Incremental Parsing of Common Lisp Code

Irène Durand & Robert Strandh

LaBRI, University of Bordeaux

April, 2018

Context

Emacs is likely the most common editor for Common Lisp code.

- The current package is not taken into account.
- The indent function can not distinguish between forms and bindings.
- No distinction between different roles of symbols.
- Incorrect indentation is not indicated.

Taking packages into account

Emacs does not take packages into account for syntax highlighting.

This code is highlighted correctly:

```
(defpackage :p (:use :common-lisp))

(in-package :p)

(defun f (x) x)
```

Taking packages into account

Emacs does not take packages into account for syntax highlighting.

This code is not highlighted correctly:

```
(defpackage :p (:use))

(in-package :p)

(defun f (x) x)
```

Distinguishing between forms and bindings

Emacs does not distinguish between forms and bindings.

This binding is indented in one way:

```
(let ((temp
       (find key *entries* :test #'eq :key #'car)))
  ...)
```

Distinguishing between forms and bindings

Emacs does not distinguish between forms and bindings.

This binding is indented in a different way:

```
(let ((prog1
          (find key *entries* :test #'eq :key #'car)))
  ...)
```

And the role of `prog1` is not taken into account.

Indicating incorrect indentation

Emacs does not indicate bad indentation.

This form contains an incorrect indentation:

```
(let* ((x (expt *result* 3))
  (declare (float x)))
  (+ x 1.0))
```

Objectives

An excellent editor for Common Lisp code:

- Take current package into account.
- Distinguish forms from other entities.
- Show incorrect indentation.
- Take roles of symbols into account.
- Provide refactoring functionality.

First step towards objectives

Create an incremental parser for Common Lisp code that yields a considerably more accurate result than existing parsers.

Recapitulation: Editor buffer protocol

Presented at ELS 2016.

Two sub-protocols:

- Edit protocol. Access, insert, or delete an item. Can be invoked a large number of times for each keystroke.
- Update protocol. Determine changes since last update. Typically invoked once for each keystroke.

For the current work, we are only interested in the update protocol.

Our technique: Parse result

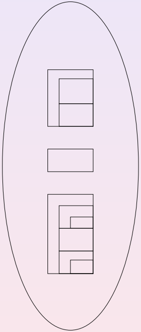
The analysis of the buffer contents returns *parse results*.

A parse result contains:

- The *start position* and *end position* (line, column) in the buffer of the parse result.
- The *type* (expression, comment, etc) of the parse result.
- A possibly empty list of *children*.

Our technique: Cache of parse results

We maintain a *cache* that maps buffer positions to parse results.



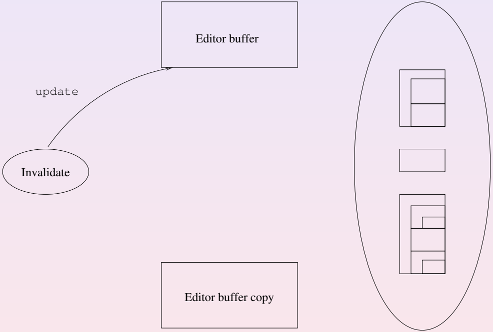
Our technique: Two phases

Our incremental parser has two phases:

- Invalidation.
- Rehabilitation.

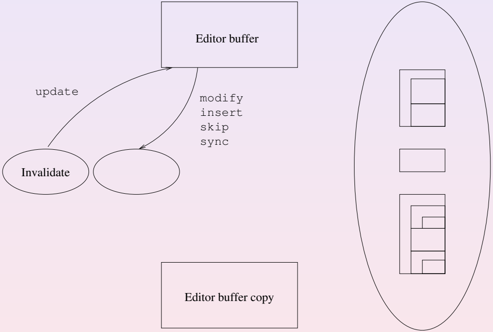
Invalidation phase

Step 1: Invoke the update protocol of the buffer.



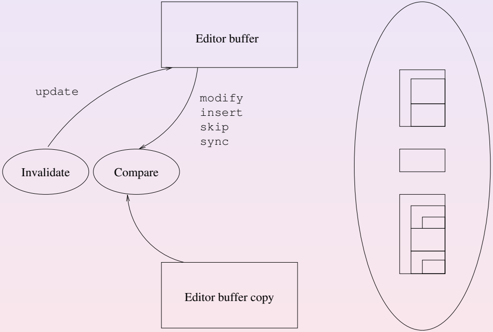
Invalidation phase

Step 2: Update protocol emits update information.



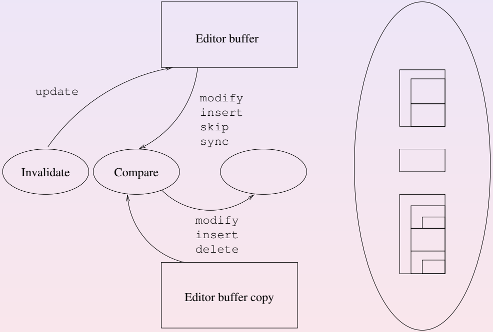
Invalidation phase

Step 3: Compare to buffer copy.



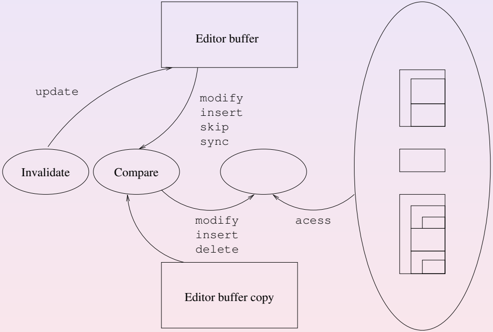
Invalidation phase

Step 4: Convert to `modify`, `insert`, `delete`.



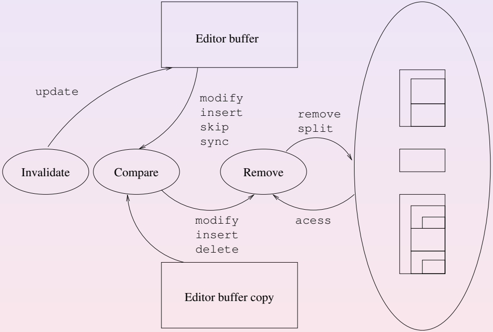
Invalidation phase

Step 5: Check whether any parse result is affected.



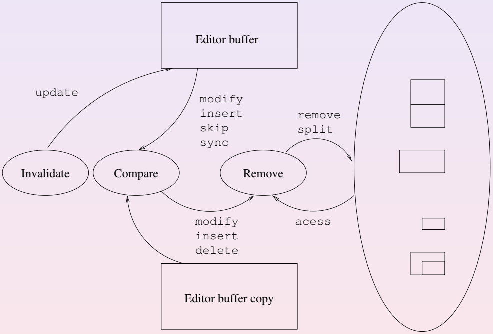
Invalidation phase

Step 6: If so, remove or split it.



Invalidation phase

Step 7: Keep parse results that are still valid.



Rehabilitation phase

We use a modified version of the standard Common Lisp function `read`:

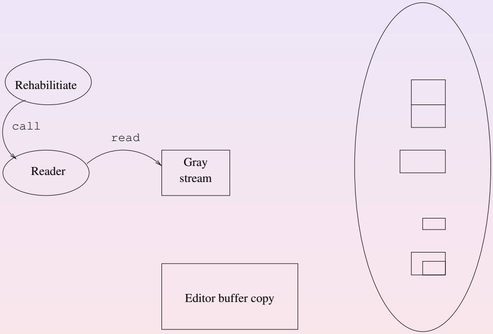
- It returns *parse results* instead of expressions.
- It also returns parse results corresponding to non-expressions.

Rehabilitation phase

The modified `read` function uses a Gray stream that accesses the contents of the text buffer.

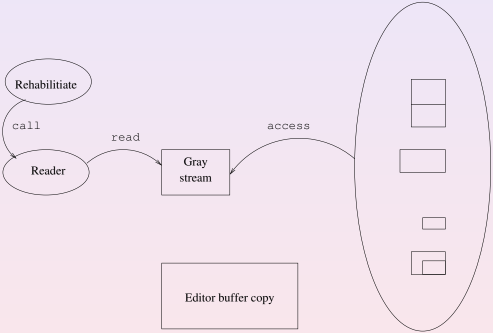
Rehabilitation phase

Step 1: Conceptually invoke `read` on entire buffer copy.



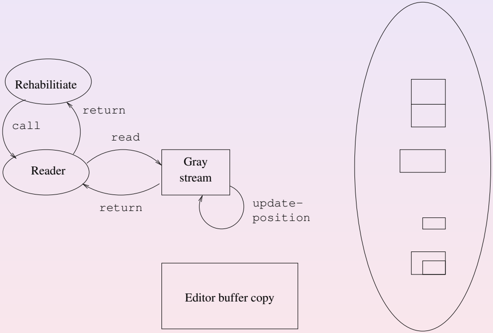
Rehabilitation phase

Step 2: Check whether a parse result exists in the cache.



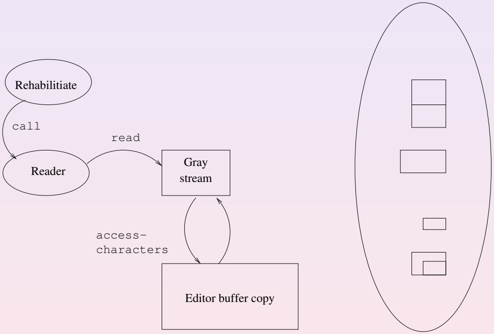
Rehabilitation phase

Step 3a: If so, update position and return from reader.



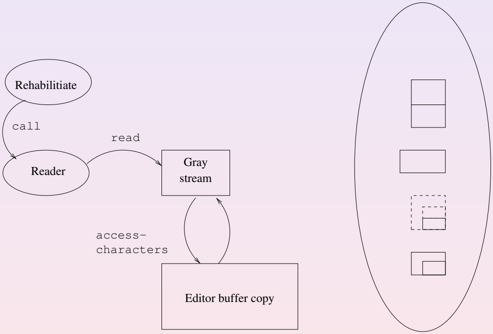
Rehabilitation phase

Step 3b: If not, access characters from buffer copy.



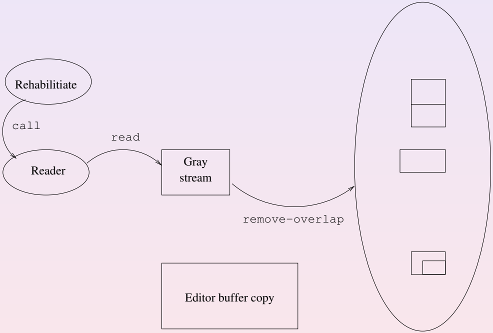
Rehabilitation phase

Step 3b: The result is a new parse result.



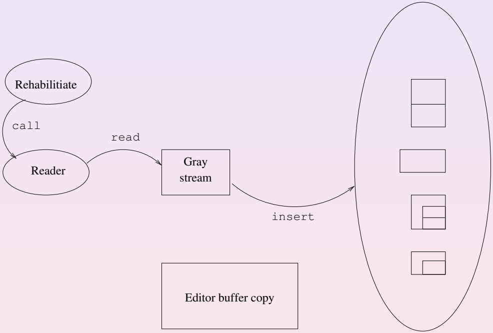
Rehabilitation phase

Step 4b: Remove overlapping parse results from cache.



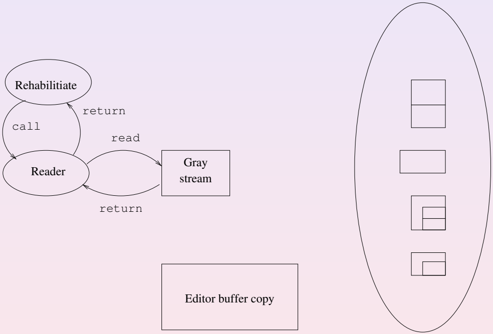
Rehabilitation phase

Step 5b: Insert new parse result into cache.



Rehabilitation phase

Step 6b: Return the new parse result.



Optimizations

- We skip a prefix of unmodified material.
- We skip a suffix of unmodified material, provided that structure is preserved.
- The cache representation is optimized for small modifications.

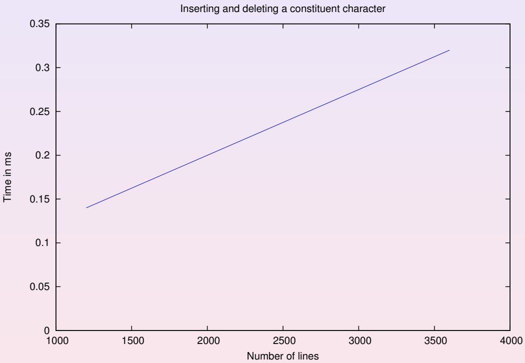
Performance

Tests run on a 4-core Intel Core processor clocked at 3.3GHz, running SBCL version 1.3.11.

Performance Inserting and deleting a constituent character

nb forms	form size	time
120	10	0.14ms
80	15	0.14ms
60	20	0.14ms
24	100	0.23ms
36	100	0.32ms

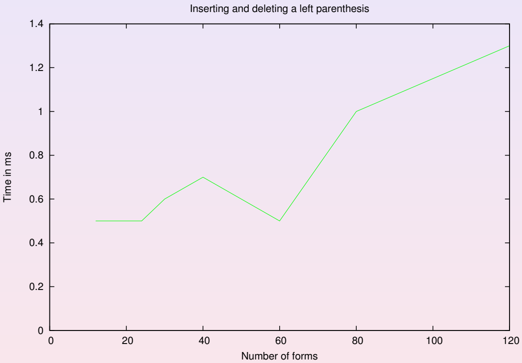
Performance Inserting and deleting a constituent character



Performance Inserting and deleting a left parenthesis

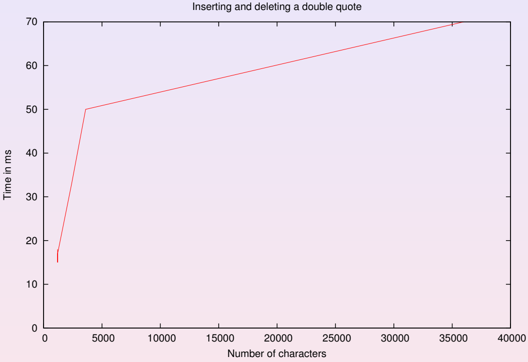
nb forms	form size	time
120	10	1.3ms
80	15	1.0ms
60	20	0.5ms
40	30	0.7ms
30	40	0.6ms
24	50	0.5ms
12	100	0.5ms

Performance Inserting and deleting a left parenthesis



Performance Inserting and deleting a double quote

nb forms	form size	characters per line	time
120	10	1	18ms
80	15	1	15ms
60	20	1	17ms
24	100	1	33ms
36	100	1	50ms
120	10	30	70ms


Inserting and deleting a double quote
Time in ms
70
60
50
40
30
20
10
0
0
5000
10000
15000
20000
25000
30000
35000
40000
Number of characters

Future work

- Use parse result to compute indentation.
- Implement incremental version of first-class global environments.
- Use new environment implementation to compile top-level forms at typing speed.
- Display information from compilation.
- Implement refactoring tools based on compilation.

Acknowledgments

We would like to thank Philipp Marek and Cyrus Harmon for providing valuable feedback on early versions of this paper.

Thank you

Questions?