Fast Incremental PEG Parsing

Zachary Yedidia and Stephen Chong*

*Harvard University

What is Incremental Parsing?

Goal: After parsing a document, when a change/edit occurs we would like to reparse much faster than the initial parse.

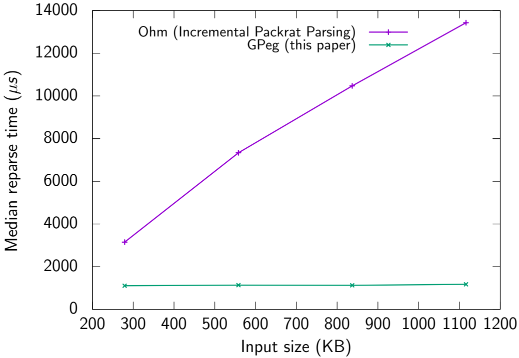
Insight: for most edits, only a localized region of the parse result is changed — other parse results can be reused.

High-level strategy:

- Save intermediate parse results.
- Determine which parse results are invalidated by an edit.
- Use any remaining parse results to reparse quickly.

Overview

We describe new methods for incremental parsing of Parsing Expression Grammars (PEGs) that enable **logarithmic** rather than **linear**-time reparses in the common case.



Overview

We describe new methods for incremental parsing of Parsing Expression Grammars (PEGs) that enable **logarithmic** rather than **linear**-time reparses in the common case.

Contributions:

- Three major improvements to the Incremental Packrat Parsing algorithm (Dubroy and Warth, SLE '17).
- GPeg: a complete implementation.[1]
- Flare: a syntax highlighting library.[2]
- Example text editor used for evaluation.
    - Integration with the Micro editor planned for the long-term.

[1] https://github.com/zyedidia/gpeg

[2] https://github.com/zyedidia/flare

Parsing Expression Grammars (PEGs)

PEGs are an alternative to Context-Free Grammars that have a few key advantages:

- No ambiguity (easier to store intermediate results).
- No lexing/parsing split (easier to define parsers).
- Possible to implement using a *parsing machine*[3] (languages can be dynamically defined).
- Can parse a similar class of languages to CFGs.

These qualities make PEGs good for defining grammars useful in text editors.

Incremental parsing allows these advantages in IDEs (and elsewhere).

[3] See LPeg (described by Ierusalimschy in SP&E '09) for an example.

Parsing Expression Grammars (cont.)

Similar to Context-Free Grammars, with two key differences:

1. The choice operation (`p1 / p2`) is not ambiguous.
2. Predicates (`&p` and `!p`) allow unlimited lookahead.

Arithmetic expressions example:

```
Top    <- Expr !.
Expr   <- Term ([-+] Term)*
Term   <- Factor ([*/] Factor)*
Factor <- Num / '(' Expr ')'
Num    <- [0-9]+
```

Parsing Expression Grammars (cont.)

Similar to Context-Free Grammars, with two key differences:

1. The choice operation (`p1 / p2`) is not ambiguous.
2. Predicates (`&p` and `!p`) allow unlimited lookahead.

Arithmetic expressions example:

```
Top    <- Expr !.
Expr   <- Term ([-+] Term)*
Term   <- Factor ([*/] Factor)*
Factor <- Num / '(' Expr ')'
Num    <- [0-9]+
```

Consequence of non-ambiguous choice

Left recursion is disallowed: `a <- a / b` loops forever.

Parsing Machine Approach (brief)

Desire: Languages should be dynamically defined.

Solution: Use a *parsing machine*[4].

Compile patterns into small programs – execute the program using an interpreter that implements the parsing machine instruction set.

[4] See LPeg (Ierusalimschy, SP&E '08) for an existing PEG parsing machine.

Parsing Machine Approach (brief)

Desire: Languages should be dynamically defined.

Solution: Use a *parsing machine*[4].

Compile patterns into small programs – execute the program using an interpreter that implements the parsing machine instruction set.

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

→

```
    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
```

[4] See LPeg (Ierusalimschy, SP&E '08) for an existing PEG parsing machine.

Incremental Parsing

Requirement: Updates after each edit must be fast.

Incremental Parsing

Requirement: Updates after each edit must be fast.

Solution: Incremental Packrat Parsing (Dubroy and Warth, SLE '17)

An adaptation of packrat parsing to an incremental setting.

Unfortunately, reparse time with Incremental Packrat Parsing is only a constant factor better than a full parse (reparsing is still linear time).

Incremental Parsing

Requirement: Updates after each edit must be fast.

Solution: Incremental Packrat Parsing (Dubroy and Warth, SLE '17)

An adaptation of packrat parsing to an incremental setting.

Unfortunately, reparse time with Incremental Packrat Parsing is only a constant factor better than a full parse (reparsing is still linear time).

Our contribution

Rethink the fundamental data structures used in Incremental Packrat Parsing.

Result: logarithmic reparse time for typical edits.

Packrat Parsing

Key idea: after attempting to parse *non-terminal* at *pos*, memoize (save) the result into a table.

If we attempt to parse *non-terminal* at *pos* (e.g., during a reparse) and it is in the table, skip the parse and use the saved result.

Packrat Parsing

Key idea: after attempting to parse *non-terminal* at *pos*, memoize (save) the result into a table.

If we attempt to parse *non-terminal* at *pos* (e.g., during a reparse) and it is in the table, skip the parse and use the saved result.

The *memoization table*[5] maps $(\textit{non-terminal}, \textit{pos}) \mapsto E$.

E is a structure that stores:

- The length of the match, or $\perp$ if the match failed.
- A possible result from the match (e.g., a parse tree).

[5] usually implemented as an array or hashtable

Packrat Parsing

Key idea: after attempting to parse *non-terminal* at *pos*, memoize (save) the result into a table.

If we attempt to parse *non-terminal* at *pos* (e.g., during a reparse) and it is in the table, skip the parse and use the saved result.

The *memoization table*[5] maps $(\textit{non-terminal}, \textit{pos}) \mapsto E$.

E is a structure that stores:

- The length of the match, or $\bot$ if the match failed.
- A possible result from the match (e.g., a parse tree).
- The number of characters **examined** to make the match (needed for incremental).

[5] usually implemented as an array or hashtable

Full Parse Example

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

2 + (3 4 * 8) / 3 0 0

Full Parse Example

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

2 + (3 4 * 8) / 3 0 0

Blue: characters in the match.
Red: additional characters examined.
Magenta: match failed.

Full Parse Example

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

2 + (3 4 * 8) / 3 0 0

Blue: characters in the match.
Red: additional characters examined.
Magenta: match failed.

Full Parse Example

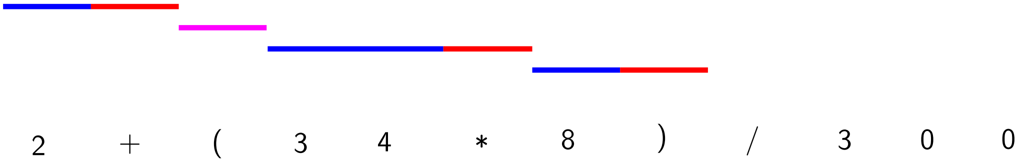
```
Expr   <- Term ([-+] Term)*
Term   <- Factor ([*/] Factor)*
Factor <- Num / '(' Expr ')'
Num    <- {{ [0-9]+ }}
```

2 + (3 4 * 8) / 3 0 0

Blue: characters in the match.
Red: additional characters examined.
Magenta: match failed.

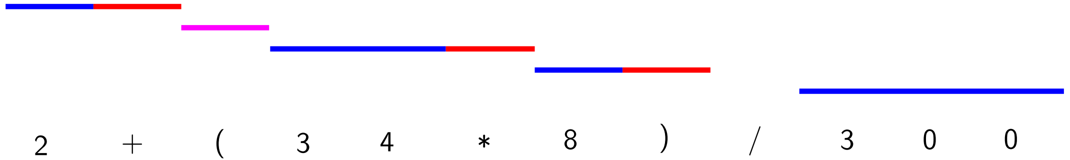
Full Parse Example

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



Blue: characters in the match.
Red: additional characters examined.
Magenta: match failed.

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



Blue: characters in the match.
Red: additional characters examined.
Magenta: match failed.

Incremental Packrat Parsing (Dubroy and Warth)

An edit $([e_{start}, e_{end}), e_{text})$ removes the interval $[e_{start}, e_{end})$ in the document and inserts e_{text} at e_{start}.

How to handle an edit?

Incremental Packrat Parsing (Dubroy and Warth)

An edit $([e_{start}, e_{end}), e_{text})$ removes the interval $[e_{start}, e_{end})$ in the document and inserts e_{text} at e_{start}.

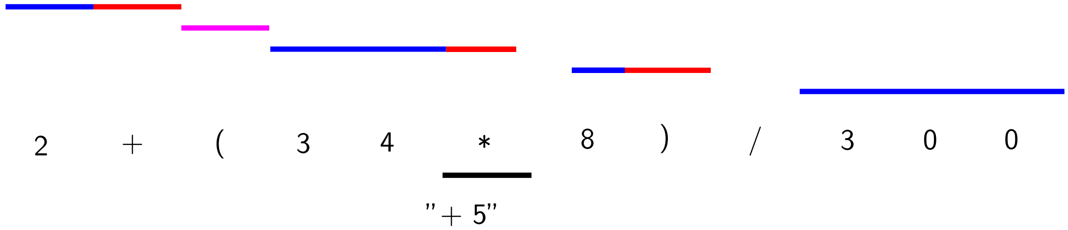
How to handle an edit?

When an edit occurs, we perform three *steps*.

1. Determine all memoization entries that are invalidated by the edit, and evict them from the memoization table.
2. Shift the start position of all memoization entries that start after the edit by the edit size $(e_{end} - e_{start} + \textsc{Len}(e_{text}))$.
3. Reparse the document from the start using the modified memoization table.

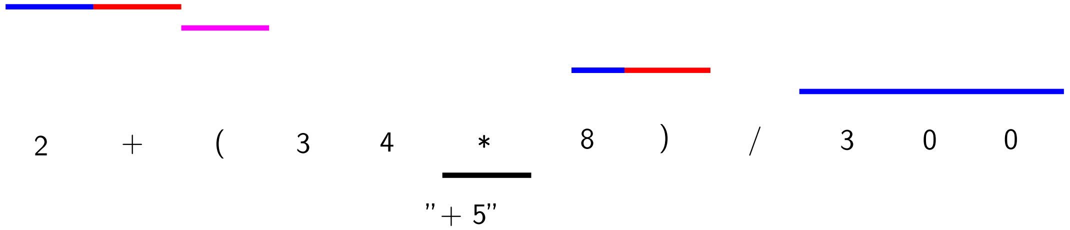
Reparse Example

Edit occurs: Remove the "*" and replace with "+5".



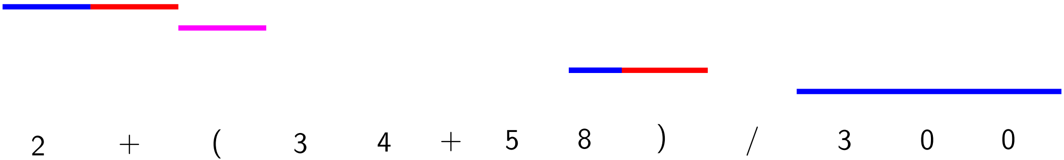
Reparse Example

Step 1: evict entries that overlap with the edit.



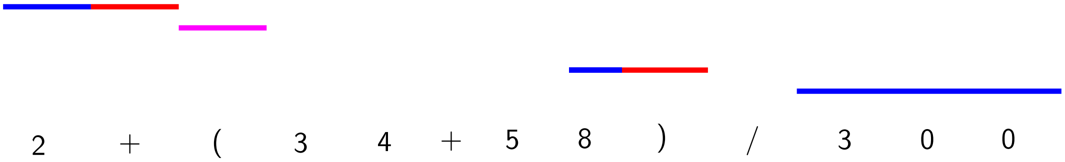
Reparse Example

Step 2: shift memoization entries over.



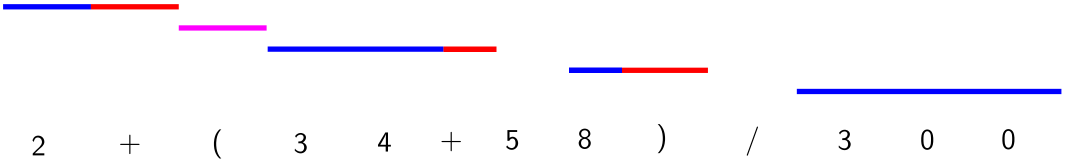
Reparse Example

Step 3: reparse from scratch.



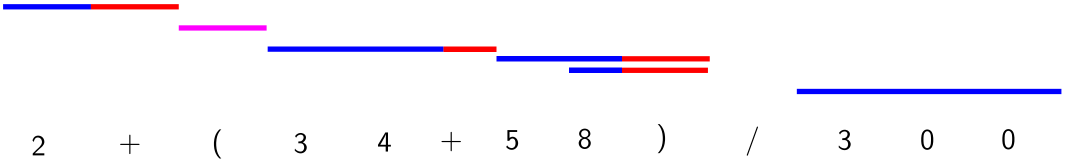
Reparse Example

Step 3: reparse from scratch.



Reparse Example

Step 3: reparse from scratch.



Incremental Packrat Parsing Summary

1. Determine all memoization entries that are invalidated by the edit, and evict them from the memoization table.
2. Shift the start position of all memoization entries that start after the edit by the edit size $(e_{end} - e_{start} + \textsc{Len}(e_{text}))$.
3. Reparse the document from the start using the modified memoization table.

Incremental Packrat Parsing Summary

1. Determine all memoization entries that are invalidated by the edit, and evict them from the memoization table.

   Linear time

2. Shift the start position of all memoization entries that start after the edit by the edit size ($e_{end} - e_{start} + \textsc{Len}(e_{text})$).

   Linear time

3. Reparse the document from the start using the modified memoization table.

   Linear time

Improvement #1: Interval Tree

Store memoization entries as intervals in an *interval tree* (implemented as an augmented AVL tree in GPeg).

Operations on a tree with n intervals:

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

Improvement #1: Interval Tree

Store memoization entries as intervals in an *interval tree* (implemented as an augmented AVL tree in GPeg).

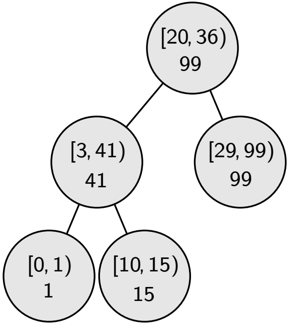
Operations on a tree with n intervals:

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

Step 1 (evict entries that overlap with the edit) is now logarithmic in the size of the memo table.

Improvement #1: Interval Tree

Store memoization entries as intervals in an *interval tree* (implemented as an augmented AVL tree in GPeg).



Improvement #2: Lazy Shifts

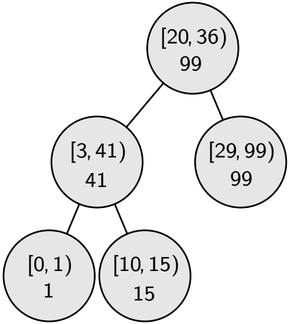
Problem: applying a shift requires iterating over every affected entry to move its start position.

Solution: apply shift requests lazily.

Improvement #2: Lazy Shifts

Problem: applying a shift requires iterating over every affected entry to move its start position.

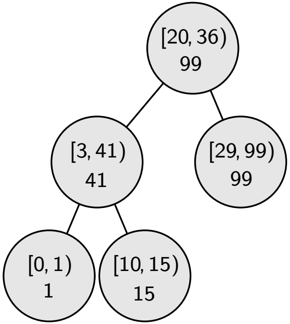
Example: interval tree with 5 intervals.



Improvement #2: Lazy Shifts

Problem: applying a shift requires iterating over every affected entry to move its start position.

Operation: Insert 4 bytes at position 5.



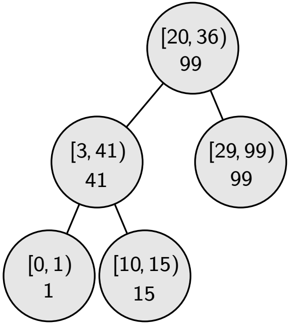
shifts:

(pos: 5, sz: 4)

Improvement #2: Lazy Shifts

Problem: applying a shift requires iterating over every affected entry to move its start position.

Operation: Insert 1 byte at position 2.

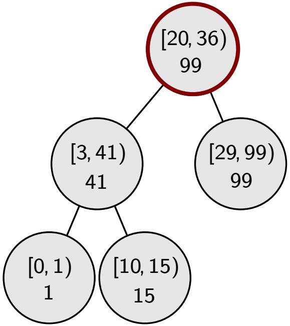


shifts:
(pos: 5, sz: 4)
(pos: 2, sz: 1)

Improvement #2: Lazy Shifts

Problem: applying a shift requires iterating over every affected entry to move its start position.

Operation: Look up interval starting at position 0.



shifts:

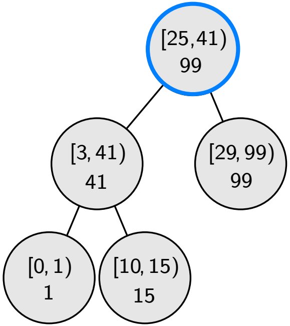
(pos: 5, sz: 4)

(pos: 2, sz: 1)

Improvement #2: Lazy Shifts

Problem: applying a shift requires iterating over every affected entry to move its start position.

Operation: Look up interval starting at position 0.

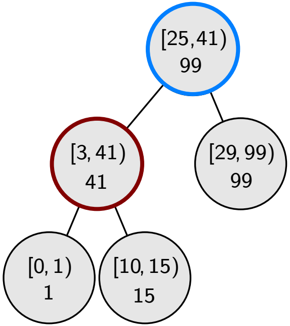


shifts:
(pos: 5, sz: 4)
(pos: 2, sz: 1)

Improvement #2: Lazy Shifts

Problem: applying a shift requires iterating over every affected entry to move its start position.

Operation: Look up interval starting at position 0.

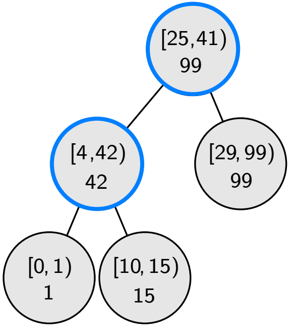


shifts:
(pos: 5, sz: 4)
(pos: 2, sz: 1)

Improvement #2: Lazy Shifts

Problem: applying a shift requires iterating over every affected entry to move its start position.

Operation: Look up interval starting at position 0.

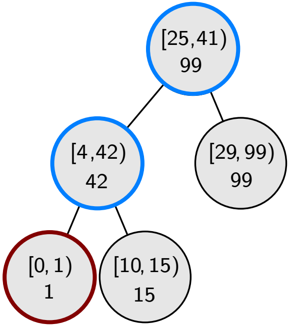


shifts:
(pos: 5, sz: 4)
(pos: 2, sz: 1)

Improvement #2: Lazy Shifts

Problem: applying a shift requires iterating over every affected entry to move its start position.

Operation: Look up interval starting at position 0.



shifts:

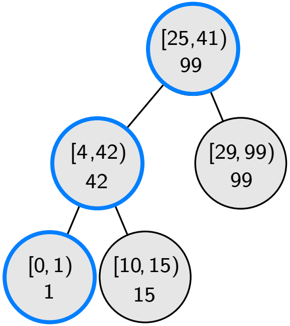
(pos: 5, sz: 4)

(pos: 2, sz: 1)

Improvement #2: Lazy Shifts

Problem: applying a shift requires iterating over every affected entry to move its start position.

Operation: Look up interval starting at position 0.



shifts:
(pos: 5, sz: 4)
(pos: 2, sz: 1)

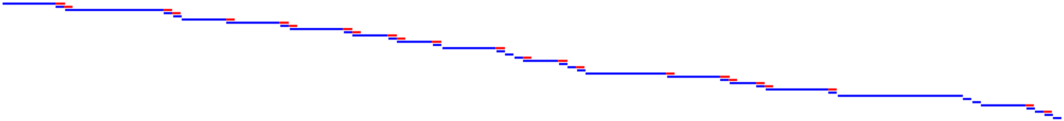
Problem #3: Linear Memoization

Problem: the pattern `p*` results in linear structures in the memoization table.

Example:

```
top   <- {{ token }}*
token <- space / keyword / string / comment / ...
...
```

Parsing using this grammar results in a memoization table with the following structure:



Each memo entry corresponds to one source token.

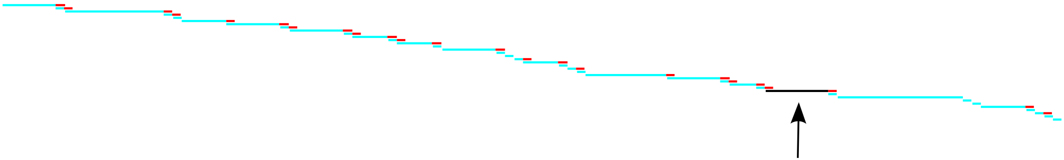
Problem #3: Linear Memoization

Problem: the pattern p* results in linear structures in the memoization table.

Example:

```
top   <- {{ token }}*
token <- space / keyword / string / comment / ...
...
```

What happens when an edit occurs?



A linear number of entries must be visited (even if just to skip).

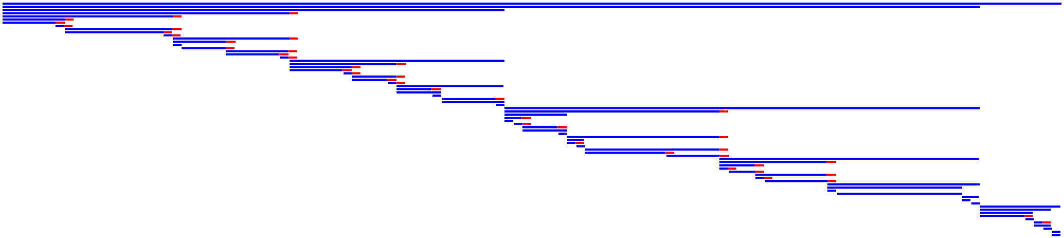
Improvement #3: Tree Memoization

Solution: enforce a new memoization strategy for `p*`. Specifically, we would like to construct "parent" entries that encompass smaller entries, and do this with a tree structure.

Improvement #3: Tree Memoization

Solution: enforce a new memoization strategy for `p*`. Specifically, we would like to construct "parent" entries that encompass smaller entries, and do this with a tree structure.

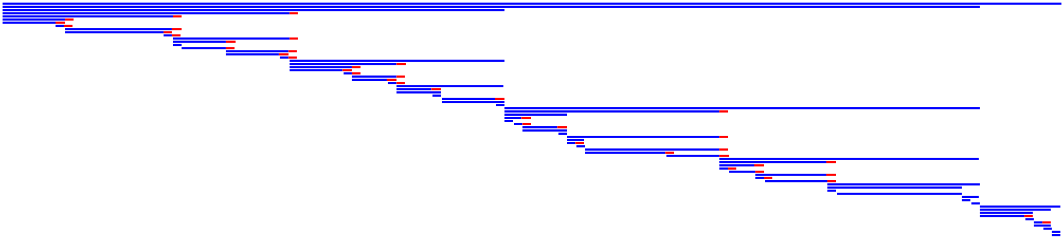
Parsing with the same grammar on the same file now produces:



Improvement #3: Tree Memoization

Solution: enforce a new memoization strategy for p*. Specifically, we would like to construct "parent" entries that encompass smaller entries, and do this with a tree structure.

Parsing with the same grammar on the same file now produces:

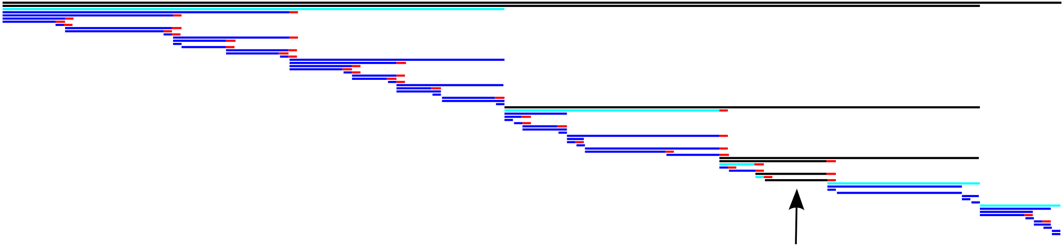


When two 1-token entries are side-by-side, the parser inserts a 2-token entry covering both. When two 2-token entries are side-by-side, a 4-token entry is inserted, etc.

Improvement #3: Tree Memoization

Solution: enforce a new memoization strategy for p*. Specifically, we would like to construct "parent" entries that encompass smaller entries, and do this with a tree structure.

What happens when an edit occurs?

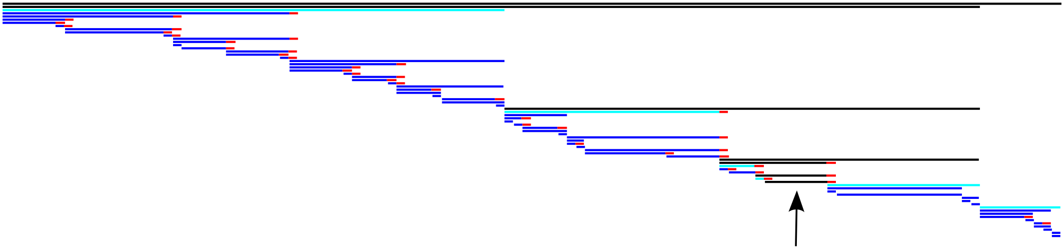


A logarithmic number of entries must be visited (shown in cyan).

Improvement #3: Tree Memoization

Solution: enforce a new memoization strategy for p∗. Specifically, we would like to construct "parent" entries that encompass smaller entries, and do this with a tree structure.

What happens when an edit occurs?



Note: there is some subtlety to ensure the tree structure is reconstructed after an edit.

Example: Token-based Syntax Highlighting

Define patterns for individual lexical elements.

```
comment       <- line_comment / block_comment
line_comment  <- '//' (!'\n' .)*
block_comment <- '/*' (!'*/' .)* '*/'?
keyword       <- "true" / "false" / "null"
```

Example: Token-based Syntax Highlighting

Define patterns for individual lexical elements.

```
comment       <- line_comment / block_comment
line_comment  <- '//' (!'\n' .)*
block_comment <- '/*' (!'*/' .)* '*/'?
keyword       <- "true" / "false" / "null"
```

Define a `token` non-terminal that attempts to match each element pattern.

```
token <- whitespace / keyword / comment / ...
```

Example: Token-based Syntax Highlighting

Define patterns for individual lexical elements.

```
comment       <- line_comment / block_comment
line_comment  <- '//' (!'\n' .)*
block_comment <- '/*' (!'*/' .)* '*/'?
keyword       <- "true" / "false" / "null"
```

Define a `token` non-terminal that attempts to match each element pattern.

```
token <- whitespace / keyword / comment / ...
```

Attempt to match `token` repeatedly, with memoization.

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

Explanation: We attempt to match `token`. If it doesn't match, we consume a character and repeatedly consume more characters while `token` still does not match. This ensures that unmatched characters are all consumed into the same memoization entry.

Example: JSON Token-based Syntax Highlighter

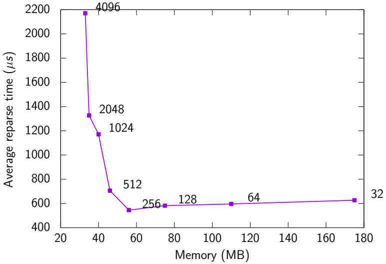
```
ws <- space+
comment <- cap{'/*' (!'*/' .)* '*/'?, "comment"}
sq_str  <- 'u'? "'" (!['\n] .)* "'"?
dq_str  <- 'U'? '"' (!["\n] .)* '"'?
string  <- cap{
    sq_str / dq_str,
    "constant.string"
}

jsonint <- [+\-]? digit+ [Ll]?
number  <- cap{(float / jsonint), "constant.number"}

keyword <- cap{
    words{"true", "false", "null"},
    "keyword"
}

operator <- cap{
    [\[\]{}:,],
    "symbol.operator"
}

token <- ws / comment / string / number / keyword / operator
```

Memoization Threshold Optimization

Do not memoize results smaller than a certain threshold (e.g., 512 bytes).

Reduces memo table size significantly.

Graph shows performance-memory tradeoff for various thresholds for a 26MB file.



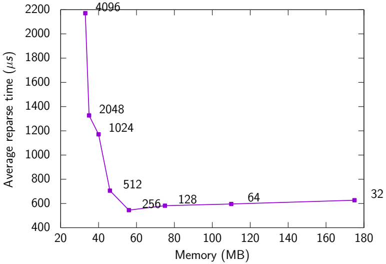
Space Optimizations

Memoization Threshold Optimization

Do not memoize results smaller than a certain threshold (e.g., 512 bytes).

Reduces memo table size significantly.

Graph shows performance-memory tradeoff for various thresholds for a 26MB file.



Capture Window Optimization

Only store parse results that exist within a requested range.

Reduces parse result size for applications that view only a particular window at a time.

Corner cases

Why do I keep saying "logarithmic for *typical* edits?"

Answer: some edits may cause unavoidable linear time reparsing by causing a global change to the parse (luckily most edits do not).

Corner cases

Why do I keep saying “logarithmic for *typical* edits?”

Answer: some edits may cause unavoidable linear time reparsing by causing a global change to the parse (luckily most edits do not).

Example: inserting `/*` at the top of a C file with no multiline comments.

```
#include <stdio.h>
// Hello world in C
int main() {
    printf("Hello world\n");
    return 0;
}
```

```
/*#include <stdio.h>
// Hello world in C
int main() {
    printf("Hello world\n");
    return 0;
}
```

Corner cases

Why do I keep saying "logarithmic for *typical* edits?"

Answer: some edits may cause unavoidable linear time reparsing by causing a global change to the parse (luckily most edits do not).

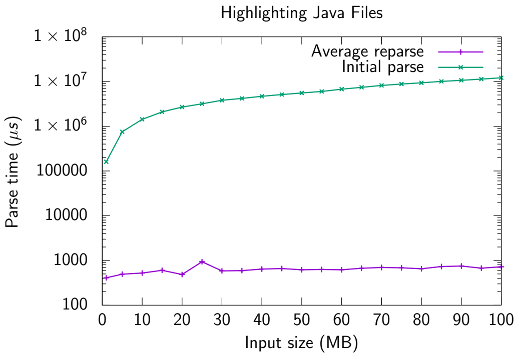
Example: inserting `/*` at the top of a C file with no multiline comments.

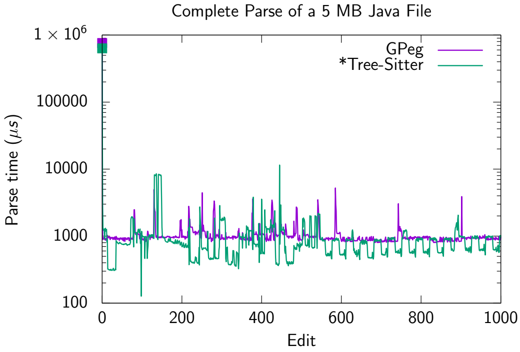
```
#include <stdio.h>
// Hello world in C
int main() {
    printf("Hello world\n");
    return 0;
}
```

```
/*#include <stdio.h>
// Hello world in C
int main() {
    printf("Hello world\n");
    return 0;
}
```

Note: since the memo table still remembers the old information, removing the `/*` will not cause a linear reparse.

Evaluation: Asymptotic Validation

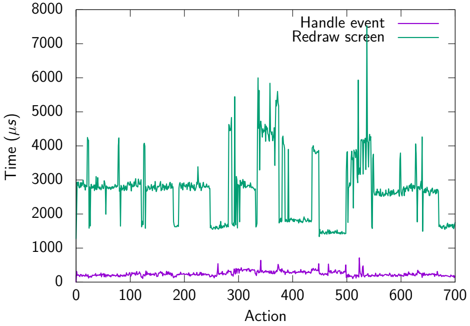


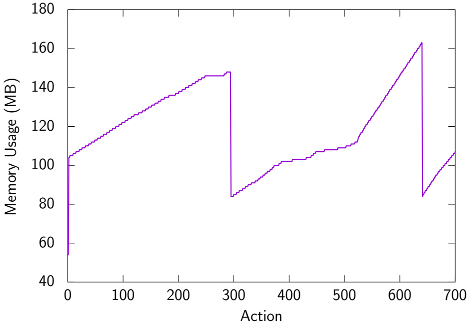


*Tree-Sitter is a well-known CFG incremental parser generator: https://tree-sitter.github.io.

Evaluation: Example Text Editor

Editing a 51 MB Java file with token-based syntax highlighting.





Conclusion

Summary: we improve Incremental Packrat Parsing by using an interval tree with lazy shifts for the memo table, and enforce tree memoization to handle linear repetition.

The implementation is available online:

- GPeg: https://github.com/zyedidia/gpeg
- Flare: https://github.com/zyedidia/flare

Thank you to my advisor Prof. Stephen Chong!

Thank you for listening!

If you have questions, please open an issue on GitHub or email me at zyedidia@stanford.edu.

Context-Sensitive Incremental Parsing: Back-references

Goal: support matching based on previously captured text. Examples: Ruby Heredocs, Lua multiline strings.

```
longstring <- '[' ref{"="*, "eq"} '[' (!(']' back{"eq"} ']') .)* (']' back{"eq"} ']')?
```

```
[==[
inside string
]==]
outside string
```

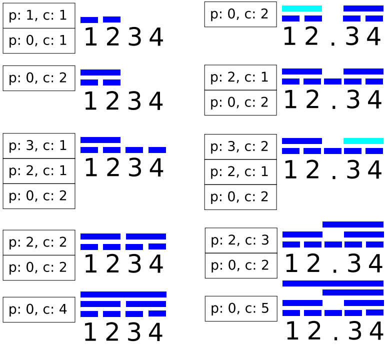
```
[===[
inside string
]==]
inside string
```

```
[===[
inside string
]===]
outside string
```

Simple solution: make sure the initial reference and any back references are in the same memoization entry.

Tree Memoization: Efficient Reconstruction

- Track repeated non-terminal counts on the machine stack.
- Consolidate stack entries by scanning down the stack.
  - Track the running sum.
  - Consolidate if the running sum is $\geq$ the next stack entry count.



Tree stays relatively balanced. However, may result in more entries than necessary in the table.

Basic Parsing Machine

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

- The instruction pointer *ip*.
- The subject pointer *sp*.
- The stack S is a list of entries:
  1. Return entries: $(ip_r)_{ret}$.
  2. Backtrack entries: $(ip_b, sp_b)_{bt}$.

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

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

Basic Parsing Machine Instructions

Instruction	Semantics
`Char` b: advances ip and consumes one byte from the subject if it matches `B` and goes to the fail state otherwise.	**if** $I[sp] = b$ **then** $ip \leftarrow ip + 1$; $sp \leftarrow sp + 1$ **else** $ip \leftarrow \bot$
`Jump` L: sets ip to L.	$ip \leftarrow L$
`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).	$S \leftarrow (L, sp)_{bt} :: S$
`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.	$S \leftarrow (ip + 1)_{ret} :: S$; $ip \leftarrow L$
`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.	$S, \_ \leftarrow \textsc{Pop}(S)$; $ip \leftarrow L$
`Return`: pops a return address from the stack and jumps to it.	$S, (ip_r)_{ret} \leftarrow \textsc{Pop}(S)$; $ip \leftarrow ip_r$
`Fail`: sets ip to the fail state: $\bot$.	$ip \leftarrow \bot$
`End`: ends matching and accepts the subject.	
`EndFail`: ends matching and fails the subject.	

Parsing Machine Compilation

Pattern	Compilation Result
'abc'	Char 'a' Char 'b' Char 'c'
.	Any 1
[a-z]	Set [a-z]
p1 p2	<p1> <p2>
p1 / p2	Choice L1 <p1> Commit L2 L1: <p2> L2: ...
p*	L1: Choice L2 <p> Commit L1 L2: ...

Pattern	Compilation Result
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 <- p	A: <p> Return

Parsing Machine Optimizations

- Special-purpose instructions: `PartialCommit`, `BackCommit`, `FailTwice`, `Span`.
- Tail-call optimization: if `Call` is followed by `Return`, can optimize to `Jump`. Turns recursion between non-terminals into flat iteration. Example: `X <- 'foo' / . X` compiles into a search loop.
- Jump replacement: if we `Jump` to another jump instruction (including `Commit`, etc.), the original jump can be directly replaced with the jump target instruction.
- Dead code elimination.
- Head-fail optimization: replace the pattern `Choice; Char` with a dedicated instruction, `TestChar`. Very important!
- Inlining (allows other optimizations to take place as well).
- Common idioms (joining alternations together, etc.). Example: `'a' / 'b' / 'c'` compiles to `Set [abc]`.