Incrementalist User's Manual

Copyright © 2013 Robert Strandh Copyright © 2024 Jan Moringen

Table of Contents

1 Introduction **1**

2 Representation of the Editor Buffer **2**

3 Parsing using the Common Lisp Reader **3**

4 Incremental Parsing **4**

4.1 Data Structures 5
4.1.1 WADs 5
4.1.2 Prefix and Suffix 6
4.2 Moving Top-level Wads 8
4.3 Incremental Update 9
4.3.1 Processing Modifications 10
4.3.2 Recreating the Cache 15

Concept index **16**

Function and macro and variable and type index . . **17**

1 Introduction

Incrementalist is a system for incrementally parsing Common Lisp code that has been developed in the context of the Climacs editor for Common Lisp code and extracted into its own system.

- We implemented a better buffer representation, and extracted it from the editor code into a separate library named Cluffer (`https://github.com/robert-strandh/Cluffer`). The new buffer representation has better performance, especially on large buffers, and it makes it easier to write sophisticated parsers for buffer contents.
- The incremental parser for Common Lisp syntax of the first version of Climacs was very hard to maintain, and while it was better than that of Emacs, it was still not good enough. To improve upon those previous approaches, Incrementalist uses Eclector (`https://github.com/s-expressionists/eclector`) in order to parse buffer contents. Eclector is a library that implements the Common Lisp reader, but that can also be customized in many ways. We take advantage of these capabilities to read material that is normally skipped, like comments, and for error recovery. By using a Common Lisp reader, we parse the buffer contents in the same way that the Common Lisp compiler would.
- Incrementalist is independent of any particular library for making graphic user interfaces, allowing it to be configured with different such libraries.

2 Representation of the Editor Buffer

Incrementalist uses the Cluffer (`https://github.com/robert-strandh/Cluffer`) library to represent its buffers. We briefly describe the essential aspects of that library below. For detailed information on how it works, see the dedicated documentation.

Cluffer proposes two distinct protocols, namely the *edit protocol* and the *update protocol*.

- The *edit protocol* provides operations for editing the buffer contents. It has been designed to be both simple and very efficient. As such, it does not provide operations on larger chunks of contents such as *regions*. It provides operations only on single items, and operations to split and join lines. These editing operations do not trigger any view updates which is why they can be invoked a large number of times for each user interaction without loss of performance. This feature is taken advantage of in operations on regions and in keyboard macros.
- The *update protocol* is designed to be run at the frequency of the event loop. It is based on the concept of *time stamps*. Any number of edit operations can be performed between two invocations of the update protocol, and the update protocol can be invoked at different times for different views, including very rarely for views that are not currently on display. Given that the amount of data displayed in a view is relatively modest, no attempt is made to minimize the modifications to the view. The smallest unit of an update is a *line* of items.

3 Parsing using the Common Lisp Reader

We use a special version of the Common Lisp reader, i.e., Eclector (`https://github.com/s-expressionists/eclector`), to parse the contents of a buffer. We use a special version of the reader for the following reasons:

- We need a different action from that of the standard reader when it comes to interpreting tokens. In particular, we do not necessarily want the incremental parser to intern symbols automatically, and we do not want the reader to fail when a symbol with an explicit package prefix does not exist or when the package corresponding to the package prefix does not exists.
- We need for the reader to return not only a resulting expression, but an object that describes the start and end locations in the buffer where the expression was read.
- The reader needs to return source elements that are not returned by an ordinary reader, such as comments and expressions that are skipped by certain other reader macros.
- The reader can not fail but must instead recover in some way when either some invalid syntax is encountered, or when end of file is encountered in the middle of a call.

4 Incremental Parsing

As mentioned in Chapter 2 [Representation of the Editor Buffer], page 2, the general control structure for buffer modifications and incremental updates was designed with the following goals:

- Most editing operations should be very fast, even when they involve fairly large chunks of buffer contents. Here, *fast* means that the response time for interactive editing should be short.
- From a software-engineering point of view, the buffer editing operations should not be aware of the presence of any *views*.

Notice that it was *not* a goal that editing operations use as little computational power as possible.

Input events can be divided into two categories:

- Input events that result in some modification to some buffer contents. Inserting and deleting items are in this category. Modifications can be the result of indirect events such as executing a keyboard macro that inserts or deletes items in one or more buffers.
- Input events that have no effect on any buffer contents. Moving a cursor, changing the size of a window, or scrolling a view are typical events in this category. These events influence only the *view* into a buffer.

When an event in the first category occurs, the following chain of events is triggered:

1. The event itself triggers the execution of some *command* that causes one or more items to be inserted and/or deleted from one or more buffers. Whether this happens as a direct result or as an indirect result of the event makes no difference. The buffers involved are modified, but no other action is taken at this time. Lines that are modified or inserted are marked with the *current time stamp* and the current time stamp is incremented, possibly more than once.
2. At the end of the execution of the command, the *syntax update* is executed for all buffers, allowing the contents to be incrementally parsed according to the syntax associated with the buffer.

   > **warning:** There seem to be cases where the syntax of one buffer depends not only on its own associated buffer, but also on the contents of other buffers. It is not a big problem if the dependency is only on the *contents* of other buffers, but if the dependency is also on the *result of the syntax analysis* of other buffers, then one syntax update might invalidate another. In that case, it might be necessary to loop until all analyses are complete. This can become very complicated because there can now be circular dependencies so that the entire editor gets caught in an infinite loop.

   Finally, visible views are repainted using whatever combination they want of the buffer contents and the result of the syntax update. The syntax update uses the time stamps of lines in the buffer and of the previous syntax update to compute an up-to-date representation of the buffer. This computation is done incrementally as much as possible.
3. Each view on display recomputes the data presented to the user and redraws the associated window. Again, time stamps are used to make this computation as incremental as possible.

4.1 Data Structures

4.1.1 WADs

We call the data structure for storing the (modified) return value of the reader together with start and end location a *wad*. It contains the following slots:

- The start and the end location of the wad in the buffer. The start line number within this location information is either *absolute* which means that it directly corresponds to a line number of the underlying buffer or *relative* which means that it represents an offset from the start line number of a parent or sibling wad. A dedicated slot indicates whether the start line number is relative or absolute.
- The "raw" expression that was read, with some possible modifications. Tokens, in particular symbols, are not represented as themselves for reasons mentioned above.
- A list of *children*. These are wads that were returned by recursive calls to the reader. The children are represented in the order they were encountered in the buffer. This order may be different from the order in which the corresponding expressions appear in the expression resulting from the call to the reader. Furthermore, the descendants of a given wad can contain wads which correspond to source elements that would not be present in the s-expression tree returned by `cl:read` at all such as comments or expressions guarded by feature expressions.

The representation of a wad is shown in Figure 4.1.

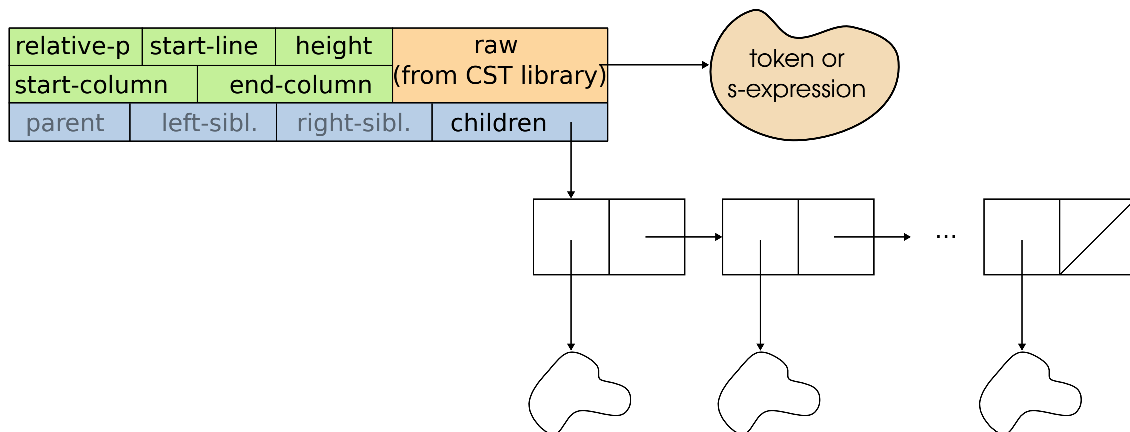

Figure 4.1: Representation of a wad. The major components are highlighted: location information in green, tree structure information in blue and "raw" information in orange. Not all wad classes contain children and "raw" data. The faintly rendered tree structure slots are computed on-demand and are thus always present but not always bound.

A location in the buffer is considered a *top-level location* if and only if, when the buffer is parsed by a number of consecutive calls to `read`, when this location is reached, the reader is in its initial state with no recursive pending invocations. Similarly A wad is considered a *top-level wad* if it is the result of an immediate call to `read`, as opposed to of a recursive call.

Let the *initial character* of some wad be the first non-whitespace character encountered during the call to the reader that produced this wad. Similarly, let the *final character* of some wad be the last character encountered during the call to the reader that produced this wad, excluding any look-ahead character that could be un-read before the wad was returned.

The value of the `start-line` slot for a wad *w* is computed as follows:

- If *w* is [term-absolute], page 5, which is the case if and only if *w* is one of the top-level wads in the prefix (see Section 4.1.2 [Prefix and Suffix], page 6) or the first top-level wad in the suffix, then the value of this slot is the absolute line number of the initial character of *w*. The first line of the buffer is numbered 0.
- Otherwise *w* is [term-relative], page 5, which is the case for different kinds of placements of *w* in the overall hierarchical structure of wads:
  - If *w* is a top-level wad in the suffix other than the first one, then the value of this slot is the number of lines between the value of the slot `start-line` of the preceding wad and the initial character of *w*. A value of 0 indicates the same line as the `start-line` of the preceding wad.
  - If *w* is the first in a list of children of some parent wad *p*, then the value of this slot is the number of lines between the start line of *p* (which is different from the value of the `start-line` slot of *p* if *p* is itself *relative*) and the initial character of *w*.
  - If *w* is the child other than the first in a list of children of some parent wad, then the value of this slot is the number of lines between the start line of the preceding sibling wad and the initial character of *w*.

The value of the slot `height` of some wad *w* is the number of lines between the start line of *w* and the final character of *w*. If *w* starts and ends on the same line, then the value of this slot is 0.

The value of the slot `start-column` is the absolute column number of the initial character of this wad. A value of 0 means the leftmost column.

The value of the slot `end-column` is the absolute column number of the final character of the wad.

4.1.2 Prefix and Suffix

Incrementalist maintains a sequence[1] of *top-level wads*. This sequence is organized as two ordinary Common Lisp lists, called the *prefix* and the *suffix*. Given a top-level location L in the buffer, the prefix contains a list of the top-level wad that precede L and the suffix contains a list of the top-level wads that follow L. The top-level wads in the prefix occur in reverse order compared to order in which they appear in the buffer. The top-level wads in the suffix occur in the same order as they appear in the buffer. the location L is typically immediately before or immediately after the top-level expression in which the *cursor* of the current view is located, but that is not a requirement. Figure 4.2 illustrates the prefix and the suffix of a buffer with five top-level expressions.

[1] It is not a Common Lisp sequence, but just a suite represented in a different way.

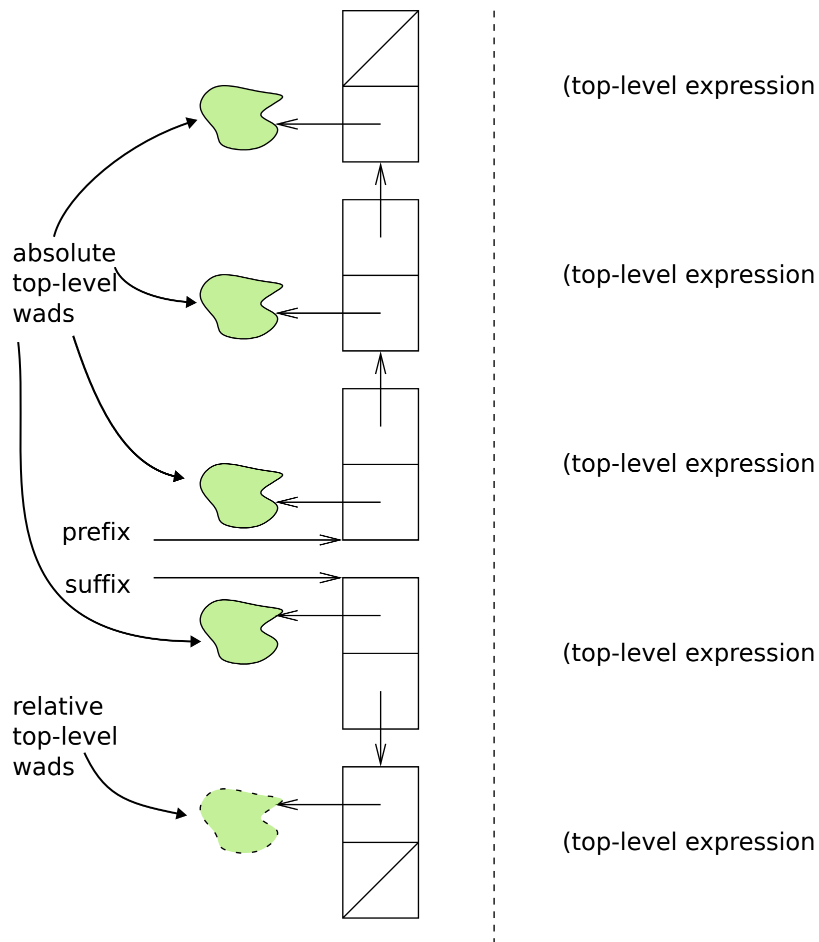

Figure 4.2: Prefix and suffix containing top-level wads.

Either the prefix or the suffix or both may be the empty list. The location L may be moved. It suffices[2] to pop an element off of one of the lists and push it onto the other.

To illustrate the above data structures, we use the following example:

```
...
34 (f 10)
35
36 (let ((x 1)
37       (y 2))
38   (g (h x)
39      (i y)
40      (j x y)))
41
42 (f 20)
...
```

[2] Some slots also need to be updated as will be discussed later.

Each line is preceded by the absolute line number. If the wad starting at line 36 is a member of the prefix or if it is the first element of the suffix, it would be represented like this:

```
36 04 (let ((x 1) (y 2)) (g (h x) (i y) (j x y)))
   00 01 ((x 1) (y 2))
      00 00 (x 1)
      01 00 (y 2)
   02 02 (g (h x) (i y) (j x y))
      00 00 (h x)
      01 00 (i y)
      02 00 (j x y)
```

Since column numbers are uninteresting for our illustration, we show only line numbers. Furthermore, we present a list as a table for a more compact presentation.

4.2 Moving Top-level Wads

Occasionally, some top-level wads need to be moved from the prefix to the suffix or from the suffix to the prefix. There could be several reasons for such moves:

- The place between the prefix and the suffix must always be near the part of the buffer currently on display when the contents are presented to the user. If the part on display changes as a result of scrolling or as a result of the user moving the current cursor, then the prefix and suffix must be adjusted to reflect the new position prior to the presentation.
- After items have been inserted into or deleted from the buffer, the incremental parser may have to adjust the prefix and the suffix so that the altered top-level wads are near the beginning of the suffix.

These adjustments are always accomplished by repeatedly moving a single top-level wad.

To move a single top-level wad P from the prefix to the suffix, the following actions are executed:

1. Modify the slot `start-line` of the first wad of the suffix so that, instead of containing the absolute line number, it contains the line number relative to the value of the slot `start-line` of P.
2. Pop P from the prefix and push it onto the suffix. Rather than using the straightforward technique, the `cons` cell referring to P can be reused so as to avoid unnecessary consing.

To move a single top-level wad P from the suffix to the prefix, the following actions are executed:

1. If P has a successor S in the suffix, then the slot `start-line` of S is adjusted so that it contains the absolute line number as opposed to the line number relative to the slot `start-line` of P.
2. Pop P from the suffix and push it onto the prefix. Rather than using the straightforward technique, the `cons` cell referring to P can be reused so as to avoid unnecessary consing.

We illustrate this process by showing four possible top-level locations in the example buffer. If all three top-level wads are located in the suffix, we have the following situation:

```
prefix
...
```

```
suffix
34 00 (f 10)
02 04 (let ((x 1) (y 2)) (g (h x) (i y) (j x y)))
06 00 (f 20)
...
```

In the example, we do not show the children of the top-level wad.

If the prefix contains the first top-level expression and the suffix the other two, we have the following situation:

```
prefix
...
34 00 (f 10)
suffix
36 04 (let ((x 1) (y 2)) (g (h x) (i y) (j x y)))
06 00 (f 20)
...
```

If the prefix contains the first two top-level expressions and the suffix the remaining one, we have the following situation:

```
prefix
...
34 00 (f 10)
36 04 (let ((x 1) (y 2)) (g (h x) (i y) (j x y)))
suffix
42 00 (f 20)
...
```

Finally, if the prefix contains all three top-level expressions, we have the following situation:

```
prefix
...
34 00 (f 10)
36 04 (let ((x 1) (y 2)) (g (h x) (i y) (j x y)))
42 00 (f 20)
suffix
...
```

4.3 Incremental Update

Modifications to the buffer are reported at the granularity of entire lines. The following operations are possible:

- A line may be modified.
- A line may be inserted.
- A line may be deleted.

Several different lines may be modified between two incremental updates, and in different ways. The first step in an incremental update step is to invalidate wads that are no longer known to be correct after these modifications. This step modifies the data structure described in Section 4.1.2 [Prefix and Suffix], page 6, in the following way:

- After the invalidation step, the prefix contains the wad preceding the first modified line, so that these wads are still valid.
- The suffix contains those wads following the last modified line. These wads are still valid, but they may no longer be top-level wads, because the nesting may have changed as a result of the modifications preceding the suffix.
- An additional list of *residual wads* is created. This list contains wads that have not been invalidated by the modifications, i.e. that appear only in lines that have not been modified.

The order of the wads in the list of residual wads is the same as the order of the wads in the buffer. The slot `start-line` of each wad in the list is the absolute line number of the initial character of that wad.

Suppose, for example, that the buffer contents in our running example was modified so that line 37 was altered in some way, and a line was inserted between the lines 39 and 40. As a result of this update, we need to represent the following wads:

```
...
34 (f 10)
35
36       (x 1)
37
38      (h x)
39      (i y)
40
41      (j x y)
42
43 (f 20)
...
```

In other words, we need to obtain the following representation:

```
prefix
...
34 00 (f 10)
residual
36 00 (x 1)
38 00 (h x)
39 00 (i y)
41 00 (j x y)
suffix
43 00 (f 20)
...
```

4.3.1 Processing Modifications

While the list of residual wads is being constructed, its elements are in the reverse order. Only when all buffer updates have been processed is the list of residual wads reversed to obtain the final representation.

All line modifications are reported in increasing order of line number. Before the first modification is processed, the prefix and the suffix are positioned as indicated above, and the list of residual wads is initialized to the empty list.

The following actions are taken, depending on the position of the modified line with respect to the suffix, and on the nature of the modification:

- If a line has been modified, and either the suffix is empty or the modified line precedes the first wad of the suffix, then no action is taken.
- If a line has been deleted, and the suffix is empty, then no action is taken.
- If a line has been deleted, and it precedes the first wad of the suffix, then the slot `start-line` of the first wad of the suffix is decremented.
- If a line has been inserted, and the suffix is empty, then no action is taken.
- If a line has been inserted, and it precedes the first wad of the suffix, then the slot `start-line` of the first wad of the suffix is incremented.
- If a line has been modified and the entire first wad of the suffix is entirely contained in this line, then remove the first wad from the suffix and start the entire process again with the same modified line. To remove the first wad from the suffix, first adjust the slot `start-line` of the second element of the suffix (if any) to reflect the absolute start line. Then pop the first element off the suffix.
- If a line has been modified, deleted, or inserted, in a way that may affect the first wad of the suffix, then this wad is first removed from the suffix and then processed as indicated below. Finally, start the entire process again with the same modified line. To remove the first wad from the suffix, first adjust the slot `start-line` of the second element of the suffix (if any) to reflect the absolute start line. Then pop the first element off the suffix.

Modifications potentially apply to elements of the suffix. When such an element needs to be taken apart, we try to salvage as many as possible of its descendants. We do this by moving the element to a *worklist* organized as a stack represented as an ordinary Common Lisp list. The top of the stack is taken apart by popping it from the stack and pushing its children. This process goes on until either the top element has no children, or it is no longer affected by a modification to the buffer, in which case it is moved to the list of residual wads.

Let us see how we process the modifications in our running example.

Line 37 has been altered, so our first task is to adjust the prefix and the suffix so that the prefix contains the last wad that is unaffected by the modifications. This adjustment results in the following situation:

```
prefix
...
34 00 (f 10)
residue
worklist
suffix
36 04 (let ((x 1) (y 2)) (g (h x) (i y) (j x y)))
06 00 (f 20)
...
```

The first wad of the suffix is affected by the fact that line 37 has been modified. We must move the children of that wad to the worklist. In doing so, we make the `start-line` of the children reflect the absolute line number, and we also make the `start-line` of the next wad of the suffix also reflect the absolute line number. We obtain the following situation:

```
prefix
...
34 00 (f 10)
residue
worklist
36 01 ((x 1) (y 2))
38 02 (g (h x) (i y) (j x y))
suffix
42 00 (f 20)
...
```

The first element of the worklist is affected by the modification of line 37. We therefore remove it from the worklist, and add its children to the top of the worklist. In doing so, we make the `start-line` of those children reflect absolute line numbers. We obtain the following situation:

```
prefix
...
34 00 (f 10)
residue
worklist
36 00 (x 1)
37 00 (y 2)
38 02 (g (h x) (i y) (j x y))
suffix
42 00 (f 20)
...
```

The first element of the worklist is unaffected by the modification, because it precedes the modified line entirely. We therefore move it to the residue list. We now have the following situation:

```
prefix
...
34 00 (f 10)
residue
36 00 (x 1)
worklist
37 00 (y 2)
38 02 (g (h x) (i y) (j x y))
suffix
42 00 (f 20)
...
```

The first wad of the top of the worklist is affected by the modification. It has no children, so we pop it off the worklist.

```
prefix
...
34 00 (f 10)
residue
36 00 (x 1)
worklist
38 02 (g (h x) (i y) (j x y))
suffix
42 00 (f 20)
...
```

The modification of line 37 is now entirely processed. We know this because the first wad on the worklist occurs beyond the modified line in the buffer. We therefore start processing the line inserted between the existing lines 39 and 40. The first item on the worklist is affected by this insertion. We therefore remove it from the worklist and push its children instead. In doing so, we make the `start-line` slot those children reflect the absolute line number. We obtain the following result:

```
prefix
...
34 00 (f 10)
residue
36 00 (x 1)
worklist
38 00 (h x)
39 00 (i y)
40 00 (j x y)
suffix
42 00 (f 20)
...
```

The first element of the worklist is unaffected by the insertion because it precedes the inserted line entirely. We therefore move it to the residue list. We now have the following situation:

```
prefix
...
34 00 (f 10)
residue
36 00 (x 1)
38 00 (h x)
worklist
39 00 (i y)
40 00 (j x y)
suffix
42 00 (f 20)
...
```

Once again, the first element of the worklist is unaffected by the insertion because it precedes the inserted line entirely. We therefore move it to the residue list. We now have the following situation:

```
prefix
...
34 00 (f 10)
residue
36 00 (x 1)
38 00 (h x)
39 00 (i y)
worklist
40 00 (j x y)
suffix
42 00 (f 20)
...
```

The first element of the worklist is affected by the insertion, in that it must have its line number incremented. In fact, every element of the worklist and also the first element of the suffix must have their line numbers incremented. Furthermore, this update finishes the processing of the inserted line. We now have the following situation:

```
prefix
...
34 00 (f 10)
residue
36 00 (x 1)
38 00 (h x)
39 00 (i y)
worklist
41 00 (j x y)
suffix
43 00 (f 20)
...
```

With no more buffer modifications to process, we terminate the procedure by moving remaining wads from the worklist to the residue list. The final situation is shown here:

```
prefix
...
34 00 (f 10)
residue
36 00 (x 1)
38 00 (h x)
39 00 (i y)
41 00 (j x y)
worklist
suffix
43 00 (f 20)
...
```

4.3.2 Recreating the Cache

Once the cache has been processed so that only wads that are known to be valid remain, the new buffer contents must be fully parsed so that its complete structure is reflected in the cache.

Conceptually, we obtain a complete cache by applying `read` repeatedly from the beginning of the buffer, until all top-level wad have been found. But doing it this way essentially for every keystroke would be too slow. In this section we explain how the partially invalidated cache is used to make this process sufficiently fast.

Concept index

E

edit protocol 2

P

prefix 6

S

suffix 6

T

top-level location 5
top-level wad 5

U

update protocol 2

W

wad 5

Function and macro and variable and type index

(Index is nonexistent)