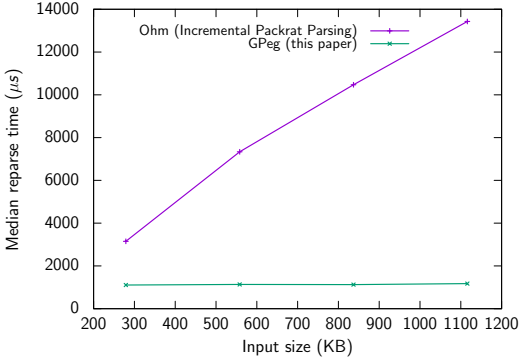
# Fast Incremental PEG Parsing

Zachary Yedidia and Stephen Chong

Harvard University

# 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 (actively developed).[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:

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

# Incremental Parsing

**Requirement**: Updates after each edit must be fast.

# Incremental Parsing

**Requirement**: Updates after each edit must be fast.

**Solution: Incremental Packrat Parsing (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 (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.

## Incremental 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.

# Incremental 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*[4] 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).

[4] usually implemented as an array or hashtable

# Incremental 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*[4] 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).
- The number of characters **examined** to make the match.

[4] 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

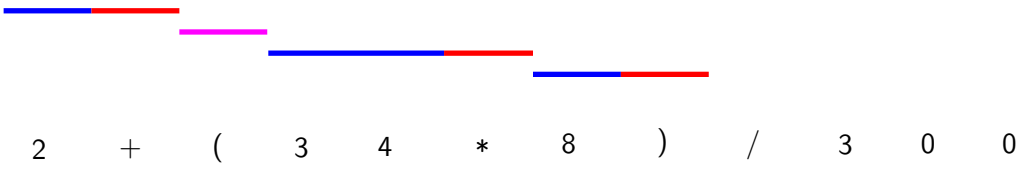
```
Expr   <- Term ([+\-] Term)*
Term   <- Factor ([*/] Factor)*
Factor <- Num / '(' Expr ')'
Num    <- {{ [0-9]+ }}
```

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

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

# Full Parse Example

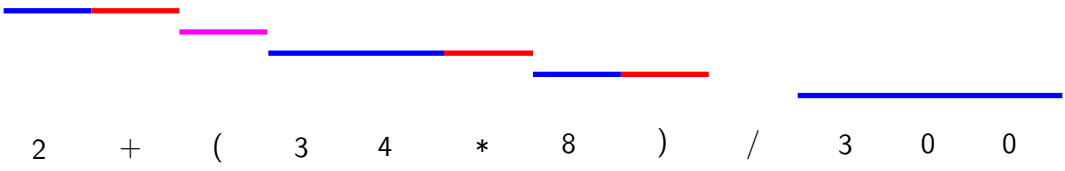
```
Expr   <- Term ([+\-] Term)*
Term   <- Factor ([*/] Factor)*
Factor <- Num / '(' Expr ')'
Num    <- {{ [0-9]+ }}
```




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

# 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.

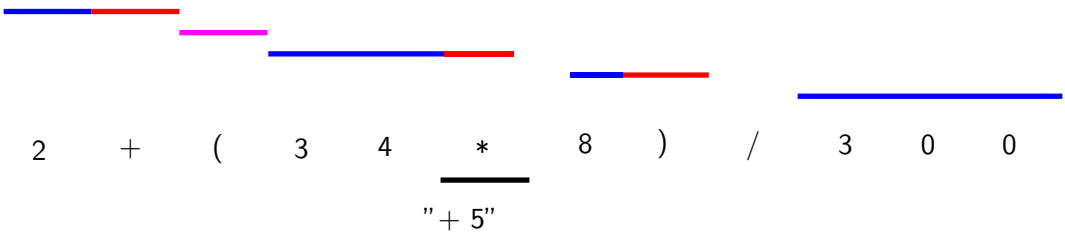
## Handling an Edit

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?

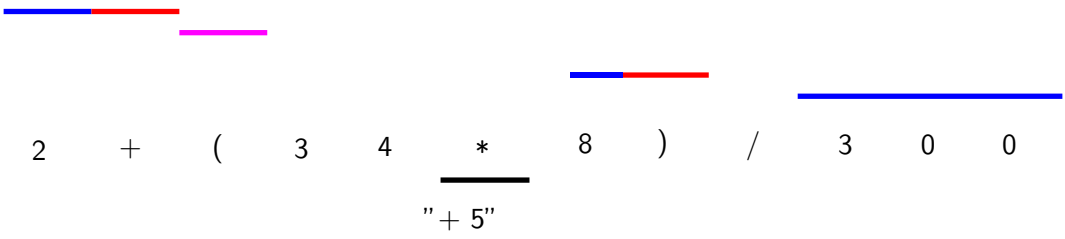
# Reparse Example

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



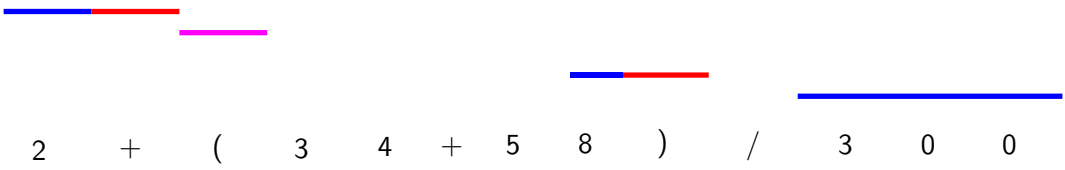
# Reparse Example

**Step 1**: evict entries that overlap with the edit.



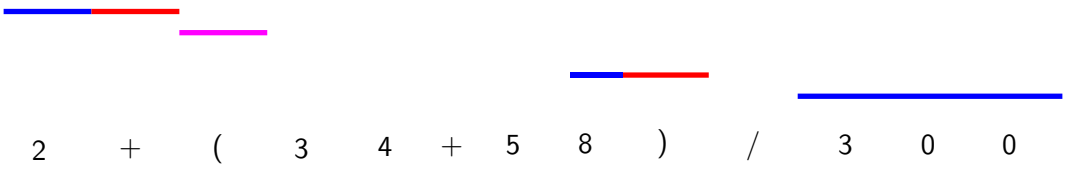
# Reparse Example

**Step 2**: shift memoization entries over.



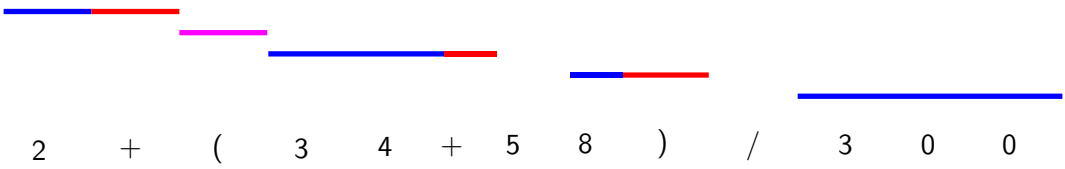
# Reparse Example

**Step 3**: reparse from scratch.



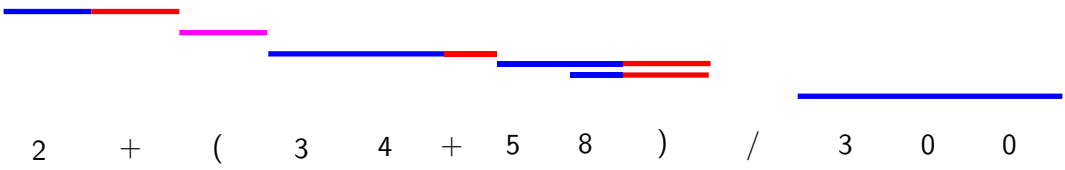
# Reparse Example

**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).

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).

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 #2: Lazy Shifts

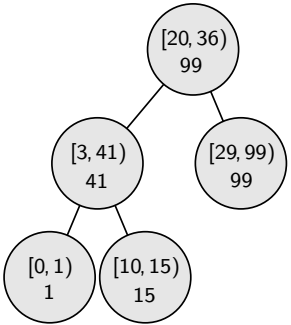
**Problem**: applying a shift requires iterating over every affected entry to move its start position.

**Solution**: apply shift requests lazily.

# Improvement #2: Lazy Shifts

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

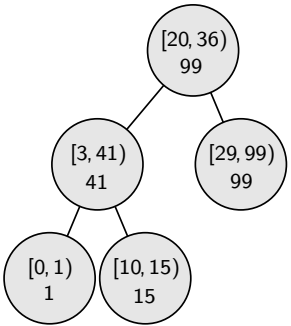
**Example**: interval tree with 5 intervals.



# 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.

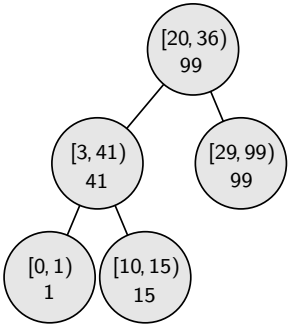



shifts:
(pos: 5, sz: 4)

# 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.

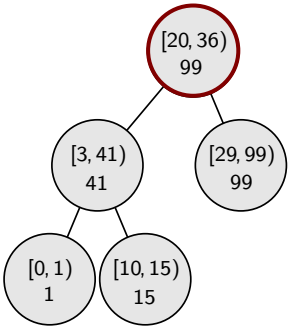



shifts:
(pos: 5, sz: 4)
(pos: 2, sz: 1)

# 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.

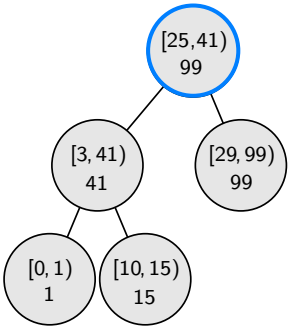



shifts:
(pos: 5, sz: 4)
(pos: 2, sz: 1)

# 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.

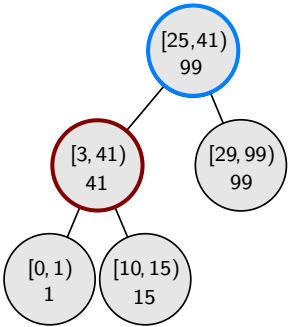



shifts:
(pos: 5, sz: 4)
(pos: 2, sz: 1)

# 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.

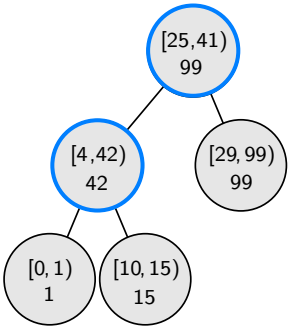



shifts:
(pos: 5, sz: 4)
(pos: 2, sz: 1)

# 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.

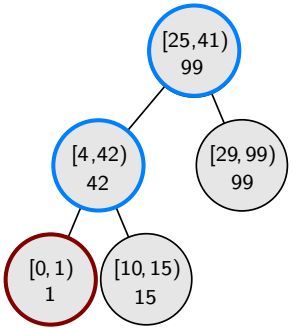



shifts:
(pos: 5, sz: 4)
(pos: 2, sz: 1)

# 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.

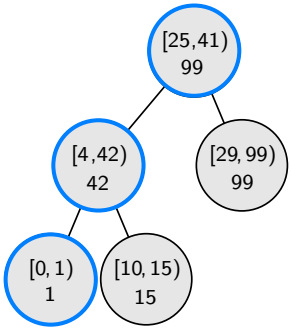



shifts:
(pos: 5, sz: 4)
(pos: 2, sz: 1)

# 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)

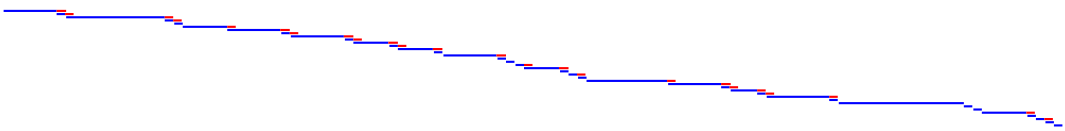
# Problem #3: Linear Memoization

**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.

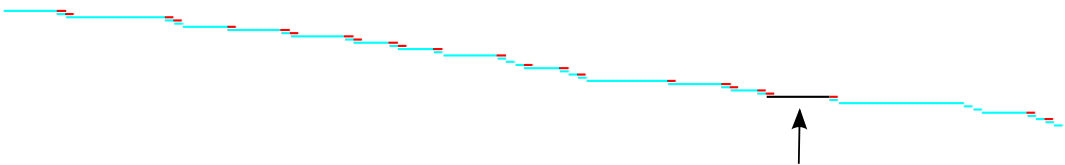
# Problem #3: Linear Memoization

**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).

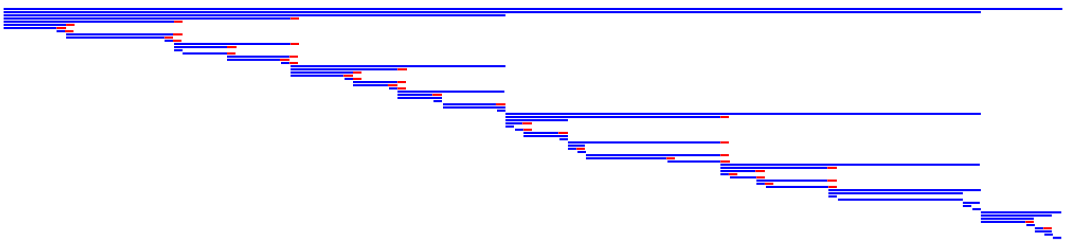
# Improvement #3: Tree Memoization

**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.

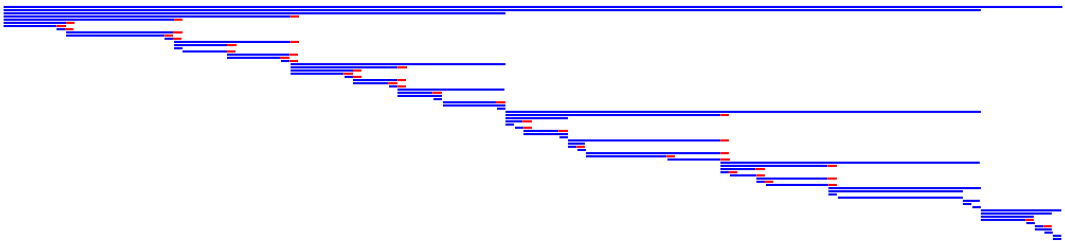
Parsing with the same grammar on the same file now produces:



# 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:

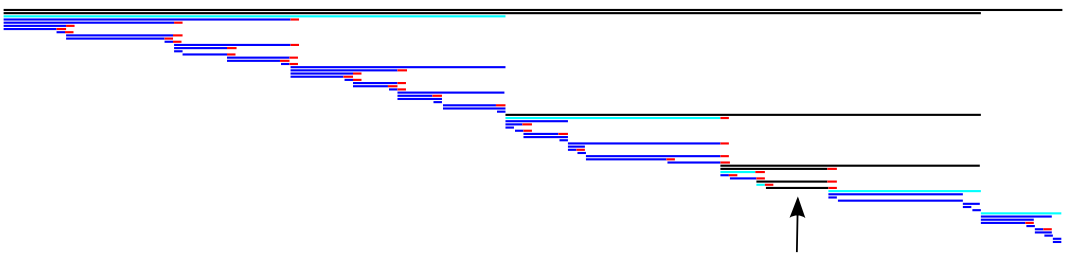


When two 1-token entries are side-by-side, the parser inserts a 2-token entry covering both. When two 2-token entries are side-by-side, a 4-token entry is inserted, etc.

# 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?

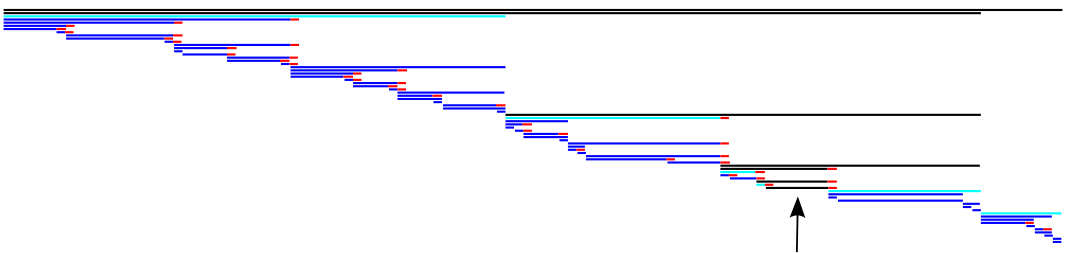


A logarithmic number of entries must be visited (shown in cyan).

# 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.

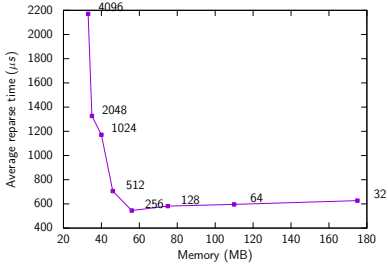
# Space Optimizations

## 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.

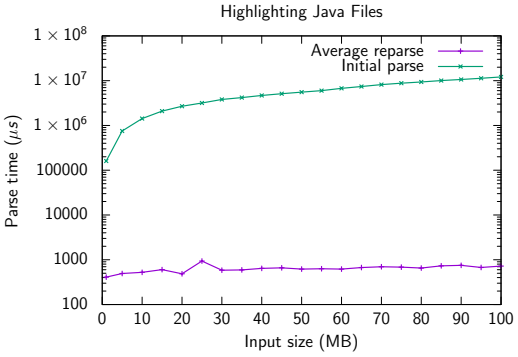


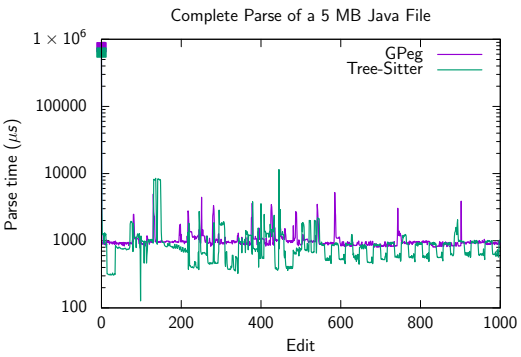

## Capture Window Optimization

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.

# Evaluation: Asymptotic Validation

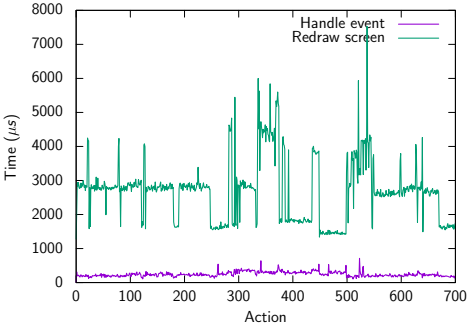


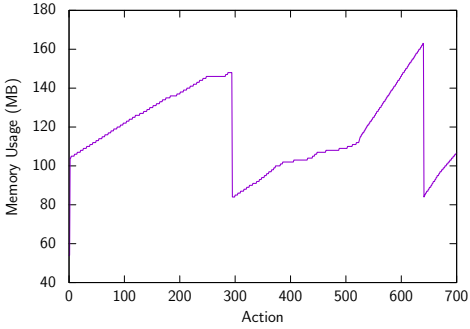




Tree-Sitter is a well-known CFG incremental parser generator: https://tree-sitter.github.io.

# 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.