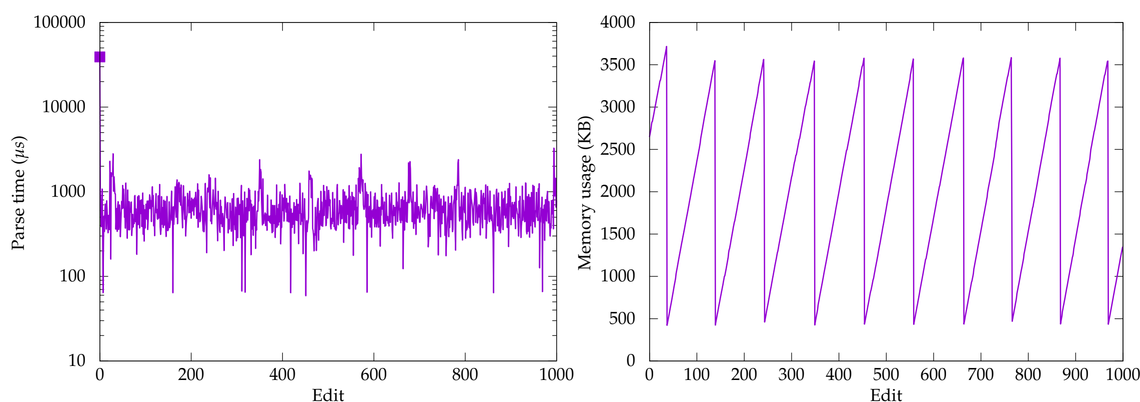
[1]Many of the larger files in the OpenJDK7 source code consist nearly solely of long strings storing locale information.



(A) JSON parser performance over time. Reparsing is roughly four orders of magnitude faster than the initial parse.

(B) JSON parser memory usage over time, almost all of which is the memoization table. The table is roughly half the size of the input, and garbage (unfreed evicted entries) grows by roughly 10MB every 300 edits.

FIGURE 5.2: JSON parser performance on a real-world 100MB JSON file [20]. Go's garbage collection runs around edit 1600, causing a small spike in reparse time.



(A) Java parser performance over time.

(B) Java parser memory usage over time.

FIGURE 5.3: Java parser performance on a 250KB Java file.

These experiments show the effectiveness of tree memoization and the interval tree. In the next section, we show that for clustered edits, shift overhead is minimal. We also find that captures are a much larger source of memory usage compared to the memoization table, offering a good potential future direction for optimization.

5.2 Case Study: Syntax Highlighting

One of the primary motivations for developing this library is for the use of syntax highlighting. While GPeg supports general incremental parsing, syntax highlighting is one of the most common workloads where incremental parsing is needed.

On top of evaluating the case of syntax highlighting, this chapter helps to answer additional questions about GPeg:

- What is the overhead of shifting in the memoization table's interval tree?
- What is the memory overhead of generating relocatable captures?
- How does the memoization entry threshold[2] affect speed and memory usage?

5.2.1 Existing Approaches

Most text editors use either a horizon-based or line-based approach to syntax highlighting. Each method has advantages and disadvantages. A horizon-based highlighter is more general and flexible, but either produces an incorrect result or is very slow. A line-based approach is more constraining for the grammar writer, but is very efficient. We build a highlighter with GPeg that we believe gives the best of both worlds. First we present the details of how each existing method works. Other methods include blends of the horizon and line-based approaches (e.g., in Nano) or full incremental parsing using a GLR parser (only implemented by Tree-Sitter [11]).

Horizon-based Highlighters

The horizon-based approach is the simplest because it does not attempt any sort of incremental parsing. Instead it only considers the portion of the text document near the viewable area. The highlighter only operates only considers text within a certain horizon and ignores the rest. While this gives a strictly bounded reparse time, the highlighting may be incorrect (if a long multiline comment begins outside the horizon, the highlighter will not be aware and will treat the comment as source code when highlighting). An existing horizon-based syntax highlighting library is Scintillua [24], which uses LPeg. An advantage of this approach is that any grammar can be used for highlighting, and no structure is imposed on the grammar for the purposes of syntax highlighting support.

Text editors that use a horizon-based highlighting approach include Vis [28] and SciTE [13].

[2] To save memory, we do not memoize matches that have fewer than n examined characters, where n is the memoization threshold.

Line-based Highlighters

The line-based approach [2] is more common because it produces a fully correct highlighting result while still maintaining efficient reparsing. A key insight made by the line-based approach is that an edit to a line will only ever change the highlighting in that line or in lines below it. The grammar is specified as a set of regions and regular expressions. Each region corresponds to a certain state of the parser (e.g., normal, comment, string, nested language...). A set of regular expressions is associated with each region and specify how to match various types of language constructs (e.g., keywords, identifiers...). Usually every regular expression in the current region's set is applied to each line in the region. After initial tokenization, each line stores the state it ended in.

Most edits to a line will cause retokenization of only the current line.[3] Sometimes an edit will also cause retokenization of lines below it (such as the insertion of a multiline comment).[4] Retokenization continues on each line until a line end state does not change (it is the same after retokenization as before). Since the parser has reached an equivalent state to where it was before the edit, it knows that the remainder of the file is accurately tokenized.

The line-based approach is very efficient but has several disadvantages:

- It imposes constraints on the highlighting grammar, and the types of syntax highlighting that can be performed.
- It is tied to the concept of lines, which is often limiting. One problem is that the highlight time grows with the size of the line. You might find that an editor stops performing syntax highlighting if a line becomes too long.[5]

Text editors that use a line-based highlighting approach include Visual Studio Code, Micro, and probably Vim and Sublime Text.[6]

5.2.2 Building a Syntax Highlighter

We can use GPeg to create a syntax highlighting library, and this approach combines the best of the horizon and line-based approaches. GPeg does not impose any unwanted structure on syntax highlighting grammars, and they can be as complex as the author desires. Nonetheless, GPeg supports fast incremental parsing which means that highlighting will be fully correct and fast.

In this section we will show how to create a Java syntax highlighting grammar that can generalize well to many languages. It is simple in the sense that it does not do a full parse of the language. Instead this grammar has definitions for various kinds of tokens, and if nothing is matched the parser just consumes the character as an unknown token type. Once parsing is complete, the highlighter returns the documented as a list of captures, each annotated with the token type. A visualizer can then apply a theme which maps token types to colors and display the file.

We identify the following token types for Java (but they are very general): `whitespace`, `class`, `keyword`, `type`, `function`, `identifier`, `string`, `comment`, `number`, `annotation`, `operator`, `special`.

[3] Visualization: https://code.visualstudio.com/assets/blogs/2017/02/08/tokenization-1.gif.
[4] Visualization: https://code.visualstudio.com/assets/blogs/2017/02/08/tokenization-2.gif.
[5] Vim's `synmaxcol` option controls when a line is too long to perform highlighting.
[6] More investigation into these editors is needed to determine their exact highlighting strategy

The grammar defines each token type as a non-terminal, which specifies how to match that token. The core of the highlighter then tries to repeatedly match tokens. If a token does not match at the current character, the highlighter consumes "unknown" characters until a token does match. We can express this with the following pattern:

```
{{ token / . (!token .)* }}*
```

It is important to use `(!token .)*` so that contiguous unknown characters get memoized together.

Since we are repeating a memoized pattern, we will benefit greatly from the tree memoization strategy for incremental parsing. The `token` non-terminal attempts to match one of the token types:

```
token <- whitespace / class / keyword / type / function / identifier / string
       / comment / number / annotation / operator / special
```

Note that the order of matching is important. More specific tokens should be attempted before others (e.g., keyword should match before identifier).

Each token can then be defined to parse the specific construct in the language. For example, the `comment` token for Java can be defined as

```
comment       <- line_comment / block_comment
line_comment  <- '//' (!'\n' .)*
block_comment <- '/*' (!'*/' .)* '*/'?
```

Note that the end of the `block_comment` is optional. This ensures that block comments are highlighted even if they have no ending marker (they are highlighted until the end of the document). This is a common behavior among syntax highlighters, though the reason this is common is that line-based highlighters must do this (a deletion of the end delimiter cannot modify the highlighting of any text above it in the document). GPeg is more powerful than line-based highlighters, meaning it can support either behavior.

5.2.3 Performance

We compare the performance of the GPeg highlighter with several other syntax highlighters:

- Scintillua [24]: a horizon-based highlighter built on top of LPeg.
- Chroma [29]: a horizon-based highlighter built as a reimplementation of the popular Pygments library in Go. Chroma uses regular expressions and a stack in much the same way as many line-based highlighters. Chroma is not built to be used in a text editor.
- Micro highlight [35]: a line-based highlighter used in the Micro text editor.

In the following experiments, the horizon-based highlighters use the full horizon to ensure that they are fully correct. As a result, we only use the horizon-based highlighters when measuring initial parse time.

We test the highlighters on a large Java file that is roughly 3.4MB constructed by appending many OpenJDK7 source code files together. This creates a source file with a wide range

Highlighter	Time (ms)
GPeg	451
Scintillua	178
Chroma	2886
Micro	627

TABLE 5.1: Initial highlight time for a 3.4 MB Java file.

of constructs and represents real-world Java code. Initial highlight times are shown in table 5.1.

Testing incremental highlighting involves making edits to the document. In this case, we must be careful to avoid high garbage collection costs associated with the edits to the document (unrelated to incremental parsing performance). For example, using a basic array for the text will be highly inefficient because insertions and deletions are expensive and create garbage when the array needs to be reallocated. To minimize these costs, we store the document in a rope data structure.[7]

In the first workload, edits are simulated in clusters of 100 edits to the same location of the document. This is the most typical workload. We also test rehighlighting for completely random edits. This workload stresses shift application because most edits are to locations that have not had shifts applied recently. Rehighlighting performance for both workloads are shown in figure 5.4.

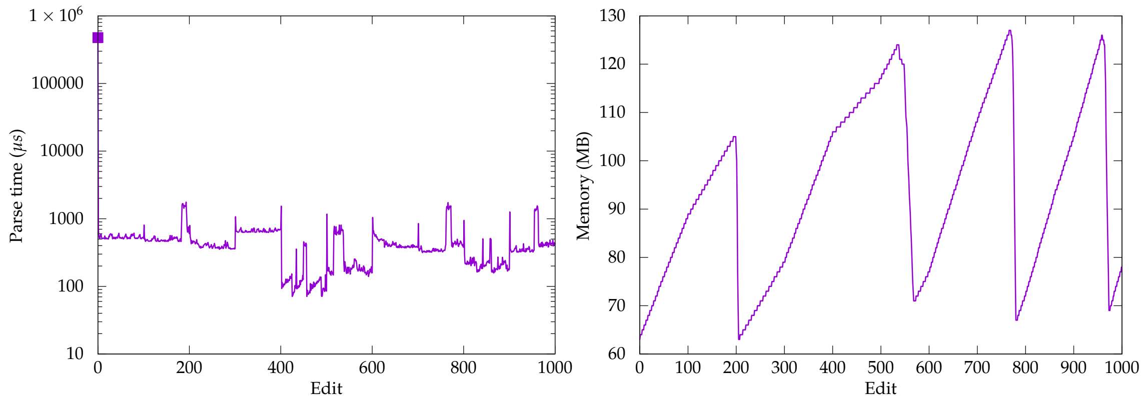
GPeg has an average reparse time of 430 μs for the clustered edit case. The Micro highlighter is more efficient, with an average reparse time of 70 μs. However, this difference is not significant for an editor's redraw window target of 10-20ms. In addition, the micro highlighter's performance is much more dependent on the size of the grammar and line length, because it applies every regular expression in the region set to each line, whereas GPeg's prioritized choice allow it to stop attempting tokens once a match has been found (this is also partly responsible for GPeg's better initial highlight time).

The memoization entry threshold plays an important role in the space-performance tradeoff. Figure 5.5 shows the tradeoff between non-garbage memory usage and rehighlight time. In the earlier experiments, we used a memoization threshold of 128. Since gains in memory usage flatten at a memoization threshold of 4096, we know that the remaining significant memory usage is in the capture result rather than the memoization table (roughly 60MB to store captures).

5.2.4 Discussion

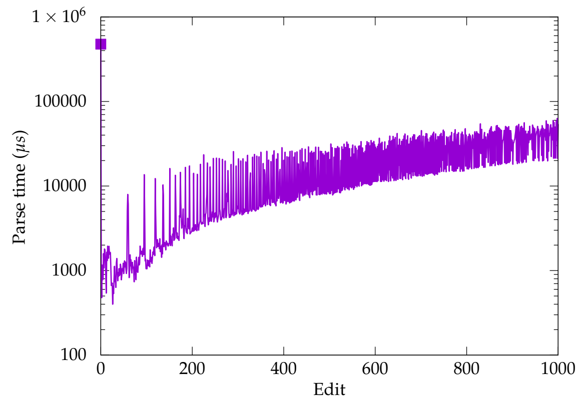
The PEG-based highlighters are the most performant for the initial parse. There is no reason to believe that the Go regexp implementation is more performant than the PEG implementation and the line-based approach must apply multiple regular expressions to every line. In contrast, the PEG parser knows the exact state of the parser and can therefore apply the necessary instructions for parsing the current token exactly. LPeg is faster than GPeg, likely for

[7] One understated benefit of using GPeg over LPeg or other parsers is that GPeg operates on Go's `io.ReaderAt` interface, meaning it can parse text stored in text sequence data structures such as ropes, gap buffers, and piece tables. This is very important for use in a text editor.



(A) Syntax rehighlighting performance for clustered edits. Clustered edits avoid the shift application degradation seen with random edits.

(B) Memory usage during syntax rehighlighting for clustered edits.



(C) Rehighlight time for random edits.

FIGURE 5.4: (A) and (B) show the performance of rehighlighting over 1000 clustered edits. Each time the location switches (every 100 edits), we see a spike in latency as the system lazily performs shifts before settling into a part of the document where most shifts have already been applied. (C) shows rehighlighting time for random edits. In this case we see significant performance degradation due to continuous shift application and increased garbage generation/collection. Highlighting is performed on a 3.4MB Java file.

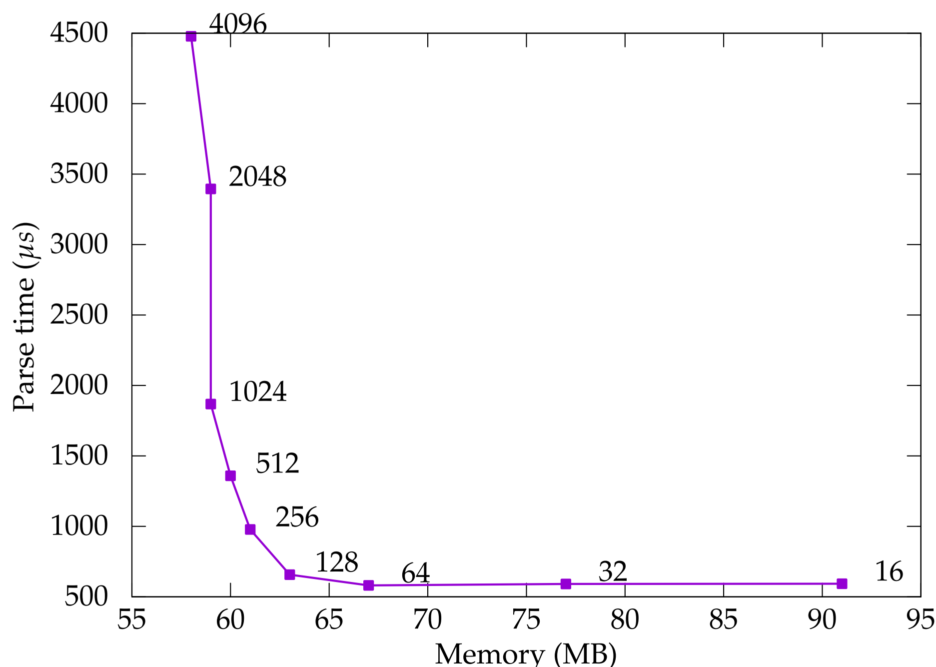


FIGURE 5.5: Tradeoff between speed and space. Each point marks a different choice of memoization threshold. This figure also demonstrates that the majority of memory usage comes from the captures rather than from the memoization table.

two reasons: LPeg is written in C rather than Go, and LPeg performs a light form of inlining that is more efficient for functions that frequently fail on the first instruction.

The Micro highlighter has the most efficient reparse by a large margin. The line-based approach in general will be very efficient because it is constant-time (assuming accessing a line is constant time). Nevertheless, GPeg reparses well within a text editor's redraw target time of 10-20ms. In addition, GPeg is more flexible and powerful than the line-based approach in many ways. It does not impose a structure on the grammar, and can express much more complex syntax highlighting more easily than a line-based regular expression approach. It also does not require that the editor maintain line information in the same way as a line-based approach. GPeg also enables more advanced highlighting because it can highlight constructs where a modification to one line causes a change to highlighting in a line above it (this is impossible in the line-based approach). GPeg can also use the language's full grammar for highlighting/parsing.

Finally, we can see that GPeg uses a lot of memory. Nearly all of this memory is used by the capture result, not the memoization table, which is a good sign. The capture result can be optimized for the particular workload of syntax highlighting, and we intend to look into optimizing captures more generally in the future. We expect that capture memory usage will be much less of a problem once the API has been improved, but the work there is still ongoing.

Given the additional flexibility and highlighting power GPeg provides while still being well within the necessary performance threshold, we expect to replace the Micro highlighter with a GPeg-powered highlighter in the future. There are of course some challenges on

that front, namely the creation of a DSL specifically for syntax highlighting, and the actual creation of grammars that cover the 100+ languages Micro supports.

5.3 Pattern Matching

While search and pattern matching is not the primary goal of GPeg, it is still worthwhile to compare the performance on this task against LPeg. The following benchmarks were performed in by Ierusalimschy [14], and we have reproduced them with GPeg. They involve searching for various patterns in the King James Bible[8].

Since PEGs run in "anchored mode" meaning that unlike regular expressions, matching always starts at the beginning of the subject string, searches must be expressed with recursion. For example, we can search for the first occurrence of a pattern `p` like so:

```
S <- p / . S
```

This tries to match `p` and if it fails, it consumes a character and tries again. To search for the last occurrence of a pattern `p`, we repeat the search as much as possible:

```
X <- S*
S <- p / . S
```

Tables 5.2 and 5.3 show performance results for searching for the first and last occurrences (respectively) of various patterns in the King James Bible. All benchmarks compare GPeg and LPeg.

While it would be possible to also include captures, the benchmarks only perform matching, and only the match time is recorded (the time to load the Bible into memory is not included). The last benchmark manually encodes the search, and thus the pattern is directly matched (it is not wrapped in a search grammar).

Since GPeg provides a search operator to automatically wrap patterns in search grammars, that feature is also used, and is marked as GPeg (2). The search operator in GPeg may perform additional optimization as described in section 3.3.4 on the dedicated search opterator.

GPeg has comparable performance to LPeg, though it is slightly slower. This is most likely because of the increased stack size to handle captures, and some overhead from Go. GPeg outperforms LPeg with the search optimization enabled, which automatically rewrites search grammars to a more optimal form for certain cases. It is possible to also manually perform this optimization in LPeg, which results in similar performance to GPeg.

Table 5.4 shows benchmarks for matching recursive grammars in the King James Bible. LPeg still outperforms GPeg despite not implementing inlining. Despite the lack of inlining, LPeg still does some optimizations around function calls, such as doing a test before the call to check whether the first instruction of the function will succeed (if the first instruction is one of `Char`, `Set`, `Any`). Since the string "Omega" is very rare, the function is rarely called so GPeg's advantage from inlining is nullified.

We can conclude that GPeg achieves comparable performance to LPeg for searching, but there are overheads, primarily from including captures in the stack. GPeg needs to include captures in the stack so that captures can be saved into memoization entries for incremental parsing.

[8] `http://www.gutenberg.org/cache/epub/10/pg10.txt`.

Pattern	LPeg	GPeg	GPeg (2)
`’@the’`	52	67	13
`’Omega’`	52	64	14
`’Alpha’`	41	52	13
`’amethysts’`	59	71	25
`’heith’`	60	75	27
`’eartt’`	64	78	33
`[a-zA-Z]+ [ \n]* ’Abram’`	1.8	2.2	2.3
`[a-zA-Z]+ [ \n]* ’Joseph’`	6.8	6.8	6.8

TABLE 5.2: Searches for the first occurrence of the given pattern in the bible. Times shown in milliseconds. GPeg (2) indicates GPeg with search optimization enabled.

Pattern	LPeg	GPeg	GPeg (2)
`’@the’`	51	66	13
`’Omega’`	55	67	13
`’Tubalcain’`	55	66	18
`[a-zA-Z]+ [ \n]* ’Abram’`	205	252	259

TABLE 5.3: Searches for the last occurrence of the given pattern in the bible. Times shown in milliseconds. GPeg (2) indicates GPeg with search optimization enabled.

Pattern	LPeg	GPeg
`S <- ’Omega’ / . S`	51	68
`(!’Omega’ .)* ’Omega’`	71	74
`S <- (!P .)* P ; P <- ’Omega’`	70	74

TABLE 5.4: Time (in milliseconds) to match the pattern in the bible.

Grammar	Code size	Padding size	Serialized size
Arithmetic	70	5	201
JSON	478	22	477
Peg	2178	145	1324
C	22628	1507	10957
Java	26520	1619	12471

TABLE 5.5: Sizes (bytes). Note: code size does not include the size of the set table (serialized size does include this, but also applies compression).

5.3.1 Pattern Matching and Memoization

Incremental pattern matching is not a workload that sees much usage, but the structure of the grammar is not much different from the syntax highlighting grammar (which essentially performs a search for language tokens). The main application here would be highlighting search results in a text editor. However, most editors use regular expressions rather than PEGs for searching, so from a practical perspective one would first have to convert the regular expression into a PEG to use GPeg for this use-case. Another, possibly more far-fetched, text editor application could be making pattern-based marks. A set of (possibly user-specified) patterns could define constructs like function definitions and the editor could then allow the user to jump to these locations, while maintaining all the matching locations using incremental parsing.

5.4 Encoding

Code size can affect performance, and is important to consider for storing compiled PEGs. Table 5.5 compares the code sizes across various grammars. The serialized size includes the set table, and uses gzip compression to reduce storage size.

Chapter 6

Related Work

6.1 Incremental Parsing

Incremental parsing was first described by Ghezzi and Mendrioli [10] [9] for LR parsers. Subsequent research focused on improving the performance of shift-reduce parsers [31] [32] [33] and selecting the optimal subset of text to reparse to attain optimal parse tree reuse [19].

There is previous work in incremental parsing for top-down parsing. Most methods focus on LL(1) grammars [26] [25], which are similar to PEGs but do not support unlimited lookahead, and are therefore more restrictive. Many of these algorithms require special integration with an editor, or only support single-site editing. Incremental packrat parsing [4] presents an algorithm for incremental PEG parsing, which this thesis builds on. Incremental packrat parsing handles unlimited lookahead, multi-site editing, and does not require significant, if any, modification to the grammar.

There are multiple existing projects that seek to provide incremental parsing for both CFGs and PEGs.

Tree Sitter [11] is a project by GitHub for incremental parsing, with an emphasis on syntax highlighting. It generates full parse trees for the language so it can be used for more general analyses as well. Tree sitter uses Generalized LR parsing with techniques presented in earlier research [31] [30] [32] [33]. The project uses context-free grammars, and a domain-specific subset of JavaScript for defining grammars. Parsers written in C are statically generated from the grammars by a Rust program.

Ohm [12] is an incremental PEG parser library using incremental packrat parsing [4]. The project is written in JavaScript and uses a custom PEG format in a subset of JavaScript to define grammars. In addition, Ohm memoizes all non-terminals, which results in large space overhead.

Papa Carlo [18] is an incremental PEG parser library written in Scala. Papa Carlo defines "fragments" which are memoized and only reparses those fragments when a change is made. Fragments must be defined by the user and are restricted to certain types of non-terminals. In particular, Papa Carlo lists that a fragment must follow certain properties: it must be "simply determined as a code sequence between two tokens" and its "syntactical meaning [must be] invariant to [its] internal content. At least in most cases." The project does not formally define these properties.

6.2 PEG Machines

LPeg [15] is a Lua library that uses a PEG parsing machine [21] [14]. GPeg is inspired from GPeg and uses many of the same instructions. LPeg is written in C but is only available as

a Lua library. LPeg has become very popular in the Lua community and is widely used. It is primarily used as a replacement for regular expressions, but supports a wide variety of capture methods allowing it to be used to generate ASTs for language grammars as well.

NPeg [3] is a PEG parsing machine written for the Nim programming language. It uses the same techniques as LPeg and GPeg, but it implements the parsers as Nim macros, meaning it only supports defining grammars at compile-time. This results in highly efficient native parsers, but is not well-suited to being used as a pattern matcher or syntax highlighter.

Chapter 7

Conclusion

This thesis describes some new techniques in incremental PEG parsing, as well as how to integrate these techniques into a parsing machine implementation. A prototype implementation called GPeg is available on GitHub and consists of about 4,500 lines of Go source code. We perform an evaluation for a wide variety of grammars and show the effectiveness of implementing the memoization table as an interval tree, using tree memoization for repetition, and lazy shifts.

There are multiple next steps possible, both for incremental parsing and the GPeg parsing library more generally.

7.1 Future Work

Throughout the thesis we have mentioned some areas for future work, including improving the lazy shift algorithm, automatic memoization, and automatic error recovery. Below are some additional topics for future work.

7.1.1 Text Editor Integration

While the current version of the GPeg library works well, the most important next steps will involve making it ready for use in other projects. The primary application that I intend to target is the Micro text editor, in particular using GPeg to replace the current syntax highlighter, and implement code folding.

These techniques can also be integrated into other editors, IDEs, and language servers.

7.1.2 Parallel Parsing

On multicore machines we believe non-incremental parse times can be improved significantly by performing parallel packrat parsing. The technique involves spawning a new virtual machine instance in a separate thread to begin parsing at a later point in the file. This worker parser will fill the memoization table as it parses, and the hope is that once the main thread arrives at the later file location, the table will be full of useful parse results allowing the main thread to skip them. This should be especially effective with tree memoization.

This technique relies on the worker parser being spawned in a "good" state, or in a state that is able to reach a good state. For light grammars, this isn't a problem since almost all points of the file tend to match the top-level pattern. For more complex grammars, an error recovery mechanism can be used to recover the worker parser into a state where it can start generating useful entries in the memoization table.

References

[1] Philip Dexter, Yu David Liu, and Kenneth Chiu. “Lazy graph processing in haskell”. In: *ACM SIGPLAN Notices* 51.12 (2016), pp. 182–192.

[2] Alexandru Dima. *Optimizations in Syntax Highlighting*. URL: https://code.visualstudio.com/blogs/2017/02/08/syntax-highlighting-optimizations.

[3] Ico Doornekamp. *NPeg*. URL: https://github.com/zevv/npeg.

[4] Patrick Dubroy and Alessandro Warth. “Incremental packrat parsing”. In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. 2017, pp. 14–25.

[5] Jeffrey Eymer, Philip Dexter, and Yu David Liu. “Toward lazy evaluation in a graph database”. In: *SPLASH 2019* (2019).

[6] Bryan Ford. “Packrat parsing: simple, powerful, lazy, linear time, functional pearl”. In: *ACM SIGPLAN Notices* 37.9 (2002), pp. 36–47.

[7] Bryan Ford. “Parsing expression grammars: a recognition-based syntactic foundation”. In: *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2004, pp. 111–122.

[8] Eclipse Foundation. *Ceylon*. URL: https://github.com/eclipse/ceylon.

[9] Carlo Ghezzi and Dino Mandrioli. “Augmenting parsers to support incrementality”. In: *Journal of the ACM (JACM)* 27.3 (1980), pp. 564–579.

[10] Carlo Ghezzi and Dino Mandrioli. “Incremental parsing”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1.1 (1979), pp. 58–70.

[11] GitHub. *Tree Sitter*. URL: https://tree-sitter.github.io/tree-sitter/.

[12] HARC. *Ohm webpage*. URL: https://ohmlang.github.io/.

[13] Neil Hodgson. *SciTE*. URL: https://www.scintilla.org/SciTE.html.

[14] Roberto Ierusalimschy. “A text pattern-matching tool based on Parsing Expression Grammars”. In: *Software: Practice and Experience* 39.3 (2009), pp. 221–258.

[15] Roberto Ierusalimschy. *LPeg: Parsing Expression Grammars for Lua*. URL: http://www.inf.puc-rio.br/~roberto/lpeg.

[16] *JDK 7*. URL: https://openjdk.java.net/projects/jdk7/.

[17] Donald E Knuth. “Top-down syntax analysis”. In: *Acta Informatica* 1.2 (1971), pp. 79–110.

[18] Ilya Lakhin. *Papa Carlo*. URL: https://lakhin.com/projects/papa-carlo/.

[19] J-M Larchevêque. “Optimal incremental parsing”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 17.1 (1995), pp. 1–15.

[20] City of Los Angeles. *Crime Data from 2020 to Present*. URL: https://catalog.data.gov/dataset/crime-data-from-2020-to-present.

[21] Sérgio Medeiros and Roberto Ierusalimschy. "A parsing machine for PEGs". In: *Proceedings of the 2008 symposium on Dynamic languages*. 2008, pp. 1–12.

[22] Sérgio Medeiros and Fabio Mascarenhas. "Syntax error recovery in parsing expression grammars". In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. 2018, pp. 1195–1202.

[23] Sérgio Queiroz de Medeiros and Fabio Mascarenhas. "Error recovery in parsing expression grammars through labeled failures and its implementation based on a parsing machine". In: *Journal of Visual Languages & Computing* 49 (2018), pp. 17–28.

[24] Mitchell. *Scintillua*. URL: https://orbitalquark.github.io/scintillua/index.html.

[25] Arvind M Murching, YV Prasad, and YN Srikant. "Incremental recursive descent parsing". In: *Computer Languages* 15.4 (1990), pp. 193–204.

[26] John J. Shilling. "Incremental LL (1) parsing in language-based editors". In: *IEEE transactions on software engineering* 19.9 (1993), pp. 935–940.

[27] Andrew Snodgrass. *Peg*. URL: https://github.com/pointlander/peg.

[28] Marc André Tanner. *Vis Editor*. URL: https://github.com/martanne/vis.

[29] Alec Thomas. *Chroma*. URL: https://github.com/alecthomas/chroma.

[30] Eric R Van Wyk and August C Schwerdfeger. "Context-aware scanning for parsing extensible languages". In: *Proceedings of the 6th international conference on Generative programming and component engineering*. 2007, pp. 63–72.

[31] Tim A Wagner. "Practical algorithms for incremental software development environments". PhD thesis. Citeseer, 1997.

[32] Tim A Wagner and Susan L Graham. "Efficient and flexible incremental parsing". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 20.5 (1998), pp. 980–1013.

[33] Tim A Wagner and Susan L Graham. "Incremental analysis of real programming languages". In: *ACM SIGPLAN Notices* 32.5 (1997), pp. 31–43.

[34] Zachary Yedidia. *Micro Editor*. URL: https://github.com/zyedidia/micro.

[35] Zachary Yedidia. *Micro Highlight*. URL: https://github.com/zyedidia/micro/tree/master/pkg/highlight.

Appendix A

Parsing Machine Specification

A.1 Semantics

The machine state is

$$\langle ip, sp, S, C, M, ep \rangle \in \mathbb{N}_\perp \times \mathbb{N} \times \textbf{Stack} \times \langle \textbf{Capture} \rangle \times \textbf{MemoTable} \times \mathbb{N}$$

where ip, sp, and ep are the instruction pointer, subject pointer, and examined pointer respectively. S is the stack, C is the top-level capture list, and M is the memoization table.

The top-level capture list is a list of captures of the form:

$$(id, content, children) \in \mathbb{N} \times \textbf{Content} \times \langle \textbf{Capture} \rangle.$$

The memoization table is a key-value store mapping keys of the form (id, sp) to entries of the form

$$(id, sp, len, exam, caps, count) \in \mathbb{N} \times \mathbb{N} \times \mathbb{N}_\perp \times \mathbb{N} \times \langle \textbf{Capture} \rangle \times \mathbb{N}.$$

The stack is a list of entries where S_1 is the top and $S_{|S|}$ is the bottom. Stack entries are

- Return entries: $(ip)_{ret}$.
- Backtrack entries: $(ip, sp)_{bt}$.
- Capture entries: $(id, sp)_{cap}$.
- Memoization entries: $(id, sp, count)_{memo}$.

Each entry e additionally stores a list of captures:

$$\langle e, caps \rangle \in \textbf{StackEntry} \times \langle \textbf{Capture} \rangle$$

We use the notation e^{caps} to refer to an entry's capture list.

Popping from the stack may propagate captures if the PopAndProp function is used:

```
1: procedure Pop(S)
2:     e ← S_1
3:     S ← S_{2...|S|}
4:     return S, e
```

```
1: procedure PopAndProp(S)
2:     e ← S_1
3:     if |S| > 1 then
4:         S_2^caps ← S_2^caps :: e^caps
5:     else
6:         C ← C :: e^caps
7:     S ← S_{2...|S|}
8:     return S, e
```

Basic instructions

- `Char` b

```
1: if I[sp] = b then
2:     ip ← ip + 1
3:     sp ← sp + 1
4:     ep ← max(sp, ep)
5: else
6:     ip ← ⊥
```

- `Jump` l

```
1: ip ← l
```

- `Choice` l

```
1: S ← (l, sp)_bt :: S
```

- `Call` l

```
1: S ← (ip + 1)_ret :: S
2: ip ← l
```

- `Commit` l

```
1: S, _ ← PopAndProp(S)
2: ip ← l
```

- `Return`

```
1: S, (ip_r)_ret ← PopAndProp(S)
2: ip ← ip_r
```

- `Fail`

```
1: ip ← ⊥
```

- `End`: ends matching and accepts the subject.
- `EndFail`: ends matching and fails the subject.
- When $ip = \bot$

```
1: while |S| > 0 do
2:     S, e ← Pop(S)
3:     if (ip_1, sp_1)_bt := e then
4:         ip ← ip_1
5:         sp ← sp_1
```

Additional basic instructions

- `Set` X

1: **if** $I[sp] \in X$ **then**
2: $ip \leftarrow ip + 1$
3: $sp \leftarrow sp + 1$
4: $ep \leftarrow max(sp, ep)$
5: **else**
6: $ip \leftarrow \bot$

- `Any` n

1: **if** $sp + n \leq |I|$ **then**
2: $ip \leftarrow ip + 1$
3: $sp \leftarrow sp + n$
4: $ep \leftarrow max(sp, ep)$
5: **else**
6: $ip \leftarrow \bot$

Optimization instructions

- `PartialCommit` l

1: $S, (ip_0, sp_0)_{bt} \leftarrow \textsc{PopAndProp}(S)$
2: $S \leftarrow \langle (ip_0, sp)_{bt}, \langle\rangle \rangle :: S$
3: $ip \leftarrow l$

- `BackCommit` l

1: $S, (ip_0, sp_0)_{bt} \leftarrow \textsc{PopAndProp}(S)$
2: $sp \leftarrow sp_0$
3: $ip \leftarrow l$

- `FailTwice`

1: $S, _ \leftarrow \textsc{Pop}(S)$
2: $ip \leftarrow \bot$

- `Span` X

1: **if** $I[sp] \in X$ **then**
2: $sp \leftarrow sp + 1$
3: $ep \leftarrow max(sp, ep)$
4: **else**
5: $ip \leftarrow ip + 1$

Head-fail optimization instructions

- `TestChar` $b\ l$

1: **if** $I[sp] = b$ **then**
2: $S \leftarrow (l, sp) :: S$
3: $sp \leftarrow sp + 1$
4: $ep \leftarrow max(sp, ep)$
5: $ip \leftarrow ip + 1$
6: **else**
7: $ip \leftarrow l$

- `TestSet` $X\ l$

1: **if** $I[sp] \in X$ **then**
2: $S \leftarrow (l, sp) :: S$
3: $sp \leftarrow sp + 1$
4: $ep \leftarrow max(sp, ep)$
5: $ip \leftarrow ip + 1$
6: **else**
7: $ip \leftarrow l$

- `TestAny` $n\ l$

1: **if** $sp + n \leq |I|$ **then**
2: $S \leftarrow (l, sp) :: S$
3: $sp \leftarrow sp + 1$
4: $ep \leftarrow max(sp, ep)$
5: $ip \leftarrow ip + 1$
6: **else**
7: $ip \leftarrow l$

- `TestCharNoChoice` $b\ l$

1: **if** $I[sp] = b$ **then**
2: $sp \leftarrow sp + 1$
3: $ep \leftarrow max(sp, ep)$
4: $ip \leftarrow ip + 1$
5: **else**
6: $ip \leftarrow l$

- `TestSetNoChoice` $X\ l$

1: **if** $I[sp] \in X$ **then**
2: $sp \leftarrow sp + 1$
3: $ep \leftarrow max(sp, ep)$
4: $ip \leftarrow ip + 1$
5: **else**
6: $ip \leftarrow l$

Extra instructions

- `Error` $M\ l$

1: RECORDERROR(M)
2: $ip \leftarrow l$

Basic capture instructions

- `CaptureBegin` *id*

1: $S \leftarrow (id, sp)_{cap} :: S$

- `CaptureEnd`

1: $S, e \leftarrow \text{POP}(S)$
2: $(id, sp_c)_{cap} \leftarrow e$
3: $c \leftarrow (id, content(sp_c, sp), e^{caps})$
4: **if** $|S| \neq 0$ **then**
5: $\quad S_1^{caps} \leftarrow S_1^{caps} :: c$
6: **else**
7: $\quad C \leftarrow C :: c$

Capture optimization instructions

- `CaptureLate` *n id*

1: $S \leftarrow (id, sp - n)_{cap} :: S$

- `CaptureFull` *n id*

1: $c \leftarrow (id, content(sp - n, sp), e^{caps})$
2: **if** $|S| \neq 0$ **then**
3: $\quad S_1^{caps} \leftarrow S_1^{caps} :: c$
4: **else**
5: $\quad C \leftarrow C :: c$

Memoization instructions

- `MemoOpen` *l id*

1: **if** $e \leftarrow M[(id, sp)]$ **then**
2: $\quad (id, sp_1, len, exam, caps) \leftarrow e$
3: $\quad$ **if** $len \neq \bot$ **then**
4: $\qquad sp \leftarrow sp + len$
5: $\qquad ep \leftarrow max(sp, ep)$
6: $\qquad S_1^{caps} \leftarrow S_1^{caps} :: caps$
7: $\qquad ip \leftarrow l$
8: $\quad$ **else**
9: $\qquad ip \leftarrow \bot$
10: **else**
11: $\quad S \leftarrow (id, sp)_{memo} :: S$

- `MemoClose`

1: $S, e \leftarrow \text{POPANDPROP}(S)$
2: $(id, sp_1) \leftarrow e$
3: $m \leftarrow (id, sp_1, sp - sp_1, ep - sp_1, e^{caps})$
4: $M \leftarrow M[(id, sp_1) \mapsto m]$

Tree memoization instructions

- `MemoTreeOpen` l id

1: **if** $e \leftarrow M[(id, sp)]$ **then**
2: $\quad (id, sp_1, len, exam, caps, count) \leftarrow e$
3: $\quad$ **if** $len \neq \bot$ **then**
4: $\quad\quad sp \leftarrow sp + len$
5: $\quad\quad ep \leftarrow max(sp, ep)$
6: $\quad\quad S \leftarrow (id, sp, count)_{memo} :: S$
7: $\quad\quad S_1^{caps} \leftarrow S_1^{caps} :: caps$
8: $\quad\quad ip \leftarrow l$
9: $\quad$ **else**
10: $\quad\quad ip \leftarrow \bot$
11: **else**
12: $\quad S \leftarrow (id, sp)_{memo} :: S$

- `MemoTreeInsert`

1: $S, e \leftarrow \textsc{PopAndProp}(S)$
2: $(id, pos, count)_{memo} \leftarrow e$
3: $S \leftarrow (id, pos, count + 1)_{memo} :: S$
4: $m \leftarrow (id, pos, sp - pos, ep - pos, e^{caps}, count + 1)$
5: $M \leftarrow M[(id, pos) \mapsto m]$

- `MemoTree`

1: $e_1 \leftarrow S_1$
2: $e_2 \leftarrow S_2$
3: $(id_1, pos_1, count_1)_{memo} \leftarrow e_1$
4: $(id_2, pos_2, count_2)_{memo} \leftarrow e_2$
5: **if** $id_1 = id_2 \wedge count_1 = count_2$ **then**
6: $\quad S, _ \leftarrow \textsc{Pop}(S)$
7: $\quad c \leftarrow (\textbf{dummy}, \emptyset, e_1^{caps})$
8: $\quad e_2^{caps} \leftarrow e_2^{caps} :: c$
9: $\quad S, _ \leftarrow \textsc{PopAndProp}(S)$
10: $\quad m \leftarrow (id_2, pos_2, sp - pos_2, ep - pos_2, \langle\rangle, 2 \cdot count)$
11: $\quad M \leftarrow M[(id_2, pos_2) \mapsto m]$
12: **else**
13: $\quad ip = ip + 1$

- `MemoTreeClose` id

1: $(id_1, pos, count)_{memo} \leftarrow S_1$
2: **if** $id = id_1$ **then**
3: $\quad S, _ \leftarrow \textsc{PopAndProp}(S)$
4: **else**
5: $\quad ip \leftarrow ip + 1$

A.2 Encoding

Instructions in the GPeg virtual machine are variable-length and encoded with an 8-bit opcode followed by arguments. All instructions are 2-byte aligned, with the smallest instruction being 2 bytes, and the largest being 6 bytes. This means that in certain cases, padding bytes must be inserted to enforce alignment.

Labels are encoded as 24-bit absolute offsets. This means that a program cannot exceed 16MiB in size. In practice, this is not a problem: the size of the entire Java 1.7 grammar when compiled is roughly 9-90KiB depending on the aggressiveness of inlining.

Character sets are encoded as 256-bit integers. Bit i is set to 1 if and only if the byte with value i is included in the set. Bitwise operations can be easily performed to determine if a byte is in the set or not. Character sets are not directly encoded into the bytecode. A table of character sets is stored at the start of the bytecode, and within the bytecode character sets are referred to by an 8-bit index. This allows reuse of the same charset if it is used multiple times throughout the grammar (very common, especially with a compiler that performs inlining). This also means that there cannot be more than 256 distinct character sets in the program.

The encoding of every instruction is given below:

Basic instructions: `Return`, `Fail`, `FailTwice`, `End`. These instructions take no arguments, and as such only the opcode matters. To satisfy 2-byte alignment, a padding byte is inserted.

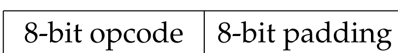


FIGURE A.2: Basic instruction encoding

Control-flow instructions: `Choice`, `Call`, `Commit`, `PartialCommit`, `BackCommit`. These instructions take one label as input, and are encoded as 4-byte instructions:

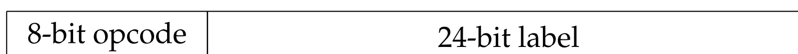


FIGURE A.3: Control flow instruction encoding

Match instructions: `Char`, `Set`, `Any`, `Span`. These instructions match characters according to various rules. All rules are encoded as 8-bit values. For sets, an 8-bit index is used for indexing into the set table, and for `Char` and `Any` the value is directly encoded as an 8-bit value.

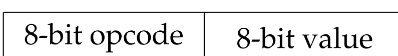


FIGURE A.4: Match instruction encoding

Test instructions: `TestChar`, `TestCharNoChoice`, `TestSet`, `TestSetNoChoice`, `TestAny`.

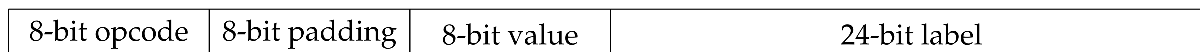


FIGURE A.5: Test instruction encoding

Memoization: `MemoOpen`, `MemoClose`, `MemoTreeOpen`, `MemoTreeInsert`, `MemoTree`, `MemoTreeClose`.

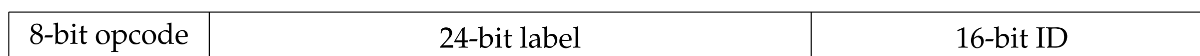


FIGURE A.6: `MemoOpen`, `MemoTreeOpen` encoding.

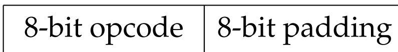


FIGURE A.7: `MemoClose`, `MemoTree`, `MemoTreeInsert` encoding.

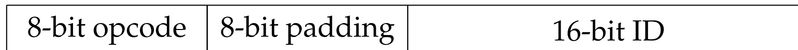


FIGURE A.8: `MemoTreeClose` instruction encoding

Basic capture instructions: `CaptureBegin`, `CaptureEnd`

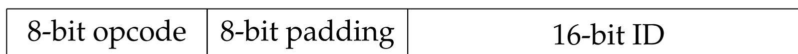


FIGURE A.9: `CaptureBegin` instruction encoding

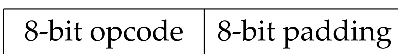


FIGURE A.10: `CaptureEnd` instruction encoding

Additional capture instructions: `CaptureLate`, `CaptureFull`

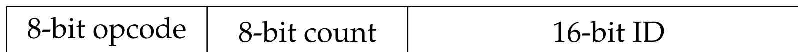


FIGURE A.11: Encoding for `CaptureLate` and `CaptureFull`.

Additional instructions: `Error`.

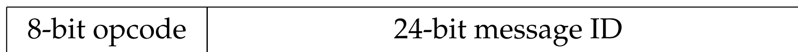


FIGURE A.12: Encoding for `Error`. Messages are stored in a separate structure similar to character sets.

Appendix B

The GPeg Library

The source code for GPeg can be found on GitHub at https://github.com/zyedidia/gpeg. At the time of writing it is roughly 4,500 lines of code implemented in a number of subpackages: `pattern`, `input`, `vm`, `memo`, `isa`, `charset`, `re`, `viz`. The `flare` package contains a syntax highlighting library which currently only supports Java. In the future we expect to move this library to its own repository.

Documentation for the project can be found at https://pkg.go.dev/github.com/zyedidia/gpeg, including documentation for each package.

For an example of usage in a separate project, see https://github.com/zyedidia/sregx, which is an implementation of structural regular expressions and uses GPeg as a recognizer for the textual structural regular expression language.