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

- The current package is not taken into account.
- The indent function cannot 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)))
  ...)

5/41
Distinguishing between forms and bindings

Emacs does not distinguish between forms and bindings.

This binding is indented in a different way:

```lisp
(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.

Editor buffer

update

modify
insert
skip
sync

Invalidate

Editor buffer copy
Step 3: Compare to buffer copy.
Step 4: Convert to modify, insert, delete.
Step 5: Check whether any parse result is affected.
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.
The modified \texttt{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

<table>
<thead>
<tr>
<th>nb forms</th>
<th>form size</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>120</td>
<td>10</td>
<td>0.14ms</td>
</tr>
<tr>
<td>80</td>
<td>15</td>
<td>0.14ms</td>
</tr>
<tr>
<td>60</td>
<td>20</td>
<td>0.14ms</td>
</tr>
<tr>
<td>24</td>
<td>100</td>
<td>0.23ms</td>
</tr>
<tr>
<td>36</td>
<td>100</td>
<td>0.32ms</td>
</tr>
</tbody>
</table>
Performance: Inserting and deleting a constituent character

![Inserting and deleting a constituent character](image)
# Performance

Inserting and deleting a left parenthesis

<table>
<thead>
<tr>
<th>nb forms</th>
<th>form size</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>120</td>
<td>10</td>
<td>1.3ms</td>
</tr>
<tr>
<td>80</td>
<td>15</td>
<td>1.0ms</td>
</tr>
<tr>
<td>60</td>
<td>20</td>
<td>0.5ms</td>
</tr>
<tr>
<td>40</td>
<td>30</td>
<td>0.7ms</td>
</tr>
<tr>
<td>30</td>
<td>40</td>
<td>0.6ms</td>
</tr>
<tr>
<td>24</td>
<td>50</td>
<td>0.5ms</td>
</tr>
<tr>
<td>12</td>
<td>100</td>
<td>0.5ms</td>
</tr>
</tbody>
</table>
Performance Inserting and deleting a left parenthesis

![Graph showing the performance of inserting and deleting a left parenthesis over time and number of forms.](image-url)
### Performance: Inserting and deleting a double quote

<table>
<thead>
<tr>
<th>nb forms</th>
<th>form size</th>
<th>characters per line</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>120</td>
<td>10</td>
<td>1</td>
<td>18ms</td>
</tr>
<tr>
<td>80</td>
<td>15</td>
<td>1</td>
<td>15ms</td>
</tr>
<tr>
<td>60</td>
<td>20</td>
<td>1</td>
<td>17ms</td>
</tr>
<tr>
<td>24</td>
<td>100</td>
<td>1</td>
<td>33ms</td>
</tr>
<tr>
<td>36</td>
<td>100</td>
<td>1</td>
<td>50ms</td>
</tr>
<tr>
<td>120</td>
<td>10</td>
<td>30</td>
<td>70ms</td>
</tr>
</tbody>
</table>
Inserting and deleting a double quote
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?