

Almog JSON Parser

Generated by Doxygen 1.9.1

1 Class Index	1
1.1 Class List	1
2 File Index	3
2.1 File List	3
3 Class Documentation	5
3.1 Lexer Struct Reference	5
3.1.1 Detailed Description	5
3.1.2 Member Data Documentation	5
3.1.2.1 beginning_of_line	6
3.1.2.2 content	6
3.1.2.3 content_len	6
3.1.2.4 cursor	6
3.1.2.5 line_num	6
3.2 Literal_Token Struct Reference	7
3.2.1 Detailed Description	7
3.2.2 Member Data Documentation	7
3.2.2.1 kind	7
3.2.2.2 text	7
3.3 Location Struct Reference	8
3.3.1 Detailed Description	8
3.3.2 Member Data Documentation	8
3.3.2.1 col	8
3.3.2.2 line_num	8
3.4 String Struct Reference	9
3.4.1 Detailed Description	9
3.4.2 Member Data Documentation	9
3.4.2.1 capacity	9
3.4.2.2 elements	9
3.4.2.3 length	9
3.5 Token Struct Reference	10
3.5.1 Detailed Description	10
3.5.2 Member Data Documentation	10
3.5.2.1 kind	10
3.5.2.2 location	11
3.5.2.3 text	11
3.5.2.4 text_len	11
3.6 Tokens Struct Reference	11
3.6.1 Detailed Description	12
3.6.2 Member Data Documentation	12
3.6.2.1 capacity	12
3.6.2.2 content	12

3.6.2.3 elements	13
3.6.2.4 length	13
4 File Documentation	15
4.1 Almog_Dynamic_Array.h File Reference	15
4.1.1 Detailed Description	17
4.1.2 Macro Definition Documentation	17
4.1.2.1 ada_appand	18
4.1.2.2 ADA_ASSERT	18
4.1.2.3 ADA_EXIT	18
4.1.2.4 ada_init_array	19
4.1.2.5 ADA_INIT_CAPACITY	19
4.1.2.6 ada_insert	20
4.1.2.7 ada_insert_unordered	21
4.1.2.8 ADA_MALLOC	21
4.1.2.9 ADA_REALLOC	22
4.1.2.10 ada_remove	22
4.1.2.11 ada_remove_unordered	23
4.1.2.12 ada_resize	23
4.2 Almog_Dynamic_Array.h	24
4.3 Almog_JSON_Parser.h File Reference	26
4.3.1 Detailed Description	26
4.4 Almog_JSON_Parser.h	27
4.5 Almog_Lexer.h File Reference	27
4.5.1 Detailed Description	30
4.5.2 Macro Definition Documentation	30
4.5.2.1 AL_ASSERT	30
4.5.2.2 AL_FREE	30
4.5.2.3 AL_UNUSED	30
4.5.2.4 keywords_count	31
4.5.2.5 literal_tokens_count	31
4.5.3 Enumeration Type Documentation	31
4.5.3.1 Token_Kind	31
4.5.4 Function Documentation	33
4.5.4.1 al_is_identifier()	33
4.5.4.2 al_is_identifier_start()	33
4.5.4.3 al_lex_entire_file()	34
4.5.4.4 al_lexer_alloc()	34
4.5.4.5 al_lexer_chop_char()	35
4.5.4.6 al_lexer_chop_while()	35
4.5.4.7 al_lexer_next_token()	36
4.5.4.8 al_lexer_peek()	37

4.5.4.9 al_lexer_start_with()	37
4.5.4.10 al_lexer_trim_left()	38
4.5.4.11 al_token_kind_name()	38
4.5.4.12 al_token_print()	39
4.5.4.13 al_tokens_free()	39
4.5.4.14 al_tokens_init()	40
4.5.5 Variable Documentation	40
4.5.5.1 keywords	40
4.5.5.2 literal_tokens	40
4.6 Almog_Lexer.h	41
4.7 Almog_String_Manipulation.h File Reference	49
4.7.1 Detailed Description	52
4.7.2 Macro Definition Documentation	53
4.7.2.1 asm_dprintCHAR	53
4.7.2.2 asm_dprintDOUBLE	53
4.7.2.3 asm_dprintERROR	54
4.7.2.4 asm_dprintFLOAT	54
4.7.2.5 asm_dprintINT	54
4.7.2.6 asm_dprintSIZE_T	55
4.7.2.7 asm_dprintSTRING	55
4.7.2.8 asm_max	55
4.7.2.9 ASM_MAX_LEN	56
4.7.2.10 asm_min	56
4.7.3 Function Documentation	57
4.7.3.1 asm_check_char_belong_to_base()	57
4.7.3.2 asm_copy_array_by_indexes()	57
4.7.3.3 asm_get_char_value_in_base()	58
4.7.3.4 asm_get_line()	58
4.7.3.5 asm_get_next_token_from_str()	59
4.7.3.6 asm_get_token_and_cut()	60
4.7.3.7 asm_isalnum()	61
4.7.3.8 asm_isalpha()	61
4.7.3.9 asm_isbdigit()	62
4.7.3.10 asm_iscntrl()	62
4.7.3.11 asm_isdigit()	62
4.7.3.12 asm_isgraph()	63
4.7.3.13 asm_islower()	63
4.7.3.14 asm_isodigit()	64
4.7.3.15 asm_isprint()	64
4.7.3.16 asm_ispunct()	65
4.7.3.17 asm_isspace()	65
4.7.3.18 asm_isupper()	65

4.7.3.19 <code>asm_isxdigit()</code>	66
4.7.3.20 <code>asm_isXdigit()</code>	66
4.7.3.21 <code>asm_length()</code>	67
4.7.3.22 <code>asm_memset()</code>	67
4.7.3.23 <code>asm_pad_left()</code>	68
4.7.3.24 <code>asm_print_many_times()</code>	69
4.7.3.25 <code>asm_remove_char_from_string()</code>	69
4.7.3.26 <code>asm_shift_left()</code>	69
4.7.3.27 <code>asm_str2double()</code>	70
4.7.3.28 <code>asm_str2float()</code>	71
4.7.3.29 <code>asm_str2int()</code>	72
4.7.3.30 <code>asm_str2size_t()</code>	72
4.7.3.31 <code>asm_str_in_str()</code>	73
4.7.3.32 <code>asm_str_is_whitespace()</code>	73
4.7.3.33 <code>asm_strip_whitespace()</code>	74
4.7.3.34 <code>asm_strncat()</code>	74
4.7.3.35 <code>asm_strncmp()</code>	75
4.7.3.36 <code>asm_strncpy()</code>	76
4.7.3.37 <code>asm_tolower()</code>	76
4.7.3.38 <code>asm_toupper()</code>	77
4.7.3.39 <code>asm_trim_left_whitespace()</code>	77
4.8 <code>Almog_String_Manipulation.h</code>	77
4.9 <code>temp.c</code> File Reference	85
4.9.1 Macro Definition Documentation	86
4.9.1.1 <code>ALMOG_JSON_PARSER_IMPLEMENTATION</code>	86
4.9.1.2 <code>ALMOG_LEXER_IMPLEMENTATION</code>	86
4.9.1.3 <code>ALMOG_STRING_MANIPULATION_IMPLEMENTATION</code>	87
4.9.2 Function Documentation	87
4.9.2.1 <code>main()</code>	87
4.10 <code>temp.c</code>	87
Index	89

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Lexer	Lexer state over a caller-provided input buffer	5
Literal_Token	Mapping between a literal operator/punctuation text and a token kind	7
Location	Source location (1-based externally in produced tokens)	8
String	Simple dynamic array of characters (used to hold file content)	9
Token	A token produced by the lexer	10
Tokens	Result of lexing an entire file	11

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Almog_Dynamic_Array.h	
Header-only C macros that implement a simple dynamic array	15
Almog_JSON_Parser.h	26
Almog_Lexer.h	
A small single-header lexer for C/C++-like source text	27
Almog_String_Manipulation.h	
Lightweight string and line manipulation helpers	49
temp.c	85

Chapter 3

Class Documentation

3.1 Lexer Struct Reference

[Lexer](#) state over a caller-provided input buffer.

```
#include <Almog_Lexer.h>
```

Public Attributes

- const char * [content](#)
- size_t [content_len](#)
- size_t [cursor](#)
- size_t [line_num](#)
- size_t [begining_of_line](#)

3.1.1 Detailed Description

[Lexer](#) state over a caller-provided input buffer.

The lexer does not own `content`; the caller must keep it valid for the lifetime of any tokens referencing it.

Internal location tracking:

- `line_num` is 0-based internally (first line is 0).
- `begining_of_line` is the cursor index of the first character of the current line (used for column calculation).

Definition at line [228](#) of file [Almog_Lexer.h](#).

3.1.2 Member Data Documentation

3.1.2.1 begining_of_line

```
size_t Lexer::begining_of_line
```

Definition at line 233 of file [Almog_Lexer.h](#).

Referenced by [al_lexer_chop_char\(\)](#).

3.1.2.2 content

```
const char* Lexer::content
```

Definition at line 229 of file [Almog_Lexer.h](#).

Referenced by [al_lexer_chop_char\(\)](#), [al_lexer_chop_while\(\)](#), [al_lexer_peek\(\)](#), [al_lexer_start_with\(\)](#), and [al_lexer_trim_left\(\)](#).

3.1.2.3 content_len

```
size_t Lexer::content_len
```

Definition at line 230 of file [Almog_Lexer.h](#).

Referenced by [al_lexer_chop_char\(\)](#), [al_lexer_chop_while\(\)](#), [al_lexer_peek\(\)](#), [al_lexer_start_with\(\)](#), and [al_lexer_trim_left\(\)](#).

3.1.2.4 cursor

```
size_t Lexer::cursor
```

Definition at line 231 of file [Almog_Lexer.h](#).

Referenced by [al_lexer_chop_char\(\)](#), [al_lexer_chop_while\(\)](#), [al_lexer_peek\(\)](#), [al_lexer_start_with\(\)](#), and [al_lexer_trim_left\(\)](#).

3.1.2.5 line_num

```
size_t Lexer::line_num
```

Definition at line 232 of file [Almog_Lexer.h](#).

Referenced by [al_lexer_chop_char\(\)](#).

The documentation for this struct was generated from the following file:

- [Almog_Lexer.h](#)

3.2 Literal_Token Struct Reference

Mapping between a literal operator/punctuation text and a token kind.

```
#include <Almog_Lexer.h>
```

Public Attributes

- enum [Token_Kind](#) `kind`
- const char *const `text`

3.2.1 Detailed Description

Mapping between a literal operator/punctuation text and a token kind.

Used internally for longest-match scanning of operators and punctuation.

Note

`text` must be a null-terminated string literal.

Definition at line [154](#) of file [Almog_Lexer.h](#).

3.2.2 Member Data Documentation

3.2.2.1 `kind`

```
enum Token\_Kind Literal_Token::kind
```

Definition at line [955](#) of file [Almog_Lexer.h](#).

3.2.2.2 `text`

```
const char* const Literal_Token::text
```

Definition at line [156](#) of file [Almog_Lexer.h](#).

The documentation for this struct was generated from the following file:

- [Almog_Lexer.h](#)

3.3 Location Struct Reference

Source location (1-based externally in produced tokens).

```
#include <Almog_Lexer.h>
```

Public Attributes

- `size_t` [line_num](#)
- `size_t` [col](#)

3.3.1 Detailed Description

Source location (1-based externally in produced tokens).

`al_lexer_next_token()` stores:

- `line_num`: 1-based line number
- `col`: 1-based column number

Definition at line [167](#) of file [Almog_Lexer.h](#).

3.3.2 Member Data Documentation

3.3.2.1 `col`

```
size_t Location::col
```

Definition at line [169](#) of file [Almog_Lexer.h](#).

Referenced by [al_token_print\(\)](#).

3.3.2.2 `line_num`

```
size_t Location::line_num
```

Definition at line [168](#) of file [Almog_Lexer.h](#).

Referenced by [al_token_print\(\)](#).

The documentation for this struct was generated from the following file:

- [Almog_Lexer.h](#)

3.4 String Struct Reference

Simple dynamic array of characters (used to hold file content).

```
#include <Almog_Lexer.h>
```

Public Attributes

- `size_t` [length](#)
- `size_t` [capacity](#)
- `char *` [elements](#)

3.4.1 Detailed Description

Simple dynamic array of characters (used to hold file content).

This struct is compatible with the dynamic array macros from "Almog_Dynamic_Array.h".

Definition at line [179](#) of file [Almog_Lexer.h](#).

3.4.2 Member Data Documentation

3.4.2.1 capacity

```
size_t String::capacity
```

Definition at line [181](#) of file [Almog_Lexer.h](#).

3.4.2.2 elements

```
char* String::elements
```

Definition at line [182](#) of file [Almog_Lexer.h](#).

3.4.2.3 length

```
size_t String::length
```

Definition at line [180](#) of file [Almog_Lexer.h](#).

The documentation for this struct was generated from the following file:

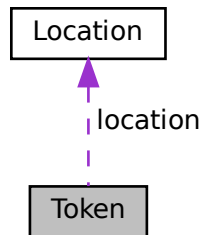
- [Almog_Lexer.h](#)

3.5 Token Struct Reference

A token produced by the lexer.

```
#include <Almog_Lexer.h>
```

Collaboration diagram for Token:



Public Attributes

- enum [Token_Kind](#) `kind`
- const char * `text`
- size_t `text_len`
- struct [Location](#) `location`

3.5.1 Detailed Description

A token produced by the lexer.

`text` points into the original input buffer passed to [al_lexer_alloc](#). The token text is not null-terminated; use `text↵_len`.

Definition at line [191](#) of file [Almog_Lexer.h](#).

3.5.2 Member Data Documentation

3.5.2.1 kind

```
enum Token\_Kind Token::kind
```

Definition at line [182](#) of file [Almog_Lexer.h](#).

Referenced by [al_token_kind_name\(\)](#), and [al_token_print\(\)](#).

3.5.2.2 location

```
struct Location Token::location
```

Definition at line 194 of file [Almog_Lexer.h](#).

Referenced by [al_token_print\(\)](#).

3.5.2.3 text

```
const char* Token::text
```

Definition at line 193 of file [Almog_Lexer.h](#).

Referenced by [al_token_print\(\)](#).

3.5.2.4 text_len

```
size_t Token::text_len
```

Definition at line 194 of file [Almog_Lexer.h](#).

Referenced by [al_token_print\(\)](#).

The documentation for this struct was generated from the following file:

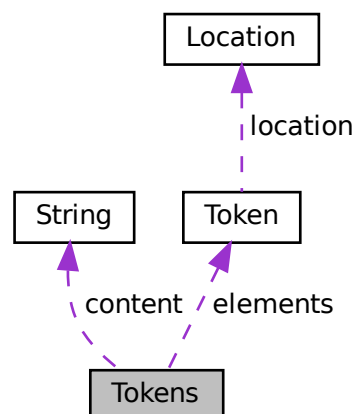
- [Almog_Lexer.h](#)

3.6 Tokens Struct Reference

Result of lexing an entire file.

```
#include <Almog_Lexer.h>
```

Collaboration diagram for Tokens:



Public Attributes

- struct [String](#) `content`
- `size_t` `length`
- `size_t` `capacity`
- struct [Token](#) * `elements`

3.6.1 Detailed Description

Result of lexing an entire file.

Owns 2 dynamic buffers:

- `content`: the concatenated file contents (with ' ' inserted after each line read by [asm_get_line\(\)](#)).
- `elements`: the token array; each token's `text` points into `content`.

Warning

Because tokens reference `content.elements`, `content` must remain alive as long as tokens are used.

Definition at line 210 of file [Almog_Lexer.h](#).

3.6.2 Member Data Documentation

3.6.2.1 capacity

```
size_t Tokens::capacity
```

Definition at line 213 of file [Almog_Lexer.h](#).

3.6.2.2 content

```
struct String Tokens::content
```

Definition at line 194 of file [Almog_Lexer.h](#).

3.6.2.3 elements

```
struct Token* Tokens::elements
```

Definition at line 214 of file [Almog_Lexer.h](#).

Referenced by [main\(\)](#).

3.6.2.4 length

```
size_t Tokens::length
```

Definition at line 212 of file [Almog_Lexer.h](#).

Referenced by [main\(\)](#).

The documentation for this struct was generated from the following file:

- [Almog_Lexer.h](#)

Chapter 4

File Documentation

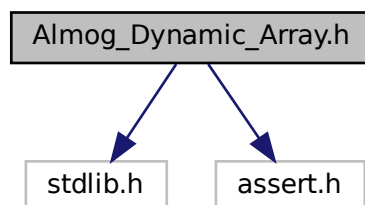
4.1 Almog_Dynamic_Array.h File Reference

Header-only C macros that implement a simple dynamic array.

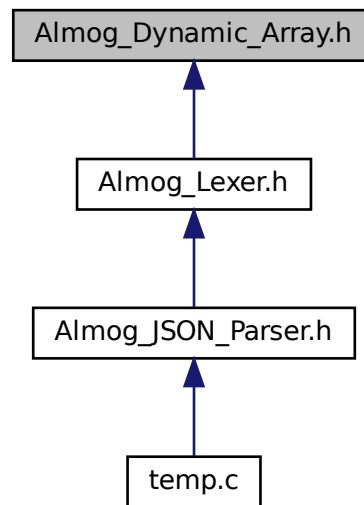
```
#include <stdlib.h>
```

```
#include <assert.h>
```

Include dependency graph for Almog_Dynamic_Array.h:



This graph shows which files directly or indirectly include this file:



Macros

- `#define ADA_INIT_CAPACITY 10`
Default initial capacity used by `ada_init_array`.
- `#define ADA_MALLOC malloc`
Allocation function used by this header (defaults to `malloc`).
- `#define ADA_EXIT exit`
- `#define ADA_REALLOC realloc`
Reallocation function used by this header (defaults to `realloc`).
- `#define ADA_ASSERT assert`
Assertion macro used by this header (defaults to `assert`).
- `#define ada_init_array(type, header)`
Initialize an array header and allocate its initial storage.
- `#define ada_resize(type, header, new_capacity)`
Resize the underlying storage to hold `new_capacity` elements.
- `#define ada_append(type, header, value)`
Append a value to the end of the array, growing if necessary.
- `#define ada_insert(type, header, value, index)`
Insert value at position `index`, preserving order ($O(n)$).
- `#define ada_insert_unordered(type, header, value, index)`
Insert value at `index` without preserving order ($O(1)$ amortized).
- `#define ada_remove(type, header, index)`
Remove element at `index`, preserving order ($O(n)$).
- `#define ada_remove_unordered(type, header, index)`
Remove element at `index` by moving the last element into its place ($O(1)$); order is not preserved.

4.1.1 Detailed Description

Header-only C macros that implement a simple dynamic array.

This header provides a minimal, macro-based dynamic array for POD-like types. The array "header" is a user-defined struct with three fields:

- `size_t` length; current number of elements
- `size_t` capacity; allocated capacity (in elements)
- `T*` elements; pointer to contiguous storage of elements (type T)

How to use: 1) Define a header struct with length/capacity/elements fields. 2) Initialize it with [ada_init_array\(T, header\)](#). 3) Modify it with `ada_appand` (append), `ada_insert`, remove variants, etc. 4) When done, `free(header.elements)` (or your custom deallocator).

Customization:

- Define `ADA_MALLOC`, `ADA_REALLOC`, and `ADA_ASSERT` before including this header to override allocation and assertion behavior.

Complexity (n = number of elements):

- Append: amortized $O(1)$
- Ordered insert/remove: $O(n)$
- Unordered insert/remove: $O(1)$

Notes and limitations:

- These are macros; arguments may be evaluated multiple times. Pass only simple lvalues (no side effects).
- Index checks rely on `ADA_ASSERT`; with `NDEBUG` they may be compiled out.
- `ada_resize` exits the process (`exit(1)`) if reallocation fails.
- `ada_insert` reads `header.elements[header.length - 1]` internally; inserting into an empty array via `ada_insert` is undefined behavior. Use `ada_appand` or `ada_insert_unordered` for that case.
- No automatic shrinking; you may call `ada_resize` manually.

Example: `typedef struct { size_t length; size_t capacity; int* elements; } ada_int_array;`

```
ada_int_array arr; ada_init_array(int, arr); ada_appand(int, arr, 42); ada_insert(int, arr, 7, 0); // requires arr.length > 0
ada_remove(int, arr, 1); free(arr.elements);
```

Definition in file [Almog_Dynamic_Array.h](#).

4.1.2 Macro Definition Documentation

4.1.2.1 ada_append

```
#define ada_append(  
    type,  
    header,  
    value )
```

Value:

```
do {  
    if (header.length >= header.capacity) {  
        ada_resize(type, header, (int)(header.capacity + header.capacity/2 + 1));  
    }  
    header.elements[header.length] = value;  
    header.length++;  
} while (0)
```

Append a value to the end of the array, growing if necessary.

Parameters

<i>type</i>	Element type stored in the array.
<i>header</i>	Lvalue of the header struct.
<i>value</i>	Value to append.

Postcondition

header.length is incremented by 1; the last element equals value.

Note

Growth factor is $(\text{int})(\text{header.capacity} * 1.5)$. Because of truncation, very small capacities may not grow (e.g., from 1 to 1). With the default `INIT_CAPACITY=10` this is typically not an issue unless you manually shrink capacity. Ensure growth always increases capacity by at least 1 if you customize this macro.

Definition at line 176 of file [Almog_Dynamic_Array.h](#).

4.1.2.2 ADA_ASSERT

```
#define ADA_ASSERT assert
```

Assertion macro used by this header (defaults to assert).

Define `ADA_ASSERT` before including this file to override. When `NDEBUG` is defined, standard `assert()` is disabled.

Definition at line 103 of file [Almog_Dynamic_Array.h](#).

4.1.2.3 ADA_EXIT

```
#define ADA_EXIT exit
```

Definition at line 79 of file [Almog_Dynamic_Array.h](#).

4.1.2.4 ada_init_array

```
#define ada_init_array(  
    type,  
    header )
```

Value:

```
do {  
    header.capacity = ADA_INIT_CAPACITY;  
    header.length = 0;  
    header.elements = (type *)ADA_MALLOC(sizeof(type) * header.capacity);  
    ADA_ASSERT(header.elements != NULL);  
} while (0)
```

Initialize an array header and allocate its initial storage.

Parameters

<i>type</i>	Element type stored in the array (e.g., int).
<i>header</i>	Lvalue of the header struct containing fields: length, capacity, and elements.

Precondition

header is a modifiable lvalue; header.elements is uninitialized or ignored and will be overwritten.

Postcondition

header.length == 0, header.capacity == INIT_CAPACITY, header.elements != NULL (or ADA_ASSERT fails).

Note

Allocation uses ADA_MALLOC and is checked via ADA_ASSERT.

Definition at line 127 of file [Almog_Dynamic_Array.h](#).

4.1.2.5 ADA_INIT_CAPACITY

```
#define ADA_INIT_CAPACITY 10
```

Default initial capacity used by ada_init_array.

You may override this by defining ADA_INIT_CAPACITY before including this file.

Definition at line 62 of file [Almog_Dynamic_Array.h](#).

4.1.2.6 ada_insert

```
#define ada_insert(
    type,
    header,
    value,
    index )
```

Value:

```
do {
    ADA_ASSERT((int)(index) >= 0);
    ADA_ASSERT((float)(index) - (int)(index) == 0);
    ada_append(type, header, header.elements[header.length-1]);
    for (int ada_for_loop_index = header.length-2; ada_for_loop_index > (int)(index); ada_for_loop_index--)
    {
        header.elements[ada_for_loop_index] = header.elements [ada_for_loop_index-1];
    }
    header.elements[(int)(index)] = value;
} while (0)
```

Insert value at position index, preserving order (O(n)).

Parameters

<i>type</i>	Element type stored in the array.
<i>header</i>	Lvalue of the header struct.
<i>value</i>	Value to insert.
<i>index</i>	Destination index in the range [0, header.length].

Precondition

$0 \leq \text{index} \leq \text{header.length}$.

$\text{header.length} > 0$ if $\text{index} == \text{header.length}$ (this macro reads the last element internally). For inserting into an empty array, use `ada_append` or `ada_insert_unordered`.

Postcondition

Element is inserted at index; subsequent elements are shifted right; `header.length` is incremented by 1.

Note

This macro asserts index is non-negative and an integer value using `ADA_ASSERT`. No explicit upper-bound assert is performed.

Definition at line 203 of file [Almog_Dynamic_Array.h](#).

4.1.2.7 ada_insert_unordered

```
#define ada_insert_unordered(
    type,
    header,
    value,
    index )
```

Value:

```
do { \
    ADA_ASSERT((int)(index) >= 0); \
    ADA_ASSERT((float)(index) - (int)(index) == 0); \
    if ((size_t)(index) == header.length) { \
        ada_appand(type, header, value); \
    } else { \
        ada_appand(type, header, header.elements[(index)]); \
        header.elements[(index)] = value; \
    } \
} while (0)
```

Insert value at index without preserving order (O(1) amortized).

If index == header.length, this behaves like an append. Otherwise, the current element at index is moved to the end, and value is written at index.

Parameters

<i>type</i>	Element type stored in the array.
<i>header</i>	Lvalue of the header struct.
<i>value</i>	Value to insert.
<i>index</i>	Index in the range [0, header.length].

Precondition

$0 \leq \text{index} \leq \text{header.length}$.

Postcondition

header.length is incremented by 1; array order is not preserved.

Definition at line 229 of file [Almog_Dynamic_Array.h](#).

4.1.2.8 ADA_MALLOC

```
#define ADA_MALLOC malloc
```

Allocation function used by this header (defaults to malloc).

Define ADA_MALLOC to a compatible allocator before including this file to override the default.

Definition at line 74 of file [Almog_Dynamic_Array.h](#).

4.1.2.9 ADA_REALLOC

```
#define ADA_REALLOC realloc
```

Reallocation function used by this header (defaults to realloc).

Define ADA_REALLOC to a compatible reallocator before including this file to override the default.

Definition at line 91 of file [Almog_Dynamic_Array.h](#).

4.1.2.10 ada_remove

```
#define ada_remove(
    type,
    header,
    index )
```

Value:

```
do {
    ADA_ASSERT((int)(index) >= 0);
    ADA_ASSERT((float)(index) - (int)(index) == 0);
    for (size_t ada_for_loop_index = (index); ada_for_loop_index < header.length-1; ada_for_loop_index++) {
        header.elements[ada_for_loop_index] = header.elements[ada_for_loop_index+1];
    }
    header.length--;
} while (0)
```

Remove element at index, preserving order (O(n)).

Parameters

<i>type</i>	Element type stored in the array.
<i>header</i>	Lvalue of the header struct.
<i>index</i>	Index in the range [0, header.length - 1].

Precondition

$0 \leq \text{index} < \text{header.length}$.

Postcondition

header.length is decremented by 1; subsequent elements are shifted left by one position. The element beyond the new length is left uninitialized.

Definition at line 253 of file [Almog_Dynamic_Array.h](#).

4.1.2.11 ada_remove_unordered

```
#define ada_remove_unordered(
    type,
    header,
    index )
```

Value:

```
do {
    ADA_ASSERT((int)(index) >= 0);
    ADA_ASSERT((float)(index) - (int)(index) == 0);
    header.elements[index] = header.elements[header.length-1];
    header.length--;
} while (0)
```

Remove element at index by moving the last element into its place ($O(1)$); order is not preserved.

Parameters

<i>type</i>	Element type stored in the array.
<i>header</i>	Lvalue of the header struct.
<i>index</i>	Index in the range $[0, \text{header.length} - 1]$.

Precondition

$0 \leq \text{index} < \text{header.length}$ and $\text{header.length} > 0$.

Postcondition

header.length is decremented by 1; array order is not preserved.

Definition at line 274 of file [Almog_Dynamic_Array.h](#).

4.1.2.12 ada_resize

```
#define ada_resize(
    type,
    header,
    new_capacity )
```

Value:

```
do {
    type *ada_temp_pointer = (type *)ADA_REALLOC((void *)header.elements, new_capacity*sizeof(type));
    if (ada_temp_pointer == NULL) {
        ADA_EXIT(1);
    }
    header.elements = ada_temp_pointer;
    ADA_ASSERT(header.elements != NULL);
    header.capacity = new_capacity;
} while (0)
```

Resize the underlying storage to hold new_capacity elements.

Parameters

<i>type</i>	Element type stored in the array.
<i>header</i>	Lvalue of the header struct.
<i>new_capacity</i>	New capacity in number of elements.

Precondition

`new_capacity` \geq `header.length` (otherwise elements beyond `new_capacity` are lost and length will not be adjusted).

Postcondition

`header.capacity` == `new_capacity` and `header.elements` points to a block large enough for `new_capacity` elements.

Warning

On allocation failure, this macro calls [ADA_EXIT\(1\)](#).

Note

Reallocation uses `ADA_REALLOC` and is also checked via `ADA_ASSERT`.

Definition at line 150 of file [Almog_Dynamic_Array.h](#).

4.2 Almog_Dynamic_Array.h

```

00001
00051 #ifndef ALMOG_DYNAMIC_ARRAY_H_
00052 #define ALMOG_DYNAMIC_ARRAY_H_
00053
00054
00061 #ifndef ADA_INIT_CAPACITY
00062 #define ADA_INIT_CAPACITY 10
00063 #endif /*ADA_INIT_CAPACITY*/
00064
00072 #ifndef ADA_MALLOC
00073 #include <stdlib.h>
00074 #define ADA_MALLOC malloc
00075 #endif /*ADA_MALLOC*/
00076
00077 #ifndef ADA_EXIT
00078 #include <stdlib.h>
00079 #define ADA_EXIT exit
00080 #endif /*ADA_EXIT*/
00081
00089 #ifndef ADA_REALLOC
00090 #include <stdlib.h>
00091 #define ADA_REALLOC realloc
00092 #endif /*ADA_REALLOC*/
00093
00101 #ifndef ADA_ASSERT
00102 #include <assert.h>
00103 #define ADA_ASSERT assert
00104 #endif /*ADA_ASSERT*/
00105
00106 /* typedef struct {
00107     size_t length;
00108     size_t capacity;
00109     int* elements;
00110 } ada_int_array; */
00111
00127 #define ada_init_array(type, header) do {

```

```

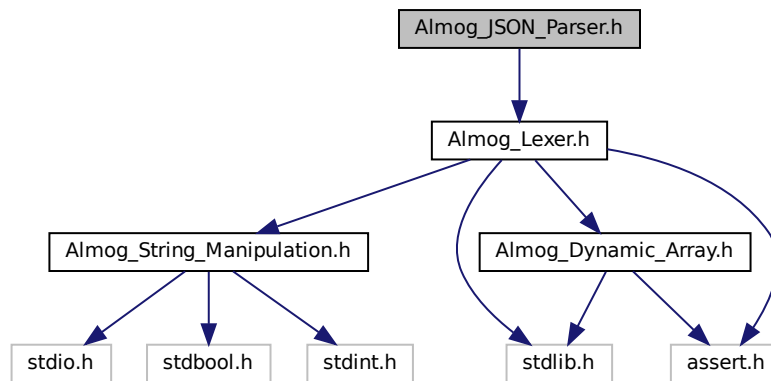
00128         header.capacity = ADA_INIT_CAPACITY;
00129         header.length = 0;
00130         header.elements = (type *)ADA_MALLOC(sizeof(type) * header.capacity);
00131         ADA_ASSERT(header.elements != NULL);
00132     } while (0)
00133
00150 #define ada_resize(type, header, new_capacity) do {
00151     \
00152     type *ada_temp_pointer = (type *)ADA_REALLOC((void *) (header.elements),
00153     new_capacity*sizeof(type)); \
00154     if (ada_temp_pointer == NULL) {
00155     \
00156         ADA_EXIT(1);
00157     \
00158     }
00159     header.elements = ada_temp_pointer;
00160     ADA_ASSERT(header.elements != NULL);
00161     header.capacity = new_capacity;
00162 } while (0)
00163
00176 #define ada_appand(type, header, value) do {
00177     \
00178     if (header.length >= header.capacity) {
00179         \
00180         ada_resize(type, header, (int)(header.capacity + header.capacity/2 + 1));
00181     }
00182     header.elements[header.length] = value;
00183     header.length++;
00184 } while (0)
00185
00203 #define ada_insert(type, header, value, index) do {
00204     \
00205     ADA_ASSERT((int)(index) >= 0);
00206     ADA_ASSERT((float)(index) - (int)(index) == 0);
00207     \
00208     ada_appand(type, header, header.elements[header.length-1]);
00209     \
00210     for (int ada_for_loop_index = header.length-2; ada_for_loop_index > (int)(index);
00211     ada_for_loop_index--) { \
00212         header.elements[ada_for_loop_index] = header.elements [ada_for_loop_index-1];
00213     }
00214     \
00215     header.elements[(int)(index)] = value;
00216 } while (0)
00217
00229 #define ada_insert_unordered(type, header, value, index) do { \
00230     ADA_ASSERT((int)(index) >= 0); \
00231     ADA_ASSERT((float)(index) - (int)(index) == 0); \
00232     if ((size_t)(index) == header.length) { \
00233         \
00234         ada_appand(type, header, value); \
00235     } else { \
00236         \
00237         ada_appand(type, header, header.elements[(int)(index)]); \
00238         header.elements[(int)(index)] = value; \
00239     } \
00240 } while (0)
00241
00253 #define ada_remove(type, header, index) do {
00254     \
00255     ADA_ASSERT((int)(index) >= 0);
00256     ADA_ASSERT((float)(index) - (int)(index) == 0);
00257     \
00258     for (size_t ada_for_loop_index = (int)(index); ada_for_loop_index < header.length-1;
00259     ada_for_loop_index++) { \
00260         header.elements[ada_for_loop_index] = header.elements[ada_for_loop_index+1];
00261     }
00262     \
00263     header.length--;
00264 } while (0)
00265
00274 #define ada_remove_unordered(type, header, index) do { \
00275     ADA_ASSERT((int)(index) >= 0); \
00276     ADA_ASSERT((float)(index) - (int)(index) == 0); \
00277     header.elements[index] = header.elements[header.length-1]; \
00278     header.length--; \
00279 } while (0)
00280
00281
00282 #endif /*ALMOG_DYNAMIC_ARRAY_H*/

```

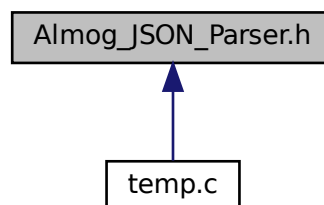
4.3 Almog_JSON_Parser.h File Reference

```
#include "Almog_Lexer.h"
```

Include dependency graph for Almog_JSON_Parser.h:



This graph shows which files directly or indirectly include this file:



4.3.1 Detailed Description

Note

This single header library is inspired by Tsoding's JSON parser implementation: <https://youtu.be/FBpgdSjJ6nQ>

Definition in file [Almog_JSON_Parser.h](#).

4.4 Almog_JSON_Parser.h

```

00001
00008 #ifndef ALMOG_JSON_PARSER_H_
00009 #define ALMOG_JSON_PARSER_H_
00010
00011 #include "Almog_Lexer.h"
00012
00013
00014
00015
00016 #endif /*ALMOG_JSON_PARSER_H_*/
00017
00018 #ifdef ALMOG_JSON_PARSER_IMPLEMENTATION
00019 #undef ALMOG_JSON_PARSER_IMPLEMENTATION
00020
00021
00022
00023
00024 #endif /*ALMOG_JSON_PARSER_IMPLEMENTATION*/
00025

```

4.5 Almog_Lexer.h File Reference

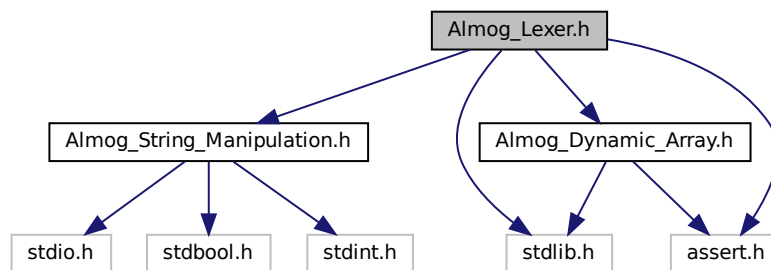
A small single-header lexer for C/C++-like source text.

```

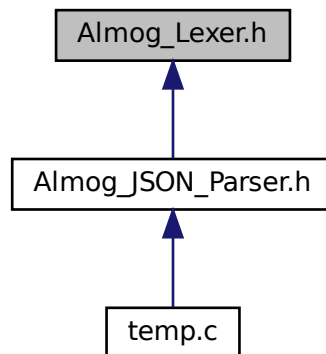
#include "Almog_String_Manipulation.h"
#include "Almog_Dynamic_Array.h"
#include <assert.h>
#include <stdlib.h>

```

Include dependency graph for Almog_Lexer.h:



This graph shows which files directly or indirectly include this file:



Classes

- struct [Literal_Token](#)
Mapping between a literal operator/punctuation text and a token kind.
- struct [Location](#)
Source location (1-based externally in produced tokens).
- struct [String](#)
Simple dynamic array of characters (used to hold file content).
- struct [Token](#)
A token produced by the lexer.
- struct [Tokens](#)
Result of lexing an entire file.
- struct [Lexer](#)
[Lexer](#) state over a caller-provided input buffer.

Macros

- #define [AL_ASSERT](#) `assert`
Assertion macro used by the lexer (defaults to `assert()`).
- #define [AL_FREE](#) `free`
Deallocation macro used by [al_tokens_free\(\)](#) (defaults to `free()`).
- #define [literal_tokens_count](#) `(sizeof(literal_tokens) / sizeof(literal_tokens[0]))`
- #define [keywords_count](#) `(sizeof(keywords) / sizeof(keywords[0]))`
- #define [AL_UNUSED](#)(x) `(void)x`
Mark a variable as intentionally unused.

Enumerations

- enum [Token_Kind](#) {
[TOKEN_EOF](#), [TOKEN_INVALID](#), [TOKEN_PP_DIRECTIVE](#), [TOKEN_COMMENT](#),
[TOKEN_STRING_LIT](#), [TOKEN_CHAR_LIT](#), [TOKEN_INT_LIT_BIN](#), [TOKEN_INT_LIT_OCT](#),
[TOKEN_INT_LIT_DEC](#), [TOKEN_INT_LIT_HEX](#), [TOKEN_FLOAT_LIT_DEC](#), [TOKEN_FLOAT_LIT_HEX](#),
[TOKEN_KEYWORD](#), [TOKEN_IDENTIFIER](#), [TOKEN_LPAREN](#), [TOKEN_RPAREN](#),
[TOKEN_LBRACKET](#), [TOKEN_RBRACKET](#), [TOKEN_LBRACE](#), [TOKEN_RBRACE](#),
[TOKEN_DOT](#), [TOKEN_COMMA](#), [TOKEN_SEMICOLON](#), [TOKEN_BSLASH](#),
[TOKEN_HASH](#), [TOKEN_QUESTION](#), [TOKEN_COLON](#), [TOKEN_EQ](#),
[TOKEN_EQEQ](#), [TOKEN_NE](#), [TOKEN_BANG](#), [TOKEN_LT](#),
[TOKEN_GT](#), [TOKEN_LE](#), [TOKEN_GE](#), [TOKEN_BITAND](#),
[TOKEN_ANDAND](#), [TOKEN_BITOR](#), [TOKEN_OROR](#), [TOKEN_CARET](#),
[TOKEN_TILDE](#), [TOKEN_LSHIFT](#), [TOKEN_RSHIFT](#), [TOKEN_PLUSPLUS](#),
[TOKEN_MINUSMINUS](#), [TOKEN_PLUS](#), [TOKEN_MINUS](#), [TOKEN_STAR](#),
[TOKEN_SLASH](#), [TOKEN_PERCENT](#), [TOKEN_PLUSEQ](#), [TOKEN_MINUSEQ](#),
[TOKEN_STAREQ](#), [TOKEN_SLASHEQ](#), [TOKEN_PERCENTEQ](#), [TOKEN_ANDEQ](#),
[TOKEN_OREQ](#), [TOKEN_XOREQ](#), [TOKEN_LSHIFTEQ](#), [TOKEN_RSHIFTEQ](#),
[TOKEN_ARROW](#), [TOKEN_ELLIPSIS](#) }

Token categories produced by the lexer.

Functions

- bool [al_is_identifier](#) (char c)
Returns whether c can appear in an identifier after the first character.
- bool [al_is_identifier_start](#) (char c)
Returns whether c can start an identifier.
- struct [Tokens](#) [al_lex_entire_file](#) (FILE *fp)
- struct [Lexer](#) [al_lexer_alloc](#) (const char *content, size_t len)
Create a lexer over an input buffer.
- char [al_lexer_chop_char](#) (struct [Lexer](#) *l)
Consume and return the next character from the input.
- void [al_lexer_chop_while](#) (struct [Lexer](#) *l, bool(*pred)(char))
Consume characters while pred returns true for the next character.
- struct [Token](#) [al_lexer_next_token](#) (struct [Lexer](#) *l)
Return the next token from the input and advance the lexer.
- bool [al_lexer_start_with](#) (struct [Lexer](#) *l, const char *prefix)
Check whether the remaining input at the current cursor starts with prefix.
- void [al_lexer_trim_left](#) (struct [Lexer](#) *l)
Consume leading whitespace characters.
- char [al_lexer_peek](#) (const struct [Lexer](#) *l, size_t off)
Peek at a character in the input without advancing the lexer.
- void [al_token_print](#) (struct [Token](#) tok)
Print a human-readable representation of tok to stdout.
- const char * [al_token_kind_name](#) (enum [Token_Kind](#) kind)
Convert a token kind enum to a stable string name.
- struct [Tokens](#) [al_tokens_init](#) (void)
- void [al_tokens_free](#) (struct [Tokens](#) tokens)

Variables

- static struct [Literal_Token](#) [literal_tokens](#) []
Operator/punctuation token table.
- static const char *const [keywords](#) []
List of keywords recognized by the lexer.

4.5.1 Detailed Description

A small single-header lexer for C/C++-like source text.

The lexer operates on a caller-provided, read-only character buffer. It produces tokens that reference slices of the original buffer (no allocations and no null-termination guarantees).

Note

This header depends on "Almog_String_Manipulation.h" for the `asm_*` character classification and string helper routines used by the implementation (e.g. `asm_isalpha`, `asm_isspace`, etc.).

This single header library is inspired by Tsoding's C-lexer implementation: <https://youtu.be/↵AqyZztKlSGQ>

Definition in file [Almog_Lexer.h](#).

4.5.2 Macro Definition Documentation

4.5.2.1 AL_ASSERT

```
#define AL_ASSERT assert
```

Assertion macro used by the lexer (defaults to `assert()`).

Define `AL_ASSERT` before including this header to override.

Definition at line 30 of file [Almog_Lexer.h](#).

4.5.2.2 AL_FREE

```
#define AL_FREE free
```

Deallocation macro used by [al_tokens_free\(\)](#) (defaults to `free()`).

Define `AL_FREE` before including this header to override.

Definition at line 41 of file [Almog_Lexer.h](#).

4.5.2.3 AL_UNUSED

```
#define AL_UNUSED(  
    x ) (void)x
```

Mark a variable as intentionally unused.

Parameters

x	Expression evaluated for side effects (if any) and then cast to void to suppress unused warnings.
---	---

Definition at line 328 of file [Almog_Lexer.h](#).

4.5.2.4 keywords_count

```
#define keywords_count (sizeof(keywords) / sizeof(keywords[0]))
```

Definition at line 319 of file [Almog_Lexer.h](#).

4.5.2.5 literal_tokens_count

```
#define literal_tokens_count (sizeof(literal_tokens) / sizeof(literal_tokens[0]))
```

Definition at line 296 of file [Almog_Lexer.h](#).

4.5.3 Enumeration Type Documentation

4.5.3.1 Token_Kind

```
enum Token_Kind
```

[Token](#) categories produced by the lexer.

The lexer attempts to classify source text into:

- high-level "word-like" tokens (identifiers, keywords, literals, comments)
- punctuation / operators (matched using the longest-match rule)
- TOKEN_INVALID for unrecognized or malformed sequences
- TOKEN_EOF at end of input

Enumerator

TOKEN_EOF	
TOKEN_INVALID	
TOKEN_PP_DIRECTIVE	
TOKEN_COMMENT	
TOKEN_STRING_LIT	

Enumerator

TOKEN_CHAR_LIT	
TOKEN_INT_LIT_BIN	
TOKEN_INT_LIT_OCT	
TOKEN_INT_LIT_DEC	
TOKEN_INT_LIT_HEX	
TOKEN_FLOAT_LIT_DEC	
TOKEN_FLOAT_LIT_HEX	
TOKEN_KEYWORD	
TOKEN_IDENTIFIER	
TOKEN_LPAREN	
TOKEN_RPAREN	
TOKEN_LBRACKET	
TOKEN_RBRACKET	
TOKEN_LBRACE	
TOKEN_RBRACE	
TOKEN_DOT	
TOKEN_COMMA	
TOKEN_SEMICOLON	
TOKEN_BSLASH	
TOKEN_HASH	
TOKEN_QUESTION	
TOKEN_COLON	
TOKEN_EQ	
TOKEN_EQEQ	
TOKEN_NE	
TOKEN_BANG	
TOKEN_LT	
TOKEN_GT	
TOKEN_LE	
TOKEN_GE	
TOKEN_BITAND	
TOKEN_ANDAND	
TOKEN_BITOR	
TOKEN_OROR	
TOKEN_CARET	
TOKEN_TILDE	
TOKEN_LSHIFT	
TOKEN_RSHIFT	
TOKEN_PLUSPLUS	
TOKEN_MINUSMINUS	
TOKEN_PLUS	
TOKEN_MINUS	
TOKEN_STAR	
TOKEN_SLASH	
TOKEN_PERCENT	
TOKEN_PLUSEQ	
TOKEN_MINUSEQ	
TOKEN_STAREQ	
TOKEN_SLASHEQ	

Enumerator

TOKEN_PERCENTEQ	
TOKEN_ANDEQ	
TOKEN_OREQ	
TOKEN_XOREQ	
TOKEN_LSHIFTEQ	
TOKEN_RSHIFTEQ	
TOKEN_ARROW	
TOKEN_ELLIPSIS	

Definition at line 54 of file [Almog_Lexer.h](#).

4.5.4 Function Documentation

4.5.4.1 al_is_identifier()

```
bool al_is_identifier (
    char c )
```

Returns whether `c` can appear in an identifier after the first character.

Matches the implementation: alphanumeric (per `asm_isalnum`) or underscore.

Parameters

<code>c</code>	Character to test.
----------------	--------------------

Returns

true if `c` is valid as a non-initial identifier character.

Definition at line 359 of file [Almog_Lexer.h](#).

References [asm_isalnum\(\)](#).

4.5.4.2 al_is_identifier_start()

```
bool al_is_identifier_start (
    char c )
```

Returns whether `c` can start an identifier.

Matches the implementation: alphabetic (per `asm_isalpha`) or underscore.

Parameters

<i>c</i>	Character to test.
----------	--------------------

Returns

true if *c* is valid as an initial identifier character.

Definition at line 372 of file [Almog_Lexer.h](#).

References [asm_isalpha\(\)](#).

4.5.4.3 al_lex_entire_file()

```
struct Tokens al_lex_entire_file (
    FILE * fp )
```

Definition at line 372 of file [Almog_Lexer.h](#).

Referenced by [main\(\)](#).

4.5.4.4 al_lexer_alloc()

```
struct Lexer al_lexer_alloc (
    const char * content,
    size_t len )
```

Create a lexer over an input buffer.

Initializes cursor and location state to the beginning of the buffer. No memory is allocated; the lexer holds only pointers/indices.

Parameters

<i>content</i>	Pointer to the input text (need not be null-terminated).
<i>len</i>	Length of <i>content</i> in bytes.

Returns

A lexer initialized to the start of *content*.

Definition at line 372 of file [Almog_Lexer.h](#).

4.5.4.5 al_lexer_chop_char()

```
char al_lexer_chop_char (
    struct Lexer * l )
```

Consume and return the next character from the input.

Advances the lexer's cursor by 1. If the consumed character is a newline ('\n'), the lexer's internal line/column bookkeeping is updated: `-line_numis` incremented `-begining_of_line`` is set to the new cursor position

Parameters

/	Lexer to advance.
---	-----------------------------------

Returns

The consumed character.

Precondition

`l->cursor < l->content_len` (enforced via `AL_ASSERT` in the implementation).

Definition at line 434 of file [Almog_Lexer.h](#).

References [AL_ASSERT](#), [Lexer::begining_of_line](#), [Lexer::content](#), [Lexer::content_len](#), [Lexer::cursor](#), and [Lexer::line_num](#).

Referenced by [al_lexer_chop_while\(\)](#), and [al_lexer_trim_left\(\)](#).

4.5.4.6 al_lexer_chop_while()

```
void al_lexer_chop_while (
    struct Lexer * l,
    bool(*) (char) pred )
```

Consume characters while `pred` returns true for the next character.

Uses [al_lexer_chop_char](#) internally, so newline bookkeeping is applied.

Parameters

/	Lexer to advance.
<i>pred</i>	Predicate called with the next character to decide whether to consume it.

Definition at line 454 of file [Almog_Lexer.h](#).

References [al_lexer_chop_char\(\)](#), [Lexer::content](#), [Lexer::content_len](#), and [Lexer::cursor](#).

4.5.4.7 al_lexer_next_token()

```
struct Token al_lexer_next_token (
    struct Lexer * l )
```

Return the next token from the input and advance the lexer.

This function first calls [al_lexer_trim_left](#), so leading whitespace is skipped (including newlines).

The returned token:

- has `text` pointing into the original buffer at the token start
- has `text_len` equal to the number of bytes consumed for the token
- has 1-based `location.line_num` and 1-based `location.col`

Tokenization behavior matches the implementation:

- End of input => `TOKEN_EOF`
- Preprocessor directive: a `#` at column 1 (after trimming) consumes until newline (and includes the newline if present) => `TOKEN_PP_DIRECTIVE`
- Identifiers: `[A-Za-z_][A-Za-z0-9_]*` => `TOKEN_IDENTIFIER`, upgraded to `TOKEN_KEYWORD` if it matches an entry in `keywords[]`
- **String** literal: starts with `"` and consumes until the next `"` or newline (includes the closing `"` if present) => `TOKEN_STRING_LIT`
- Character literal: starts with `'` and consumes until the next `'` or newline (includes the closing `'` if present) => `TOKEN_CHAR_LIT`
- Line comment: starts with `//` and consumes until newline (and includes the newline if present) => `TOKEN_COMMENT`
- Block comment: starts with `/ *` and consumes until the first `* /` (includes the final `/`), or until end of input => `TOKEN_COMMENT`
- Number literals:
 - decimal integers/floats with optional exponent (`e/E`)
 - hex integers and hex floats (hex float requires `p/P` exponent when a fractional part is present)
 - binary integers with `0b/0B`
 - explicit octal integers with `0o/0O`
 - accepts common integer suffixes (`uUllzZ`) and float suffixes (`fFlL`)
 - certain malformed forms are returned as `TOKEN_INVALID`
- Otherwise: matches the longest operator/punctuation from `literal_tokens[]` (longest-match rule) and returns its kind
- If nothing matches, consumes one character and returns `TOKEN_INVALID`

Warning

Escape sequences in string/character literals are not interpreted; a quote character ends the literal even if preceded by a backslash.

Parameters

<i>l</i>	Lexer to tokenize from.
----------	---

Returns

The next token.

Definition at line 454 of file [Almog_Lexer.h](#).

4.5.4.8 al_lexer_peek()

```
char al_lexer_peek (
    const struct Lexer * l,
    size_t off )
```

Peek at a character in the input without advancing the lexer.

Parameters

<i>l</i>	Lexer to read from.
<i>off</i>	Offset from the current cursor (0 means current character).

Returns

The character at `cursor + off`, or `"\0"` if out of range.

Definition at line 781 of file [Almog_Lexer.h](#).

References [Lexer::content](#), [Lexer::content_len](#), and [Lexer::cursor](#).

4.5.4.9 al_lexer_start_with()

```
bool al_lexer_start_with (
    struct Lexer * l,
    const char * prefix )
```

Check whether the remaining input at the current cursor starts with `prefix`.

Parameters

<i>l</i>	Lexer whose input is tested.
<i>prefix</i>	Null-terminated prefix string to match.

Returns

true if `prefix` is empty or fully matches at the current cursor; false otherwise.

Definition at line 739 of file [Almog_Lexer.h](#).

References [asm_length\(\)](#), [Lexer::content](#), [Lexer::content_len](#), and [Lexer::cursor](#).

4.5.4.10 al_lexer_trim_left()

```
void al_lexer_trim_left (
    struct Lexer * l )
```

Consume leading whitespace characters.

Whitespace is defined by `asm_isspace` from "Almog_String_Manipulation.h". Uses [al_lexer_chop_char](#), so new-lines update line/column bookkeeping.

Parameters

/	Lexer to advance.
---	-----------------------------------

Definition at line 764 of file [Almog_Lexer.h](#).

References [al_lexer_chop_char\(\)](#), [asm_isspace\(\)](#), [Lexer::content](#), [Lexer::content_len](#), and [Lexer::cursor](#).

4.5.4.11 al_token_kind_name()

```
const char * al_token_kind_name (
    enum Token_Kind kind )
```

Convert a token kind enum to a stable string name.

The returned pointer refers to a string literal.

Parameters

<i>kind</i>	Token kind.
-------------	-----------------------------

Returns

A string name such as "TOKEN_IDENTIFIER", or asserts on unknown kinds in the implementation's default case.

Definition at line 813 of file [Almog_Lexer.h](#).

References [AL_ASSERT](#), [Token::kind](#), [TOKEN_ANDAND](#), [TOKEN_ANDEQ](#), [TOKEN_ARROW](#), [TOKEN_BANG](#), [TOKEN_BITAND](#), [TOKEN_BITOR](#), [TOKEN_BSLASH](#), [TOKEN_CARET](#), [TOKEN_CHAR_LIT](#), [TOKEN_COLON](#),

TOKEN_COMMA, TOKEN_COMMENT, TOKEN_DOT, TOKEN_ELLIPSIS, TOKEN_EOF, TOKEN_EQ, TOKEN_EQEQ, TOKEN_FLOAT_LIT_DEC, TOKEN_FLOAT_LIT_HEX, TOKEN_GE, TOKEN_GT, TOKEN_HASH, TOKEN_IDENTIFIER, TOKEN_INT_LIT_BIN, TOKEN_INT_LIT_DEC, TOKEN_INT_LIT_HEX, TOKEN_INT_LIT_OCT, TOKEN_INVALID, TOKEN_KEYWORD, TOKEN_LBRACE, TOKEN_LBRACKET, TOKEN_LE, TOKEN_LPAREN, TOKEN_LSHIFT, TOKEN_LSHIFTEQ, TOKEN_LT, TOKEN_MINUS, TOKEN_MINUSEQ, TOKEN_MINUSMINUS, TOKEN_NE, TOKEN_OREQ, TOKEN_OROR, TOKEN_PERCENT, TOKEN_PERCENTEQ, TOKEN_PLUS, TOKEN_PLUSEQ, TOKEN_PLUSPLUS, TOKEN_PP_DIRECTIVE, TOKEN_QUESTION, TOKEN_RBRACE, TOKEN_RBRACKET, TOKEN_RPAREN, TOKEN_RSHIFT, TOKEN_RSHIFTEQ, TOKEN_SEMICOLON, TOKEN_SLASH, TOKEN_SLASHEQ, TOKEN_STAR, TOKEN_STAREQ, TOKEN_STRING_LIT, TOKEN_TILDE, and TOKEN_XOREQ.

Referenced by [al_token_print\(\)](#).

4.5.4.12 al_token_print()

```
void al_token_print (
    struct Token tok )
```

Print a human-readable representation of `tok` to stdout.

Output format matches the implementation: `line:col:(KIND) -> "TEXT"`

Note

The token text is printed using a precision specifier (`%.*s`) and does not need to be null-terminated.

Parameters

<i>tok</i>	Token to print.
------------	---------------------------------

Definition at line 799 of file [Almog_Lexer.h](#).

References [al_token_kind_name\(\)](#), [Location::col](#), [Token::kind](#), [Location::line_num](#), [Token::location](#), [Token::text](#), and [Token::text_len](#).

Referenced by [main\(\)](#).

4.5.4.13 al_tokens_free()

```
void al_tokens_free (
    struct Tokens tokens )
```

Definition at line 955 of file [Almog_Lexer.h](#).

Referenced by [main\(\)](#).

4.5.4.14 `al_tokens_init()`

```
struct Tokens al_tokens_init (
    void )
```

Definition at line 813 of file [Almog_Lexer.h](#).

4.5.5 Variable Documentation

4.5.5.1 keywords

```
const char* const keywords[] [static]
```

Initial value:

```
= {
    "auto", "break", "case", "char", "const", "continue", "default", "do", "double",
    "else", "enum", "extern", "float", "for", "goto", "if", "int", "long", "register",
    "return", "short", "signed", "sizeof", "static", "struct", "switch", "typedef",
    "union", "unsigned", "void", "volatile", "while", "alignas", "alignof", "and",
    "and_eq", "asm", "atomic_cancel", "atomic_commit", "atomic_noexcept", "bitand",
    "bitor", "bool", "catch", "char16_t", "char32_t", "char8_t", "class", "co_await",
    "co_return", "co_yield", "compl", "concept", "const_cast", "constexpr", "consteval",
    "constinit", "decltype", "delete", "dynamic_cast", "explicit", "export", "false",
    "friend", "inline", "mutable", "namespace", "new", "noexcept", "not", "not_eq",
    "nullptr", "operator", "or", "or_eq", "private", "protected", "public", "reflexpr",
    "reinterpret_cast", "requires", "static_assert", "static_cast", "synchronized",
    "template", "this", "thread_local", "throw", "true", "try", "typeid", "typename",
    "using", "virtual", "wchar_t", "xor", "xor_eq",
}
```

List of keywords recognized by the lexer.

If an identifier's spelling matches one of these strings exactly, the lexer produces `TOKEN_KEYWORD` instead of `TOKEN_IDENTIFIER`.

Definition at line 304 of file [Almog_Lexer.h](#).

4.5.5.2 `literal_tokens`

```
struct Literal-Token literal_tokens[] [static]
```

Operator/punctuation token table.

The lexer uses this table to apply a longest-match rule for multi-character operators (e.g. `">>="` over `">>"` and `">"`).

Note

This table is defined in the header as `static`, so each translation unit gets its own copy.

Definition at line 1 of file [Almog_Lexer.h](#).

4.6 Almog_Lexer.h

```

00001
00016 #ifndef ALMOG_LEXER_H_
00017 #define ALMOG_LEXER_H_
00018
00019 #include "Almog_String_Manipulation.h"
00020 #include "Almog_Dynamic_Array.h"
00021
00028 #ifndef AL_ASSERT
00029 #include <assert.h>
00030 #define AL_ASSERT assert
00031 #endif /* AL_ASSERT */
00032
00039 #ifndef AL_FREE
00040 #include <stdlib.h>
00041 #define AL_FREE free
00042 #endif /* AL_FREE */
00043
00054 enum Token_Kind {
00055     /* Sentinel / unknown */
00056     TOKEN_EOF,
00057     TOKEN_INVALID,
00058
00059     /* High-level / multi-char / "word-like" */
00060     TOKEN_PP_DIRECTIVE,
00061     TOKEN_COMMENT,
00062     TOKEN_STRING_LIT,
00063     TOKEN_CHAR_LIT,
00064     TOKEN_INT_LIT_BIN,
00065     TOKEN_INT_LIT_OCT,
00066     TOKEN_INT_LIT_DEC,
00067     TOKEN_INT_LIT_HEX,
00068     TOKEN_FLOAT_LIT_DEC,
00069     TOKEN_FLOAT_LIT_HEX,
00070     TOKEN_KEYWORD,
00071     TOKEN_IDENTIFIER,
00072
00073
00074     /* Grouping / separators */
00075     TOKEN_LPAREN,
00076     TOKEN_RPAREN,
00077     TOKEN_LBRACKET,
00078     TOKEN_RBRACKET,
00079     TOKEN_LBRACE,
00080     TOKEN_RBRACE,
00081
00082     /* Punctuation */
00083     TOKEN_DOT,
00084     TOKEN_COMMA,
00085     TOKEN_SEMICOLON,
00086     TOKEN_BSLASH,
00087     TOKEN_HASH,
00088
00089     /* Ternary */
00090     TOKEN_QUESTION,
00091     TOKEN_COLON,
00092
00093     /* Assignment / equality */
00094     TOKEN_EQ,
00095     TOKEN_EQEQ,
00096     TOKEN_NE,
00097     TOKEN_BANG,
00098
00099     /* Relational */
00100     TOKEN_LT,
00101     TOKEN_GT,
00102     TOKEN_LE,
00103     TOKEN_GE,
00104
00105     /* Bitwise / boolean */
00106     TOKEN_BITAND,
00107     TOKEN_ANDAND,
00108     TOKEN_BITOR,
00109     TOKEN_OROR,
00110     /* Bitwise unary */
00111     TOKEN_CARET,
00112     TOKEN_TILDE,
00113
00114     /* Shifts */
00115     TOKEN_LSHIFT,
00116     TOKEN_RSHIFT,
00117
00118     /* Inc / dec */
00119     TOKEN_PLUSPLUS,
00120     TOKEN_MINUSMINUS,
00121

```

```

00122     /* Arithmetic */
00123     TOKEN_PLUS,
00124     TOKEN_MINUS,
00125     TOKEN_STAR,
00126     TOKEN_SLASH,
00127     TOKEN_PERCENT,
00128
00129     /* Compound assignment */
00130     TOKEN_PLUSEQ,
00131     TOKEN_MINUSEQ,
00132     TOKEN_STAREQ,
00133     TOKEN_SLASHEQ,
00134     TOKEN_PERCENTEQ,
00135     TOKEN_ANDEQ,
00136     TOKEN_OREQ,
00137     TOKEN_XOREQ,
00138     TOKEN_LSHIFTEQ,
00139     TOKEN_RSHIFTEQ,
00140
00141     /* Member access / varargs */
00142     TOKEN_ARROW,
00143     TOKEN_ELLIPSIS,
00144 };
00145
00154 struct Literal_Token {
00155     enum Token_Kind kind;
00156     const char * const text;
00157 };
00158
00167 struct Location {
00168     size_t line_num;
00169     size_t col;
00170 };
00171
00179 struct String {
00180     size_t length;
00181     size_t capacity;
00182     char* elements;
00183 };
00184
00191 struct Token {
00192     enum Token_Kind kind;
00193     const char *text;
00194     size_t text_len;
00195     struct Location location;
00196 };
00197
00210 struct Tokens {
00211     struct String content;
00212     size_t length;
00213     size_t capacity;
00214     struct Token* elements;
00215 };
00216
00228 struct Lexer {
00229     const char * content;
00230     size_t content_len;
00231     size_t cursor;
00232     size_t line_num;
00233     size_t begining_of_line;
00234 };
00235
00245 static struct Literal_Token literal_tokens[] = {
00246     {.text = "(" , .kind = TOKEN_LPAREN},
00247     {.text = ")" , .kind = TOKEN_RPAREN},
00248     {.text = "[" , .kind = TOKEN_LBRACKET},
00249     {.text = "]" , .kind = TOKEN_RBRACKET},
00250     {.text = "{" , .kind = TOKEN_LBRACE},
00251     {.text = "}" , .kind = TOKEN_RBRACE},
00252     {.text = "#" , .kind = TOKEN_HASH},
00253     {.text = "...", .kind = TOKEN_ELLIPSIS},
00254     {.text = "." , .kind = TOKEN_DOT},
00255     {.text = "," , .kind = TOKEN_COMMA},
00256     {.text = "?" , .kind = TOKEN_QUESTION},
00257     {.text = ":" , .kind = TOKEN_COLON},
00258     {.text = "==" , .kind = TOKEN_EQEQ},
00259     {.text = "!=" , .kind = TOKEN_NE},
00260     {.text = "=" , .kind = TOKEN_EQ},
00261     {.text = "!" , .kind = TOKEN_BANG},
00262     {.text = ";" , .kind = TOKEN_SEMICOLON},
00263     {.text = "\\\" , .kind = TOKEN_BSLASH},
00264     {.text = "->" , .kind = TOKEN_ARROW},
00265     {.text = ">" , .kind = TOKEN_GT},
00266     {.text = ">=" , .kind = TOKEN_GE},
00267     {.text = "<" , .kind = TOKEN_LT},
00268     {.text = "<=" , .kind = TOKEN_LE},
00269     {.text = "<<=" , .kind = TOKEN_LSHIFTEQ},

```



```

00270     {.text = ">=", .kind = TOKEN_RSHIFTEQ},
00271     {.text = "++", .kind = TOKEN_PLUSPLUS},
00272     {.text = "--", .kind = TOKEN_MINUSMINUS},
00273     {.text = "<<", .kind = TOKEN_LSHIFT},
00274     {.text = ">>", .kind = TOKEN_RSHIFT},
00275     {.text = "+=", .kind = TOKEN_PLUSEQ},
00276     {.text = "-=", .kind = TOKEN_MINUSEQ},
00277     {.text = "*=", .kind = TOKEN_STAREQ},
00278     {.text = "/=", .kind = TOKEN_SLASHEQ},
00279     {.text = "%=", .kind = TOKEN_PERCENTEQ},
00280     {.text = "&=", .kind = TOKEN_ANDEQ},
00281     {.text = "|=", .kind = TOKEN_OREQ},
00282     {.text = "^=", .kind = TOKEN_XOREQ},
00283     {.text = "||", .kind = TOKEN_OROR},
00284     {.text = "&&", .kind = TOKEN_ANDAND},
00285     {.text = "|", .kind = TOKEN_BITOR},
00286     {.text = "&", .kind = TOKEN_BITAND},
00287     {.text = "^", .kind = TOKEN_CARET},
00288     {.text = "~", .kind = TOKEN_TILDE},
00289     {.text = "+", .kind = TOKEN_PLUS},
00290     {.text = "-", .kind = TOKEN_MINUS},
00291     {.text = "*", .kind = TOKEN_STAR},
00292     {.text = "/", .kind = TOKEN_SLASH},
00293     {.text = "%", .kind = TOKEN_PERCENT},
00294 };
00295
00296 #define literal_tokens_count (sizeof(literal_tokens) / sizeof(literal_tokens[0]))
00297
00304 static const char * const keywords[] = {
00305     "auto", "break", "case", "char", "const", "continue", "default", "do", "double",
00306     "else", "enum", "extern", "float", "for", "goto", "if", "int", "long", "register",
00307     "return", "short", "signed", "sizeof", "static", "struct", "switch", "typedef",
00308     "union", "unsigned", "void", "volatile", "while", "alignas", "alignof", "and",
00309     "and_eq", "asm", "atomic_cancel", "atomic_commit", "atomic_noexcept", "bitand",
00310     "bitor", "bool", "catch", "char16_t", "char32_t", "char8_t", "class", "co_await",
00311     "co_return", "co_yield", "compl", "concept", "const_cast", "consteval", "constexpr",
00312     "constinit", "decltype", "delete", "dynamic_cast", "explicit", "export", "false",
00313     "friend", "inline", "mutable", "namespace", "new", "noexcept", "not", "not_eq",
00314     "nullptr", "operator", "or", "or_eq", "private", "protected", "public", "reflexpr",
00315     "reinterpret_cast", "requires", "static_assert", "static_cast", "synchronized",
00316     "template", "this", "thread_local", "throw", "true", "try", "typeid", "typename",
00317     "using", "virtual", "wchar_t", "xor", "xor_eq",
00318 };
00319 #define keywords_count (sizeof(keywords) / sizeof(keywords[0]))
00320
00328 #define AL_UNUSED(x) (void)x
00329
00330 bool al_is_identifier(char c);
00331 bool al_is_identifier_start(char c);
00332 struct Tokens al_lex_entire_file(FILE *fp);
00333 struct Lexer al_lexer_alloc(const char *content, size_t len);
00334 char al_lexer_chop_char(struct Lexer *l);
00335 void al_lexer_chop_while(struct Lexer *l, bool (*pred)(char));
00336 struct Token al_lexer_next_token(struct Lexer *l);
00337 bool al_lexer_start_with(struct Lexer *l, const char *prefix);
00338 void al_lexer_trim_left(struct Lexer *l);
00339 char al_lexer_peek(const struct Lexer *l, size_t off);
00340 void al_token_print(struct Token tok);
00341 const char * al_token_kind_name(enum Token_Kind kind);
00342 struct Tokens al_tokens_init(void);
00343 void al_tokens_free(struct Tokens tokens);
00344
00345 #endif /*ALMOG_LEXER_H*/
00346
00347 #ifdef ALMOG_LEXER_IMPLEMENTATION
00348 #undef ALMOG_LEXER_IMPLEMENTATION
00349
00359 bool al_is_identifier(char c)
00360 {
00361     return asm_isalnum(c) || c == '_';
00362 }
00363
00372 bool al_is_identifier_start(char c)
00373 {
00374     return asm_isalpha(c) || c == '_';
00375 }
00376
00377 struct Tokens al_lex_entire_file(FILE *fp)
00378 {
00379     struct Tokens tokens = al_tokens_init();
00380
00381     char temp_str[ASM_MAX_LEN];
00382     int len = 0;
00383     while ((len = asm_get_line(fp, temp_str)) != EOF) {
00384         for (int i = 0; i < len; i++) {
00385             ada_append(char, tokens.content, temp_str[i]);
00386         }

```

```

00387     ada_appand(char, tokens.content, '\n');
00388 }
00389
00390 struct Lexer l = al_lexer_alloc(tokens.content.elements, tokens.content.length);
00391
00392 struct Token t = al_lexer_next_token(&l);
00393 while (t.kind != TOKEN_EOF) {
00394     ada_appand(struct Token, tokens, t);
00395     t = al_lexer_next_token(&l);
00396 }
00397 ada_appand(struct Token, tokens, t);
00398
00399 return tokens;
00400 }
00401
00412 struct Lexer al_lexer_alloc(const char *content, size_t len)
00413 {
00414     struct Lexer l = {0};
00415     l.content = content;
00416     l.content_len = len;
00417     return l;
00418 }
00419
00434 char al_lexer_chop_char(struct Lexer *l)
00435 {
00436     AL_ASSERT(l->cursor < l->content_len);
00437     char c = l->content[l->cursor++];
00438     if (c == '\n') {
00439         l->line_num++;
00440         l->begining_of_line = l->cursor;
00441     }
00442     return c;
00443 }
00444
00454 void al_lexer_chop_while(struct Lexer *l, bool (*pred)(char))
00455 {
00456     while (l->cursor < l->content_len && pred(l->content[l->cursor])) {
00457         al_lexer_chop_char(l);
00458     }
00459 }
00460
00504 struct Token al_lexer_next_token(struct Lexer *l)
00505 {
00506     al_lexer_trim_left(l);
00507
00508     struct Token token = {
00509         .kind = TOKEN_INVALID,
00510         .text = &(l->content[l->cursor]),
00511         .text_len = 0,
00512         .location.line_num = l->line_num+1,
00513         .location.col = l->cursor - l->begining_of_line+1,
00514     };
00515     size_t start = l->cursor;
00516
00517     if (l->cursor >= l->content_len) {
00518         token.kind = TOKEN_EOF;
00519     } else if (l->content[l->cursor] == '#' && token.location.col == 1) {
00520         token.kind = TOKEN_PP_DIRECTIVE;
00521         for (; l->cursor < l->content_len && l->content[l->cursor] != '\n';) {
00522             al_lexer_chop_char(l);
00523         }
00524     } else if (l->cursor < l->content_len) {
00525         al_lexer_chop_char(l);
00526     }
00527     } else if (al_is_identifier_start(l->content[l->cursor])) {
00528         token.kind = TOKEN_IDENTIFIER;
00529         for (; l->cursor < l->content_len && al_is_identifier(l->content[l->cursor]); ) {
00530             al_lexer_chop_char(l);
00531         }
00532         {
00533             size_t ident_len = l->cursor - start;
00534             for (size_t i = 0; i < keywords_count; i++) {
00535                 size_t kw_len = asm_length(keywords[i]);
00536                 if (ident_len == kw_len && asm_strncmp(token.text, keywords[i], kw_len)) {
00537                     token.kind = TOKEN_KEYWORD;
00538                     break;
00539                 }
00540             }
00541         }
00542     } else if (l->content[l->cursor] == '"') {
00543         token.kind = TOKEN_STRING_LITERAL;
00544         al_lexer_chop_char(l);
00545         for (; (l->cursor < l->content_len) && (l->content[l->cursor] != '"') &&
(l->content[l->cursor] != '\n'); ) {
00546             al_lexer_chop_char(l);
00547         }
00548         if ((l->cursor < l->content_len) && (l->content[l->cursor] == '"')) {

```

```

00549         al_lexer_chop_char(l);
00550     }
00551     } else if (l->content[l->cursor] == '\\') {
00552         token.kind = TOKEN_CHAR_LIT;
00553         al_lexer_chop_char(l);
00554         for ( ; (l->cursor < l->content_len) && (l->content[l->cursor] != '\\') &&
(l->content[l->cursor] != '\\n'); ) {
00555             al_lexer_chop_char(l);
00556         }
00557         if ((l->cursor < l->content_len) && (l->content[l->cursor] == '\\')) {
00558             al_lexer_chop_char(l);
00559         }
00560     } else if (al_lexer_start_with(l, "//")) {
00561         token.kind = TOKEN_COMMENT;
00562         for ( ; l->cursor < l->content_len && l->content[l->cursor] != '\\n'; ) {
00563             al_lexer_chop_char(l);
00564         }
00565         if (l->cursor < l->content_len) {
00566             al_lexer_chop_char(l);
00567         }
00568     } else if (al_lexer_start_with(l, "/*")) {
00569         token.kind = TOKEN_COMMENT;
00570         al_lexer_chop_char(l);
00571         al_lexer_chop_char(l);
00572         for ( ; l->cursor < l->content_len; ) {
00573             if ((l->content[l->cursor-1] == '*' && (l->content[l->cursor] == '/')) {
00574                 al_lexer_chop_char(l);
00575                 break;
00576             }
00577             al_lexer_chop_char(l);
00578         }
00579     } else if (asm_isdigit(l->content[l->cursor]) || (l->content[l->cursor] == '.' &&
asm_isdigit(al_lexer_peek(l, 1)))) {
00580         token.kind = TOKEN_INT_LIT_DEC;
00581         bool is_float = false;
00582         bool invalid = false;
00583
00584         if (l->content[l->cursor] == '.') {
00585             token.kind = TOKEN_FLOAT_LIT_DEC;
00586             is_float = true;
00587             al_lexer_chop_char(l);
00588             al_lexer_chop_while(l, asm_isdigit);
00589
00590             /* optional exponent */
00591             if (al_lexer_peek(l, 0) == 'e' || al_lexer_peek(l, 0) == 'E') {
00592                 is_float = true;
00593                 al_lexer_chop_char(l);
00594                 if (al_lexer_peek(l, 0) == '+' || al_lexer_peek(l, 0) == '-') {
00595                     al_lexer_chop_char(l);
00596                 }
00597                 if (!asm_isdigit(al_lexer_peek(l, 0))) {
00598                     invalid = true; /* ".5e" or ".5e+" */
00599                 }
00600                 al_lexer_chop_while(l, asm_isdigit);
00601             }
00602         } else {
00603             /* starts with digit */
00604             if (al_lexer_peek(l, 0) == '0' && (al_lexer_peek(l, 1) == 'x' || al_lexer_peek(l, 1) ==
'X')) {
00605                 token.kind = TOKEN_INT_LIT_HEX;
00606                 al_lexer_chop_char(l);
00607                 al_lexer_chop_char(l);
00608
00609                 size_t mantissa_digits = 0;
00610                 while (asm_isXdigit(al_lexer_peek(l, 0)) || asm_isxdigit(al_lexer_peek(l, 0))) {
00611                     mantissa_digits++;
00612                     al_lexer_chop_char(l);
00613                 }
00614                 if (al_lexer_peek(l, 0) == '.') {
00615                     token.kind = TOKEN_FLOAT_LIT_HEX;
00616                     is_float = true;
00617                     al_lexer_chop_char(l);
00618                     while (asm_isXdigit(al_lexer_peek(l, 0)) || asm_isxdigit(al_lexer_peek(l, 0))) {
00619                         mantissa_digits++;
00620                         al_lexer_chop_char(l);
00621                     }
00622                 }
00623                 if (mantissa_digits == 0) {
00624                     invalid = true; /* "0x" or "0x." */
00625                 }
00626
00627                 /* Hex float requires p/P exponent if it's a float form. */
00628                 if (al_lexer_peek(l, 0) == 'p' || al_lexer_peek(l, 0) == 'P') {
00629                     is_float = true;
00630                     al_lexer_chop_char(l);
00631                     if (al_lexer_peek(l, 0) == '+' || al_lexer_peek(l, 0) == '-') {
00632                         al_lexer_chop_char(l);

```

```

00633         }
00634         if (!asm_isdigit(al_lexer_peek(l, 0))) {
00635             invalid = true; /* "0x1.fp" / "0x1p+" */
00636         }
00637         al_lexer_chop_while(l, asm_isdigit);
00638     } else if (is_float) {
00639         /* Had a '.' in hex mantissa but no p-exponent => invalid hex float */
00640         invalid = true;
00641     }
00642     } else if (al_lexer_peek(l, 0) == '0' && (al_lexer_peek(l, 1) == 'b' || al_lexer_peek(l,
1) == 'B')) {
00643         token.kind = TOKEN_INT_LIT_BIN;
00644         al_lexer_chop_char(l);
00645         al_lexer_chop_char(l);
00646         if (!asm_isbdigit(al_lexer_peek(l, 0))) {
00647             invalid = true; /* "0b" */
00648         }
00649         al_lexer_chop_while(l, asm_isbdigit);
00650     } else if (al_lexer_peek(l, 0) == '0' && (al_lexer_peek(l, 1) == 'o' || al_lexer_peek(l,
1) == 'O')) {
00651         token.kind = TOKEN_INT_LIT_OCT;
00652         al_lexer_chop_char(l);
00653         al_lexer_chop_char(l);
00654         if (!asm_isodigit(al_lexer_peek(l, 0))) {
00655             invalid = true; /* "0o" */
00656         }
00657         while (asm_isodigit(al_lexer_peek(l, 0))) {
00658             al_lexer_chop_char(l);
00659         }
00660     } else {
00661         token.kind = TOKEN_INT_LIT_DEC;
00662         al_lexer_chop_while(l, asm_isdigit);
00663
00664         if (al_lexer_peek(l, 0) == '.') {
00665             token.kind = TOKEN_FLOAT_LIT_DEC;
00666             is_float = true;
00667             al_lexer_chop_char(l);
00668             al_lexer_chop_while(l, asm_isdigit);
00669         }
00670
00671         if (al_lexer_peek(l, 0) == 'e' || al_lexer_peek(l, 0) == 'E') {
00672             is_float = true;
00673             al_lexer_chop_char(l);
00674             if (al_lexer_peek(l, 0) == '+' || al_lexer_peek(l, 0) == '-') {
00675                 al_lexer_chop_char(l);
00676             }
00677             if (!asm_isdigit(al_lexer_peek(l, 0))) {
00678                 invalid = true; /* "1e" / "1e+" */
00679             }
00680             al_lexer_chop_while(l, asm_isdigit);
00681         }
00682     }
00683 }
00684
00685 /* Suffix handling */
00686 if (is_float) {
00687     /* float suffixes: f/F/l/L (accept at most one, but we'll be permissive) */
00688     while (al_lexer_peek(l, 0) == 'f' || al_lexer_peek(l, 0) == 'F' ||
00689           al_lexer_peek(l, 0) == 'l' || al_lexer_peek(l, 0) == 'L') {
00690         al_lexer_chop_char(l);
00691     }
00692 } else {
00693     /* integer suffixes: u/U/l/L/z/Z (permissive) */
00694     while (al_lexer_peek(l, 0) == 'u' || al_lexer_peek(l, 0) == 'U' ||
00695           al_lexer_peek(l, 0) == 'l' || al_lexer_peek(l, 0) == 'L' ||
00696           al_lexer_peek(l, 0) == 'z' || al_lexer_peek(l, 0) == 'Z') {
00697         al_lexer_chop_char(l);
00698     }
00699 }
00700
00701 if (invalid) token.kind = TOKEN_INVALID;
00702 } else {
00703     size_t longest_matching_token = 0;
00704     enum Token_Kind best_kind = TOKEN_INVALID;
00705     for (size_t i = 0; i < literal_tokens_count; i++) {
00706         if (al_lexer_start_with(l, literal_tokens[i].text)) {
00707             /* NOTE: assumes that literal_tokens[i].text does not have any '\n' */
00708             size_t text_len = asm_length(literal_tokens[i].text);
00709             if (text_len > longest_matching_token) {
00710                 longest_matching_token = text_len;
00711                 best_kind = literal_tokens[i].kind;
00712             }
00713         }
00714     }
00715     if (longest_matching_token > 0) {
00716         token.kind = best_kind;
00717         for (size_t i = 0; i < longest_matching_token; i++) {
00718             al_lexer_chop_char(l);

```

```

00718     }
00719     } else {
00720         token.kind = TOKEN_INVALID;
00721         al_lexer_chop_char(l);
00722     }
00723 }
00724
00725 token.text_len = l->cursor - start;
00726
00727 return token;
00728 }
00729
00739 bool al_lexer_start_with(struct Lexer *l, const char *prefix)
00740 {
00741     size_t prefix_len = asm_length(prefix);
00742     if (prefix_len == 0) {
00743         return true;
00744     }
00745     if (l->cursor + prefix_len > l->content_len) {
00746         return false;
00747     }
00748     for (size_t i = 0; i < prefix_len; i++) {
00749         if (prefix[i] != l->content[l->cursor + i]) {
00750             return false;
00751         }
00752     }
00753     return true;
00754 }
00755
00764 void al_lexer_trim_left(struct Lexer *l)
00765 {
00766     for (; l->cursor < l->content_len; ) {
00767         if (!asm_isspace(l->content[l->cursor])) {
00768             break;
00769         }
00770         al_lexer_chop_char(l);
00771     }
00772 }
00773
00781 char al_lexer_peek(const struct Lexer *l, size_t off)
00782 {
00783     size_t i = l->cursor + off;
00784     if (i >= l->content_len) return '\0';
00785     return l->content[i];
00786 }
00787
00799 void al_token_print(struct Token tok)
00800 {
00801     printf("%4zu:%3zu:(%-19s) -> \"%s.%s\"\\n", tok.location.line_num, tok.location.col,
00802         al_token_kind_name(tok.kind), (int)tok.text_len, tok.text);
00803 }
00804
00813 const char *al_token_kind_name(enum Token_Kind kind)
00814 {
00815     switch (kind) {
00816         case TOKEN_EOF:
00817             return ("TOKEN_EOF");
00818         case TOKEN_INVALID:
00819             return ("TOKEN_INVALID");
00820         case TOKEN_PP_DIRECTIVE:
00821             return ("TOKEN_PP_DIRECTIVE");
00822         case TOKEN_IDENTIFIER:
00823             return ("TOKEN_IDENTIFIER");
00824         case TOKEN_LPAREN:
00825             return ("TOKEN_LPAREN");
00826         case TOKEN_RPAREN:
00827             return ("TOKEN_RPAREN");
00828         case TOKEN_LBRACKET:
00829             return ("TOKEN_LBRACKET");
00830         case TOKEN_RBRACKET:
00831             return ("TOKEN_RBRACKET");
00832         case TOKEN_LBRACE:
00833             return ("TOKEN_LBRACE");
00834         case TOKEN_RBRACE:
00835             return ("TOKEN_RBRACE");
00836         case TOKEN_DOT:
00837             return ("TOKEN_DOT");
00838         case TOKEN_COMMA:
00839             return ("TOKEN_COMMA");
00840         case TOKEN_SEMICOLON:
00841             return ("TOKEN_SEMICOLON");
00842         case TOKEN_BSLASH:
00843             return ("TOKEN_BSLASH");
00844         case TOKEN_QUESTION:
00845             return ("TOKEN_QUESTION");
00846         case TOKEN_COLON:
00847             return ("TOKEN_COLON");

```

```
00848     case TOKEN_LT:
00849         return ("TOKEN_LT");
00850     case TOKEN_GT:
00851         return ("TOKEN_GT");
00852     case TOKEN_GE:
00853         return ("TOKEN_GE");
00854     case TOKEN_LE:
00855         return ("TOKEN_LE");
00856     case TOKEN_KEYWORD:
00857         return ("TOKEN_KEYWORD");
00858     case TOKEN_INT_LIT_BIN:
00859         return ("TOKEN_INT_LIT_BIN");
00860     case TOKEN_INT_LIT_OCT:
00861         return ("TOKEN_INT_LIT_OCT");
00862     case TOKEN_INT_LIT_DEC:
00863         return ("TOKEN_INT_LIT_DEC");
00864     case TOKEN_INT_LIT_HEX:
00865         return ("TOKEN_INT_LIT_HEX");
00866     case TOKEN_FLOAT_LIT_DEC:
00867         return ("TOKEN_FLOAT_LIT_DEC");
00868     case TOKEN_FLOAT_LIT_HEX:
00869         return ("TOKEN_FLOAT_LIT_HEX");
00870     case TOKEN_COMMENT:
00871         return ("TOKEN_COMMENT");
00872     case TOKEN_STRING_LIT:
00873         return ("TOKEN_STRING_LIT");
00874     case TOKEN_CHAR_LIT:
00875         return ("TOKEN_CHAR_LIT");
00876     case TOKEN_EQ:
00877         return ("TOKEN_EQ");
00878     case TOKEN_EQEQ:
00879         return ("TOKEN_EQEQ");
00880     case TOKEN_NE:
00881         return ("TOKEN_NE");
00882     case TOKEN_BANG:
00883         return ("TOKEN_BANG");
00884     case TOKEN_BITAND:
00885         return ("TOKEN_BITAND");
00886     case TOKEN_ANDAND:
00887         return ("TOKEN_ANDAND");
00888     case TOKEN_BITOR:
00889         return ("TOKEN_BITOR");
00890     case TOKEN_OROR:
00891         return ("TOKEN_OROR");
00892     case TOKEN_CARET:
00893         return ("TOKEN_CARET");
00894     case TOKEN_TILDE:
00895         return ("TOKEN_TILDE");
00896     case TOKEN_PLUSPLUS:
00897         return ("TOKEN_PLUSPLUS");
00898     case TOKEN_MINUSMINUS:
00899         return ("TOKEN_MINUSMINUS");
00900     case TOKEN_LSHIFT:
00901         return ("TOKEN_LSHIFT");
00902     case TOKEN_RSHIFT:
00903         return ("TOKEN_RSHIFT");
00904     case TOKEN_PLUS:
00905         return ("TOKEN_PLUS");
00906     case TOKEN_MINUS:
00907         return ("TOKEN_MINUS");
00908     case TOKEN_STAR:
00909         return ("TOKEN_STAR");
00910     case TOKEN_SLASH:
00911         return ("TOKEN_SLASH");
00912     case TOKEN_HASH:
00913         return ("TOKEN_HASH");
00914     case TOKEN_PERCENT:
00915         return ("TOKEN_PERCENT");
00916     case TOKEN_PLUSEQ:
00917         return ("TOKEN_PLUSEQ");
00918     case TOKEN_MINUSEQ:
00919         return ("TOKEN_MINUSEQ");
00920     case TOKEN_STAREQ:
00921         return ("TOKEN_STAREQ");
00922     case TOKEN_SLASHEQ:
00923         return ("TOKEN_SLASHEQ");
00924     case TOKEN_PERCENTEQ:
00925         return ("TOKEN_PERCENTEQ");
00926     case TOKEN_ANDEQ:
00927         return ("TOKEN_ANDEQ");
00928     case TOKEN_OREQ:
00929         return ("TOKEN_OREQ");
00930     case TOKEN_XOREQ:
00931         return ("TOKEN_XOREQ");
00932     case TOKEN_LSHIFTEQ:
00933         return ("TOKEN_LSHIFTEQ");
00934     case TOKEN_RSHIFTEQ:
```

```

00935         return ("TOKEN_RSHIFTEQ");
00936     case TOKEN_ARROW:
00937         return ("TOKEN_ARROW");
00938     case TOKEN_ELLIPSIS:
00939         return ("TOKEN_ELLIPSIS");
00940     default:
00941         AL_ASSERT(0 && "Unknown kind");
00942     }
00943     return NULL;
00944 }
00945
00946 struct Tokens al_tokens_init(void)
00947 {
00948     struct Tokens tokens = {0};
00949     ada_init_array(struct Token, tokens);
00950     ada_init_array(char, tokens.content);
00951
00952     return tokens;
00953 }
00954
00955 void al_tokens_free(struct Tokens tokens)
00956 {
00957     AL_FREE(tokens.content.elements);
00958     AL_FREE(tokens.elements);
00959 }
00960
00961 #endif /*ALMOG_LEXER_IMPLEMENTATION*/
00962

```

4.7 Almog_String_Manipulation.h File Reference

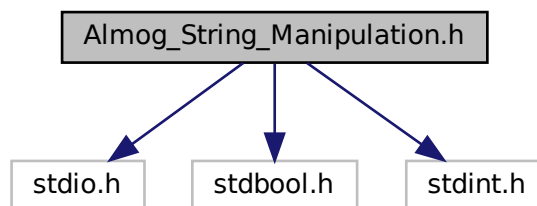
Lightweight string and line manipulation helpers.

```

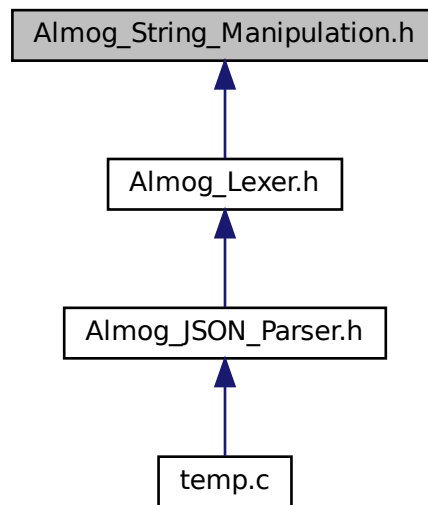
#include <stdio.h>
#include <stdbool.h>
#include <stdint.h>

```

Include dependency graph for Almog_String_Manipulation.h:



This graph shows which files directly or indirectly include this file:



Macros

- `#define ASM_MAX_LEN (int)1e3`
Maximum number of characters processed in some string operations.
- `#define asm_dprintSTRING(expr) printf(#expr " = %s\n", expr)`
Debug-print a C string expression as "expr = value\n".
- `#define asm_dprintCHAR(expr) printf(#expr " = %c\n", expr)`
Debug-print a character expression as "expr = c\n".
- `#define asm_dprintINT(expr) printf(#expr " = %d\n", expr)`
Debug-print an integer expression as "expr = n\n".
- `#define asm_dprintFLOAT(expr) printf(#expr " = %g\n", expr)`
Debug-print a float expression as "expr = n\n".
- `#define asm_dprintDOUBLE(expr) printf(#expr " = %g\n", expr)`
Debug-print a double expression as "expr = n\n".
- `#define asm_dprintSIZE_T(expr) printf(#expr " = %zu\n", expr)`
Debug-print a size_t expression as "expr = n\n".
- `#define asm_dprintERROR(fmt, ...)`
- `#define asm_min(a, b) ((a) < (b) ? (a) : (b))`
Return the smaller of two values (macro).
- `#define asm_max(a, b) ((a) > (b) ? (a) : (b))`
Return the larger of two values (macro).

Functions

- bool [asm_check_char_belong_to_base](#) (const char c, const size_t base)
Check if a character is a valid digit in a given base.
- void [asm_copy_array_by_indexes](#) (char *const target, const int start, const int end, const char *const src)
Copy a substring from `src` into `target` by indices and null-terminate.
- int [asm_get_char_value_in_base](#) (const char c, const size_t base)
*Convert a digit character to its numeric value in base-*N*.*
- int [asm_get_line](#) (FILE *fp, char *const dst)
Read a single line from a stream into a buffer.
- int [asm_get_next_token_from_str](#) (char *const dst, const char *const src, const char delimiter)
Copy characters from the start of a string into a token buffer.
- int [asm_get_token_and_cut](#) (char *const dst, char *src, const char delimiter, const bool leave_delimiter)
Extract the next token into `dst` and remove the corresponding prefix from `src`.
- bool [asm_isalnum](#) (char c)
Test for an alphanumeric character (ASCII).
- bool [asm_isalpha](#) (char c)
Test for an alphabetic character (ASCII).
- bool [asm_isbdigit](#) (const char c)
Test for a binary digit (ASCII).
- bool [asm_iscntrl](#) (char c)
Test for a control character (ASCII).
- bool [asm_isdigit](#) (char c)
Test for a decimal digit (ASCII).
- bool [asm_isgraph](#) (char c)
Test for any printable character except space (ASCII).
- bool [asm_islower](#) (char c)
Test for a lowercase letter (ASCII).
- bool [asm_isodigit](#) (const char c)
Test for an octal digit (ASCII).
- bool [asm_isprint](#) (char c)
Test for any printable character including space (ASCII).
- bool [asm_ispunct](#) (char c)
Test for a punctuation character (ASCII).
- bool [asm_isspace](#) (char c)
Test for a whitespace character (ASCII).
- bool [asm_isupper](#) (char c)
Test for an uppercase letter (ASCII).
- bool [asm_isxdigit](#) (char c)
Test for a hexadecimal digit (lowercase or decimal).
- bool [asm_isXdigit](#) (char c)
Test for a hexadecimal digit (uppercase or decimal).
- size_t [asm_length](#) (const char *const str)
Compute the length of a null-terminated C string.
- void * [asm_memset](#) (void *const des, const unsigned char value, const size_t n)
Set a block of memory to a repeated byte value.
- void [asm_pad_left](#) (char *const s, const size_t padding, const char pad)
Left-pad a string in-place.
- void [asm_print_many_times](#) (const char *const str, const size_t n)
*Print a string *n* times, then print a newline.*
- void [asm_remove_char_from_string](#) (char *const s, const size_t index)

- Remove a single character from a string by index.*

 - void `asm_shift_left` (char *const s, const size_t shift)

Shift a string left in-place by `shift` characters.
- int `asm_str_in_str` (const char *const src, const char *const word_to_search)

Count occurrences of a substring within a string.
- double `asm_str2double` (const char *const s, const char **const end, const size_t base)

Convert a string to double in the given base with exponent support.
- float `asm_str2float` (const char *const s, const char **const end, const size_t base)

Convert a string to float in the given base with exponent support.
- int `asm_str2int` (const char *const s, const char **const end, const size_t base)

Convert a string to int in the given base.
- size_t `asm_str2size_t` (const char *const s, const char **const end, const size_t base)

Convert a string to size_t in the given base.
- void `asm_strip_whitespace` (char *const s)

Remove all ASCII whitespace characters from a string in-place.
- bool `asm_str_is_whitespace` (const char *const s)

Check whether a string contains only ASCII whitespace characters.
- int `asm_strncat` (char *const s1, const char *const s2, const size_t N)

Append up to N characters from s2 to the end of s1.
- int `asm_strncmp` (const char *s1, const char *s2, const size_t N)

Compare up to N characters for equality (boolean result).
- int `asm_strncpy` (char *const s1, const char *const s2, const size_t N)

Copy up to N characters from s2 into s1 (non-standard).
- void `asm_tolower` (char *const s)

Convert all ASCII letters in a string to lowercase in-place.
- void `asm_toupper` (char *const s)

Convert all ASCII letters in a string to uppercase in-place.
- void `asm_trim_left_whitespace` (char *const s)

Remove leading ASCII whitespace from a string in-place.

4.7.1 Detailed Description

Lightweight string and line manipulation helpers.

This single-header module provides small utilities for working with C strings:

- Reading a single line from a FILE stream
- Measuring string length
- Extracting the next token from a string using a delimiter (does not skip whitespace)
- Cutting the extracted token (and leading whitespace) from the source buffer
- Copying a substring by indices
- Counting occurrences of a substring
- A boolean-style strncmp (returns 1 on equality, 0 otherwise)
- ASCII-only character classification helpers (isalnum, isalpha, ...)
- ASCII case conversion (toupper / tolower)
- In-place whitespace stripping and left padding

- Base-N string-to-number conversion for int, size_t, float, and double

Usage

- In exactly one translation unit, define `ALMOG_STRING_MANIPULATION_IMPLEMENTATION` before including this header to compile the implementation.
- In all other files, include the header without the macro to get declarations only.

Notes and limitations

- All destination buffers must be large enough; functions do not grow or allocate buffers.
- `asm_get_line` and `asm_length` enforce `ASM_MAX_LEN` characters (not counting the terminating `'\0'`). Longer lines cause an early return with an error message.
- `asm_strncmp` differs from the standard C `strncmp`: this version returns 1 if equal and 0 otherwise.
- Character classification and case-conversion helpers are ASCII-only and not locale aware.

Definition in file [Almog_String_Manipulation.h](#).

4.7.2 Macro Definition Documentation

4.7.2.1 asm_dprintCHAR

```
#define asm_dprintCHAR(  
    expr ) printf(#expr " = %c\n", expr)
```

Debug-print a character expression as "expr = c\n".

Parameters

<i>expr</i>	An expression that yields a character (or an int promoted from a character). The expression is evaluated exactly once.
-------------	--

Definition at line 83 of file [Almog_String_Manipulation.h](#).

4.7.2.2 asm_dprintDOUBLE

```
#define asm_dprintDOUBLE(  
    expr ) printf(#expr " = %g\n", expr)
```

Debug-print a double expression as "expr = n\n".

Parameters

<i>expr</i>	An expression that yields a double. The expression is evaluated exactly once.
-------------	---

Definition at line 110 of file [Almog_String_Manipulation.h](#).

4.7.2.3 asm_dprintERROR

```
#define asm_dprintERROR(  
    fmt,  
    ... )
```

Value:

```
fprintf(stderr, "\n%s:%d:\n[Error] in function '%s':\n  
fmt "\n\n", __FILE__, __LINE__, __func__, __VA_ARGS__)" \
```

Definition at line 121 of file [Almog_String_Manipulation.h](#).

4.7.2.4 asm_dprintFLOAT

```
#define asm_dprintFLOAT(  
    expr ) printf(#expr " = %g\n", expr)
```

Debug-print a float expression as "expr = n\n".

Parameters

<i>expr</i>	An expression that yields a float. The expression is evaluated exactly once.
-------------	--

Definition at line 101 of file [Almog_String_Manipulation.h](#).

4.7.2.5 asm_dprintINT

```
#define asm_dprintINT(  
    expr ) printf(#expr " = %d\n", expr)
```

Debug-print an integer expression as "expr = n\n".

Parameters

<i>expr</i>	An expression that yields an int. The expression is evaluated exactly once.
-------------	---

Definition at line 92 of file [Almog_String_Manipulation.h](#).

4.7.2.6 asm_dprintSIZE_T

```
#define asm_dprintSIZE_T(  
    expr ) printf(#expr " = %zu\n", expr)
```

Debug-print a size_t expression as "expr = n\n".

Parameters

<i>expr</i>	An expression that yields a size_t. The expression is evaluated exactly once.
-------------	---

Definition at line 119 of file [Almog_String_Manipulation.h](#).

4.7.2.7 asm_dprintSTRING

```
#define asm_dprintSTRING(  
    expr ) printf(#expr " = %s\n", expr)
```

Debug-print a C string expression as "expr = value\n".

Parameters

<i>expr</i>	An expression that yields a pointer to char (const or non-const). The expression is evaluated exactly once.
-------------	---

Definition at line 74 of file [Almog_String_Manipulation.h](#).

4.7.2.8 asm_max

```
#define asm_max(  
    a,  
    b ) ((a) > (b) ? (a) : (b))
```

Return the larger of two values (macro).

Parameters

<i>a</i>	First value.
<i>b</i>	Second value.

Returns

The larger of *a* and *b*.

Note

Each parameter may be evaluated more than once. Do not pass expressions with side effects (e.g., ++i, function calls with state).

Definition at line 149 of file [Almog_String_Manipulation.h](#).

4.7.2.9 ASM_MAX_LEN

```
#define ASM_MAX_LEN (int)1e3
```

Maximum number of characters processed in some string operations.

This constant limits:

- The number of characters read by `asm_get_line` from a stream (excluding the terminating null byte).
- The maximum number of characters inspected by `asm_length`.

If `asm_get_line` reads `ASM_MAX_LEN` characters without encountering '`'` or EOF, it prints an error to `stderr` and returns -1. In that error case, the buffer is truncated and null-terminated by overwriting the last stored character (so the resulting string length is `ASM_MAX_LEN - 1`).

Definition at line 64 of file [Almog_String_Manipulation.h](#).

4.7.2.10 asm_min

```
#define asm_min(  
    a,  
    b ) ((a) < (b) ? (a) : (b))
```

Return the smaller of two values (macro).

Parameters

<i>a</i>	First value.
<i>b</i>	Second value.

Returns

The smaller of `a` and `b`.

Note

Each parameter may be evaluated more than once. Do not pass expressions with side effects (e.g., ++i, function calls with state).

Definition at line 136 of file [Almog_String_Manipulation.h](#).

4.7.3 Function Documentation

4.7.3.1 `asm_check_char_belong_to_base()`

```
bool asm_check_char_belong_to_base (
    const char c,
    const size_t base )
```

Check if a character is a valid digit in a given base.

Parameters

<i>c</i>	Character to test (e.g., '0'-'9', 'a'-'z', 'A'-'Z').
<i>base</i>	Numeric base in the range [2, 36].

Returns

true if *c* is a valid digit for *base*, false otherwise.

Note

If *base* is outside [2, 36], an error is printed to `stderr` and false is returned.

Definition at line 206 of file [Almog_String_Manipulation.h](#).

References [asm_dprintERROR](#), and [asm_isdigit\(\)](#).

Referenced by [asm_get_char_value_in_base\(\)](#), [asm_str2double\(\)](#), [asm_str2float\(\)](#), [asm_str2int\(\)](#), and [asm_str2size_t\(\)](#).

4.7.3.2 `asm_copy_array_by_indexes()`

```
void asm_copy_array_by_indexes (
    char *const target,
    const int start,
    const int end,
    const char *const src )
```

Copy a substring from *src* into *target* by indices and null-terminate.

Copies characters with indices *i* = *start*, *start* + 1, ..., *end* from *src* into *target* (note: *end* is inclusive in this implementation), then ensures *target* is null-terminated.

Parameters

<i>target</i>	Destination buffer. Must be large enough to hold (end - start + 1) characters plus the null terminator.
<i>start</i>	Inclusive start index within <i>src</i> (0-based).
<i>end</i>	Inclusive end index within <i>src</i> (must satisfy <i>end</i> >= <i>start</i>).
<i>src</i>	Source string buffer.

Warning

No bounds checking is performed. The caller must ensure valid indices and sufficient target capacity.

Definition at line 241 of file [Almog_String_Manipulation.h](#).

4.7.3.3 asm_get_char_value_in_base()

```
int asm_get_char_value_in_base (
    const char c,
    const size_t base )
```

Convert a digit character to its numeric value in base-N.

Parameters

<i>c</i>	Digit character ('0'–'9', 'a'–'z', 'A'–'Z').
<i>base</i>	Numeric base in the range [2, 36] (used for validation).

Returns

The numeric value of *c* in the range [0, 35].

Note

This function assumes *c* is a valid digit character. Call [asm_check_char_belong_to_base\(\)](#) first if validation is needed.

Definition at line 264 of file [Almog_String_Manipulation.h](#).

References [asm_check_char_belong_to_base\(\)](#), [asm_isdigit\(\)](#), and [asm_isupper\(\)](#).

Referenced by [asm_str2double\(\)](#), [asm_str2float\(\)](#), [asm_str2int\(\)](#), and [asm_str2size_t\(\)](#).

4.7.3.4 asm_get_line()

```
int asm_get_line (
    FILE * fp,
    char *const dst )
```

Read a single line from a stream into a buffer.

Reads characters from the FILE stream until a newline ('
') or EOF is encountered. The newline, if present, is not copied. The result is always null-terminated on normal (non-error) completion.

Parameters

<i>fp</i>	Input stream (must be non-NULL).
<i>dst</i>	Destination buffer. Must have capacity of at least ASM_MAX_LEN + 1 bytes.

Returns

Number of characters stored in *dst* (excluding the terminating null byte).

Return values

-1	EOF was encountered before any character was read, or the line exceeded ASM_MAX_LEN characters (error).
----	---

Note

If the line reaches ASM_MAX_LEN characters before a newline or EOF is seen, the function prints an error message to stderr and returns -1. In that case, *dst* is truncated and null-terminated by overwriting the last stored character.

An empty line (just '
' returns 0 (not -1).

Definition at line 297 of file [Almog_String_Manipulation.h](#).

References [asm_dprintERROR](#), and [ASM_MAX_LEN](#).

4.7.3.5 asm_get_next_token_from_str()

```
int asm_get_next_token_from_str (
    char *const dst,
    const char *const src,
    const char delimiter )
```

Copy characters from the start of a string into a token buffer.

Copies characters from *src* into *dst* until one of the following is encountered in *src*:

- the delimiter character,
- or the string terminator ('\0').

The delimiter (if present) is not copied into *dst*. The resulting token in *dst* is always null-terminated.

Parameters

<i>dst</i>	Destination buffer for the extracted token. Must be large enough to hold the token plus the null terminator.
<i>src</i>	Source C string to parse (not modified by this function).
<i>delimiter</i>	Delimiter character to stop at.

Returns

The number of characters copied into `dst` (excluding the null terminator). This is also the index in `src` of the delimiter or `'\0'` that stopped the copy.

Note

This function does not skip leading whitespace and does not treat newline (`'\n'`) specially; newlines are copied like any other character.

If `src` starts with `delimiter` or `'\0'`, an empty token is produced (`dst` becomes `""`), and 0 is returned.

Definition at line 344 of file [Almog_String_Manipulation.h](#).

Referenced by [asm_get_token_and_cut\(\)](#).

4.7.3.6 `asm_get_token_and_cut()`

```
int asm_get_token_and_cut (
    char *const dst,
    char * src,
    const char delimiter,
    const bool leave_delimiter )
```

Extract the next token into `dst` and remove the corresponding prefix from `src`.

Calls `asm_get_next_token_from_str(dst, src, delimiter)` to extract a token from the beginning of `src` into `dst`. Then modifies `src` in-place by left-shifting it.

If `leave_delimiter` is true, `src` is left-shifted by the value returned from [asm_get_next_token_from_str\(\)](#) (i.e., the delimiter—if present—remains as the first character in the updated `src`).

If `leave_delimiter` is false, `src` is left-shifted by that return value plus one (intended to also remove the delimiter).

Parameters

<i>dst</i>	Destination buffer for the extracted token (must be large enough for the token plus the null terminator).
<i>src</i>	Source buffer, modified in-place by this function.
<i>delimiter</i>	Delimiter character used to stop token extraction.
<i>leave_delimiter</i>	If true, do not remove the delimiter from <code>src</code> ; if false, remove one additional character after the token.

Returns

1 if [asm_get_next_token_from_str\(\)](#) returned a non-zero value, otherwise 0.

Note

This function always calls [asm_shift_left\(\)](#) even when the returned value from [asm_get_next_token_from_str\(\)](#) is 0. In particular, when `leave_delimiter` is false and the returned value is 0, `src` will be left-shifted by 1.

Definition at line 387 of file [Almog_String_Manipulation.h](#).

References [asm_get_next_token_from_str\(\)](#), and [asm_shift_left\(\)](#).

4.7.3.7 asm_isalnum()

```
bool asm_isalnum (
    char c )
```

Test for an alphanumeric character (ASCII).

Parameters

<code>c</code>	Character to test.
----------------	--------------------

Returns

true if `c` is '0'-'9', 'A'-'Z', or 'a'-'z'; false otherwise.

Definition at line 408 of file [Almog_String_Manipulation.h](#).

References [asm_isalpha\(\)](#), and [asm_isdigit\(\)](#).

Referenced by [al_is_identifier\(\)](#).

4.7.3.8 asm_isalpha()

```
bool asm_isalpha (
    char c )
```

Test for an alphabetic character (ASCII).

Parameters

<code>c</code>	Character to test.
----------------	--------------------

Returns

true if `c` is 'A'-'Z' or 'a'-'z'; false otherwise.

Definition at line 419 of file [Almog_String_Manipulation.h](#).

References [asm_islower\(\)](#), and [asm_isupper\(\)](#).

Referenced by [al_is_identifier_start\(\)](#), and [asm_isalnum\(\)](#).

4.7.3.9 [asm_isbdigit\(\)](#)

```
bool asm_isbdigit (  
    const char c )
```

Test for a binary digit (ASCII).

Parameters

<i>c</i>	Character to test.
----------	--------------------

Returns

true if *c* is '0' or '1'; false otherwise.

Definition at line 430 of file [Almog_String_Manipulation.h](#).

4.7.3.10 [asm_iscntrl\(\)](#)

```
bool asm_iscntrl (  
    char c )
```

Test for a control character (ASCII).

Parameters

<i>c</i>	Character to test.
----------	--------------------

Returns

true if *c* is in the range [0, 31] or 127; false otherwise.

Definition at line 445 of file [Almog_String_Manipulation.h](#).

4.7.3.11 [asm_isdigit\(\)](#)

```
bool asm_isdigit (  
    char c )
```

Test for a decimal digit (ASCII).

Parameters

<code>c</code>	Character to test.
----------------	--------------------

Returns

true if `c` is '0'–'9'; false otherwise.

Definition at line 460 of file [Almog_String_Manipulation.h](#).

Referenced by [asm_check_char_belong_to_base\(\)](#), [asm_get_char_value_in_base\(\)](#), [asm_isalnum\(\)](#), [asm_isxdigit\(\)](#), and [asm_isXdigit\(\)](#).

4.7.3.12 asm_isgraph()

```
bool asm_isgraph (  
    char c )
```

Test for any printable character except space (ASCII).

Parameters

<code>c</code>	Character to test.
----------------	--------------------

Returns

true if `c` is in the range [33, 126]; false otherwise.

Definition at line 475 of file [Almog_String_Manipulation.h](#).

Referenced by [asm_isprint\(\)](#).

4.7.3.13 asm_islower()

```
bool asm_islower (  
    char c )
```

Test for a lowercase letter (ASCII).

Parameters

<code>c</code>	Character to test.
----------------	--------------------

Returns

true if `c` is 'a'-'z'; false otherwise.

Definition at line 490 of file [Almog_String_Manipulation.h](#).

Referenced by [asm_isalpha\(\)](#), and [asm_toupper\(\)](#).

4.7.3.14 asm_isodigit()

```
bool asm_isodigit (  
    const char c )
```

Test for an octal digit (ASCII).

Parameters

<code>c</code>	Character to test.
----------------	--------------------

Returns

true if `c` is '0'-'7'; false otherwise.

Definition at line 505 of file [Almog_String_Manipulation.h](#).

4.7.3.15 asm_isprint()

```
bool asm_isprint (  
    char c )
```

Test for any printable character including space (ASCII).

Parameters

<code>c</code>	Character to test.
----------------	--------------------

Returns

true if `c` is space (' ') or `asm_isgraph(c)` is true; false otherwise.

Definition at line 521 of file [Almog_String_Manipulation.h](#).

References [asm_isgraph\(\)](#).

4.7.3.16 asm_ispunct()

```
bool asm_ispunct (
    char c )
```

Test for a punctuation character (ASCII).

Parameters

<i>c</i>	Character to test.
----------	--------------------

Returns

true if *c* is a printable, non-alphanumeric, non-space character; false otherwise.

Definition at line 533 of file [Almog_String_Manipulation.h](#).

4.7.3.17 asm_isspace()

```
bool asm_isspace (
    char c )
```

Test for a whitespace character (ASCII).

Parameters

<i>c</i>	Character to test.
----------	--------------------

Returns

true if *c* is one of ' ',
'\t', '\v', '\f', or '\r'; false otherwise.

Definition at line 549 of file [Almog_String_Manipulation.h](#).

Referenced by [al_lexer_trim_left\(\)](#), [asm_str2double\(\)](#), [asm_str2float\(\)](#), [asm_str2int\(\)](#), [asm_str2size_t\(\)](#), [asm_str_is_whitespace\(\)](#), [asm_strip_whitespace\(\)](#), and [asm_trim_left_whitespace\(\)](#).

4.7.3.18 asm_isupper()

```
bool asm_isupper (
    char c )
```

Test for an uppercase letter (ASCII).

Parameters

<code>c</code>	Character to test.
----------------	--------------------

Returns

true if `c` is 'A'–'Z'; false otherwise.

Definition at line 565 of file [Almog_String_Manipulation.h](#).

Referenced by [asm_get_char_value_in_base\(\)](#), [asm_isalpha\(\)](#), and [asm_tolower\(\)](#).

4.7.3.19 asm_isxdigit()

```
bool asm_isxdigit (  
    char c )
```

Test for a hexadecimal digit (lowercase or decimal).

Parameters

<code>c</code>	Character to test.
----------------	--------------------

Returns

true if `c` is '0'–'9' or 'a'–'f'; false otherwise.

Definition at line 580 of file [Almog_String_Manipulation.h](#).

References [asm_isdigit\(\)](#).

4.7.3.20 asm_isXdigit()

```
bool asm_isXdigit (  
    char c )
```

Test for a hexadecimal digit (uppercase or decimal).

Parameters

<code>c</code>	Character to test.
----------------	--------------------

Returns

true if `c` is '0'–'9' or 'A'–'F'; false otherwise.

Definition at line 595 of file [Almog_String_Manipulation.h](#).

References [asm_isdigit\(\)](#).

4.7.3.21 asm_length()

```
size_t asm_length (
    const char *const str )
```

Compute the length of a null-terminated C string.

Parameters

<i>str</i>	Null-terminated string (must be non-NULL).
------------	--

Returns

The number of characters before the terminating null byte.

Note

If more than `ASM_MAX_LEN` characters are scanned without encountering a null terminator, an error is printed to `stderr` and **SIZE_MAX** is returned.

Definition at line 614 of file [Almog_String_Manipulation.h](#).

References [asm_dprintERROR](#), and [ASM_MAX_LEN](#).

Referenced by [al_lexer_start_with\(\)](#), [asm_pad_left\(\)](#), [asm_remove_char_from_string\(\)](#), [asm_shift_left\(\)](#), [asm_str_in_str\(\)](#), [asm_str_is_whitespace\(\)](#), [asm_strip_whitespace\(\)](#), [asm_strncat\(\)](#), [asm_strncpy\(\)](#), [asm_tolower\(\)](#), [asm_toupper\(\)](#), and [asm_trim_left_whitespace\(\)](#).

4.7.3.22 asm_memset()

```
void * asm_memset (
    void *const des,
    const unsigned char value,
    const size_t n )
```

Set a block of memory to a repeated byte value.

Writes `value` into each of the first `n` bytes of the memory region pointed to by `des`. This function mirrors the behavior of the standard C `memset()`, but implements it using a simple byte-wise loop.

Parameters

<i>des</i>	Destination memory block to modify. Must point to a valid buffer of at least <i>n</i> bytes.
<i>value</i>	Unsigned byte value to store repeatedly.
<i>n</i>	Number of bytes to set.

Returns

The original pointer *des*.

Note

This implementation performs no optimizations (such as word-sized writes); the memory block is filled one byte at a time.

Behavior is undefined if *des* overlaps with invalid or non-writable memory.

Definition at line 649 of file [Almog_String_Manipulation.h](#).

4.7.3.23 asm_pad_left()

```
void asm_pad_left (
    char *const s,
    const size_t padding,
    const char pad )
```

Left-pad a string in-place.

Shifts the contents of *s* to the right by *padding* positions and fills the vacated leading positions with *pad*.

Parameters

<i>s</i>	String to pad. Modified in-place.
<i>padding</i>	Number of leading spaces to insert.
<i>pad</i>	The padding character to insert.

Warning

The buffer backing *s* must have enough capacity for the original string length plus *padding* and the terminating null byte. No bounds checking is performed.

Definition at line 672 of file [Almog_String_Manipulation.h](#).

References [asm_length\(\)](#).

4.7.3.24 asm_print_many_times()

```
void asm_print_many_times (
    const char *const str,
    const size_t n )
```

Print a string *n* times, then print a newline.

Parameters

<i>str</i>	String to print (as-is with <code>printf("%s", ...)</code>).
<i>n</i>	Number of times to print <i>str</i> .

Definition at line 689 of file [Almog_String_Manipulation.h](#).

4.7.3.25 asm_remove_char_from_string()

```
void asm_remove_char_from_string (
    char *const s,
    const size_t index )
```

Remove a single character from a string by index.

Deletes the character at position *index* from *s* by shifting subsequent characters one position to the left.

Parameters

<i>s</i>	String to modify in-place. Must be null-terminated.
<i>index</i>	Zero-based index of the character to remove.

Note

If *index* is out of range, an error is printed to `stderr` and the string is left unchanged.

Definition at line 709 of file [Almog_String_Manipulation.h](#).

References [asm_dprintERROR](#), and [asm_length\(\)](#).

Referenced by [asm_strip_whitespace\(\)](#).

4.7.3.26 asm_shift_left()

```
void asm_shift_left (
    char *const s,
    const size_t shift )
```

Shift a string left in-place by *shift* characters.

Removes the first *shift* characters from *s* by moving the remaining characters to the front. The resulting string is always null-terminated.

Parameters

<i>s</i>	String to modify in-place. Must be null-terminated.
<i>shift</i>	Number of characters to remove from the front.

Note

If `shift` is 0, `s` is unchanged.

If `shift` is greater than or equal to the string length, `s` becomes the empty string.

Definition at line 738 of file [Almog_String_Manipulation.h](#).

References [asm_length\(\)](#).

Referenced by [asm_get_token_and_cut\(\)](#), and [asm_trim_left_whitespace\(\)](#).

4.7.3.27 asm_str2double()

```
double asm_str2double (
    const char *const s,
    const char **const end,
    const size_t base )
```

Convert a string to double in the given base with exponent support.

Parses an optional sign, then a sequence of base-N digits, optionally a fractional part separated by a '.' character, and optionally an exponent part indicated by 'e' or 'E' followed by an optional sign and decimal digits.

Parameters

<i>s</i>	String to convert. Leading ASCII whitespace is skipped.
<i>end</i>	If non-NULL, *end is set to point to the first character not used in the conversion.
<i>base</i>	Numeric base in the range [2, 36].

Returns

The converted double value. Returns 0.0 on invalid base.

Note

Only digits '0'-'9', 'a'-'z', and 'A'-'Z' are recognized as base-N digits for the mantissa (the part before the exponent).

The exponent is always parsed in base 10 and represents the power of the specified base. For example, "1.5e2" in base 10 means $1.5 * 10^2 = 150$, while "A.8e2" in base 16 means $10.5 * 16^2 = 2688$.

The exponent can be positive or negative (e.g., "1e-3" = 0.001).

On invalid base, an error is printed to stderr, *end (if non-NULL) is set to `s`, and 0.0 is returned.

Examples:

```
asm_str2double("1.5e2", NULL, 10) // Returns 150.0
asm_str2double("-3.14e-1", NULL, 10) // Returns -0.314
asm_str2double("FF.0e1", NULL, 16) // Returns 4080.0 (255 × 16^1)
```

Definition at line 812 of file [Almog_String_Manipulation.h](#).

References [asm_check_char_belong_to_base\(\)](#), [asm_dprintERROR](#), [asm_get_char_value_in_base\(\)](#), [asm_isspace\(\)](#), and [asm_str2int\(\)](#).

4.7.3.28 asm_str2float()

```
float asm_str2float (
    const char *const s,
    const char **const end,
    const size_t base )
```

Convert a string to float in the given base with exponent support.

Identical to [asm_str2double](#) semantically, but returns a float and uses float arithmetic for the fractional part.

Parameters

<i>s</i>	String to convert. Leading ASCII whitespace is skipped.
<i>end</i>	If non-NULL, *end is set to point to the first character not used in the conversion.
<i>base</i>	Numeric base in the range [2, 36].

Returns

The converted float value. Returns 0.0f on invalid base.

Note

Only digits '0'-'9', 'a'-'z', and 'A'-'Z' are recognized as base-N digits for the mantissa (the part before the exponent).

The exponent is always parsed in base 10 and represents the power of the specified base. For example, "1.5e2" in base 10 means $1.5 \times 10^2 = 150$, while "A.8e2" in base 16 means $10.5 \times 16^2 = 2688$.

The exponent can be positive or negative (e.g., "1e-3" = 0.001).

On invalid base, an error is printed to stderr, *end (if non-NULL) is set to *s*, and 0.0f is returned.

Examples:

```
asm_str2float("1.5e2", NULL, 10) // Returns 150.0f
asm_str2float("-3.14e-1", NULL, 10) // Returns -0.314f
asm_str2float("FF.0e1", NULL, 16) // Returns 4080.0f (255 × 16^1)
```

Definition at line 899 of file [Almog_String_Manipulation.h](#).

References [asm_check_char_belong_to_base\(\)](#), [asm_dprintERROR](#), [asm_get_char_value_in_base\(\)](#), [asm_isspace\(\)](#), and [asm_str2int\(\)](#).

4.7.3.29 `asm_str2int()`

```
int asm_str2int (
    const char *const s,
    const char **const end,
    const size_t base )
```

Convert a string to int in the given base.

Parses an optional sign and then a sequence of base-N digits.

Parameters

<i>s</i>	String to convert. Leading ASCII whitespace is skipped.
<i>end</i>	If non-NULL, *end is set to point to the first character not used in the conversion.
<i>base</i>	Numeric base in the range [2, 36].

Returns

The converted int value. Returns 0 on invalid base.

Note

Only digits '0'-'9', 'a'-'z', and 'A'-'Z' are recognized as base-N digits.

On invalid base, an error is printed to stderr, *end (if non-NULL) is set to *s*, and 0 is returned.

Definition at line 973 of file [Almog_String_Manipulation.h](#).

References [asm_check_char_belong_to_base\(\)](#), [asm_dprintERROR](#), [asm_get_char_value_in_base\(\)](#), and [asm_isspace\(\)](#).

Referenced by [asm_str2double\(\)](#), and [asm_str2float\(\)](#).

4.7.3.30 `asm_str2size_t()`

```
size_t asm_str2size_t (
    const char *const s,
    const char **const end,
    const size_t base )
```

Convert a string to size_t in the given base.

Parses an optional leading '+' sign, then a sequence of base-N digits. Negative numbers are rejected.

Parameters

<i>s</i>	String to convert. Leading ASCII whitespace is skipped.
<i>end</i>	If non-NULL, *end is set to point to the first character not used in the conversion.
<i>base</i>	Numeric base in the range [2, 36].

Returns

The converted `size_t` value. Returns 0 on invalid base or if a negative sign is encountered.

Note

On invalid base or a negative sign, an error is printed to `stderr`, `*end` (if non-NULL) is set to `s`, and 0 is returned.

Definition at line 1018 of file [Almog_String_Manipulation.h](#).

References [asm_check_char_belong_to_base\(\)](#), [asm_dprintERROR](#), [asm_get_char_value_in_base\(\)](#), and [asm_isspace\(\)](#).

4.7.3.31 `asm_str_in_str()`

```
int asm_str_in_str (
    const char *const src,
    const char *const word_to_search )
```

Count occurrences of a substring within a string.

Counts how many times `word_to_search` appears in `src`. Occurrences may overlap.

Parameters

<code>src</code>	The string to search in (must be null-terminated).
<code>word_to_search</code>	The substring to find (must be null-terminated and non-empty).

Returns

The number of (possibly overlapping) occurrences found.

Note

If `word_to_search` is the empty string, the behavior is not well-defined and should be avoided.

Definition at line 769 of file [Almog_String_Manipulation.h](#).

References [asm_length\(\)](#), and [asm_strncmp\(\)](#).

4.7.3.32 `asm_str_is_whitespace()`

```
bool asm_str_is_whitespace (
    const char *const s )
```

Check whether a string contains only ASCII whitespace characters.

Parameters

s	Null-terminated string to test.
----------	---------------------------------

Returns

true if every character in *s* satisfies [asm_isspace\(\)](#), or if *s* is the empty string; false otherwise.

Definition at line 1086 of file [Almog_String_Manipulation.h](#).

References [asm_isspace\(\)](#), and [asm_length\(\)](#).

4.7.3.33 asm_strip_whitespace()

```
void asm_strip_whitespace (
    char *const s )
```

Remove all ASCII whitespace characters from a string in-place.

Scans *s* and deletes all characters for which [asm_isspace\(\)](#) is true, compacting the string and preserving the original order of non-whitespace characters.

Parameters

s	String to modify in-place. Must be null-terminated.
----------	---

Definition at line 1065 of file [Almog_String_Manipulation.h](#).

References [asm_isspace\(\)](#), [asm_length\(\)](#), and [asm_remove_char_from_string\(\)](#).

4.7.3.34 asm_strncat()

```
int asm_strncat (
    char *const s1,
    const char *const s2,
    const size_t N )
```

Append up to *N* characters from *s2* to the end of *s1*.

Appends characters from *s2* to the end of *s1* until either:

- *N* characters were appended, or
- a '\0' is encountered in *s2*.

After appending, this implementation writes a terminating '\0' to *s1*.

Parameters

<i>s1</i>	Destination string buffer (must be null-terminated).
<i>s2</i>	Source string buffer (must be null-terminated).
<i>N</i>	Maximum number of characters to append. If <i>N</i> == 0, the limit defaults to ASM_MAX_LEN.

Returns

The number of characters appended to *s1*.

Warning

This function uses ASM_MAX_LEN as an upper bound for the resulting length (excluding the terminating '\0'). The caller must ensure *s1* has capacity of at least ASM_MAX_LEN bytes.

Definition at line 1118 of file [Almog_String_Manipulation.h](#).

References [asm_dprintERROR](#), [asm_length\(\)](#), and [ASM_MAX_LEN](#).

4.7.3.35 asm_strncmp()

```
int asm_strncmp (
    const char * s1,
    const char * s2,
    const size_t N )
```

Compare up to *N* characters for equality (boolean result).

Returns 1 if the first *N* characters of *s1* and *s2* are all equal; otherwise returns 0. Unlike the standard C strncmp, which returns 0 on equality and a non-zero value on inequality/order, this function returns a boolean-like result (1 == equal, 0 == different).

Parameters

<i>s1</i>	First string (may be shorter than <i>N</i>).
<i>s2</i>	Second string (may be shorter than <i>N</i>).
<i>N</i>	Number of characters to compare.

Returns

1 if equal for the first *N* characters, 0 otherwise.

Note

If either string ends before *N* characters and the other does not, the strings are considered different.

Definition at line 1160 of file [Almog_String_Manipulation.h](#).

Referenced by [asm_str_in_str\(\)](#).

4.7.3.36 `asm_strncpy()`

```
int asm_strncpy (
    char *const s1,
    const char *const s2,
    const size_t N )
```

Copy up to N characters from `s2` into `s1` (non-standard).

Copies $n = \min(N, \text{len}(s2))$ characters from `s2` into `s1` and then writes a terminating `'\0'`.

Parameters

<code>s1</code>	Destination string buffer (must be null-terminated).
<code>s2</code>	Source string buffer (must be null-terminated).
<code>N</code>	Maximum number of characters to copy from <code>s2</code> .

Returns

The number of characters copied (i.e., (n)). Returns 0 and prints an error if $(n > \text{len}(s1))$.

Warning

This function does not check the capacity of `s1`. Instead, it checks the *current length* of the string in `s1` and refuses to copy more than that. This differs from the standard `strncpy()`.

Definition at line 1192 of file [Almog_String_Manipulation.h](#).

References [asm_dprintERROR](#), and [asm_length\(\)](#).

4.7.3.37 `asm_tolower()`

```
void asm_tolower (
    char *const s )
```

Convert all ASCII letters in a string to lowercase in-place.

Parameters

<code>s</code>	String to modify in-place. Must be null-terminated.
----------------	---

Definition at line 1220 of file [Almog_String_Manipulation.h](#).

References [asm_isupper\(\)](#), and [asm_length\(\)](#).

4.7.3.38 asm_toupper()

```
void asm_toupper (
    char *const s )
```

Convert all ASCII letters in a string to uppercase in-place.

Parameters

s	String to modify in-place. Must be null-terminated.
---	---

Definition at line 1235 of file [Almog_String_Manipulation.h](#).

References [asm_islower\(\)](#), and [asm_length\(\)](#).

4.7.3.39 asm_trim_left_whitespace()

```
void asm_trim_left_whitespace (
    char *const s )
```

Remove leading ASCII whitespace from a string in-place.

Finds the first character in *s* for which [asm_isspace\(\)](#) is false and left-shifts the string so that character becomes the first character.

Parameters

s	String to modify in-place. Must be null-terminated.
---	---

Definition at line 1253 of file [Almog_String_Manipulation.h](#).

References [asm_isspace\(\)](#), [asm_length\(\)](#), and [asm_shift_left\(\)](#).

4.8 Almog_String_Manipulation.h

```
00001
00041 #ifndef ALMOG_STRING_MANIPULATION_H_
00042 #define ALMOG_STRING_MANIPULATION_H_
00043
00044 #include <stdio.h>
00045 #include <stdbool.h>
00046 #include <stdint.h>
00047
00063 #ifndef ASM_MAX_LEN
00064 #define ASM_MAX_LEN (int)1e3
00065 #endif
00066
00074 #define asm_dprintSTRING(expr) printf(#expr " = %s\n", expr)
00075
00083 #define asm_dprintCHAR(expr) printf(#expr " = %c\n", expr)
00084
00092 #define asm_dprintINT(expr) printf(#expr " = %d\n", expr)
00093
00101 #define asm_dprintFLOAT(expr) printf(#expr " = %g\n", expr)
00102
00110 #define asm_dprintDOUBLE(expr) printf(#expr " = %g\n", expr)
```

```

00111
00119 #define asm_dprintSIZE_T(expr) printf(#expr " = %zu\n", expr)
00120
00121 #define asm_dprintERROR(fmt, ...) \
00122     fprintf(stderr, "\n%s:%d:\n[Error] in function '%s':\n"      " \
00123     fmt "\n\n", __FILE__, __LINE__, __func__, __VA_ARGS__)
00124
00136 #define asm_min(a, b) ((a) < (b) ? (a) : (b))
00137
00149 #define asm_max(a, b) ((a) > (b) ? (a) : (b))
00150
00151 bool    asm_check_char_belong_to_base(const char c, const size_t base);
00152 void    asm_copy_array_by_indexes(char * const target, const int start, const int end, const char *
const src);
00153 int     asm_get_char_value_in_base(const char c, const size_t base);
00154 int     asm_get_line(FILE *fp, char * const dst);
00155 int     asm_get_next_token_from_str(char * const dst, const char * const src, const char delimiter);
00156 int     asm_get_token_and_cut(char * const dst, char *src, const char delimiter, const bool
leave_delimiter);
00157 bool    asm_isalnum(const char c);
00158 bool    asm_isalpha(const char c);
00159 bool    asm_isbdigit(const char c);
00160 bool    asm_iscntrl(const char c);
00161 bool    asm_isdigit(const char c);
00162 bool    asm_isgraph(const char c);
00163 bool    asm_islower(const char c);
00164 bool    asm_isodigit(const char c);
00165 bool    asm_isprint(const char c);
00166 bool    asm_ispunct(const char c);
00167 bool    asm_isspace(const char c);
00168 bool    asm_isupper(const char c);
00169 bool    asm_isxdigit(const char c);
00170 bool    asm_isXdigit(const char c);
00171 size_t  asm_length(const char * const str);
00172 void *  asm_memset(void * const des, const unsigned char value, const size_t n);
00173 void    asm_pad_left(char * const s, const size_t padding, const char pad);
00174 void    asm_print_many_times(const char * const str, const size_t n);
00175 void    asm_remove_char_from_string(char * const s, const size_t index);
00176 void    asm_shift_left(char * const s, const size_t shift);
00177 int     asm_str_in_str(const char * const src, const char * const word_to_search);
00178 double  asm_str2double(const char * const s, const char ** const end, const size_t base);
00179 float   asm_str2float(const char * const s, const char ** const end, const size_t base);
00180 int     asm_str2int(const char * const s, const char ** const end, const size_t base);
00181 size_t  asm_str2size_t(const char * const s, const char ** const end, const size_t base);
00182 void    asm_strip_whitespace(char * const s);
00183 bool    asm_str_is_whitespace(const char * const s);
00184 int     asm_strncat(char * const s1, const char * const s2, const size_t N);
00185 int     asm_strncmp(const char * const s1, const char * const s2, const size_t N);
00186 int     asm_strncpy(char * const s1, const char * const s2, const size_t N);
00187 void    asm_tolower(char * const s);
00188 void    asm_toupper(char * const s);
00189 void    asm_trim_left_whitespace(char *s);
00190
00191 #endif /*ALMOG_STRING_MANIPULATION_H_*/
00192
00193 #ifdef ALMOG_STRING_MANIPULATION_IMPLEMENTATION
00194 #undef ALMOG_STRING_MANIPULATION_IMPLEMENTATION
00195
00206 bool asm_check_char_belong_to_base(const char c, const size_t base)
00207 {
00208     if (base > 36 || base < 2) {
00209         #ifndef NO_ERRORS
00210             asm_dprintERROR("Supported bases are [2...36]. Inputted: %zu", base);
00211         #endif
00212         return false;
00213     }
00214     if (base <= 10) {
00215         return c >= '0' && c <= '9'+(char)base-10;
00216     }
00217     if (base > 10) {
00218         return asm_isdigit(c) || (c >= 'A' && c <= ('A'+(char)base-11)) || (c >= 'a' && c <=
('a'+(char)base-11));
00219     }
00220
00221     return false;
00222 }
00223
00241 void asm_copy_array_by_indexes(char * const target, const int start, const int end, const char * const
src)
00242 {
00243     if (start > end) return;
00244     int j = 0;
00245     for (int i = start; i <= end; i++) {
00246         target[j] = src[i];
00247         j++;
00248     }
00249     if (target[j-1] != '\0') {

```

```

00250         target[j] = '\0';
00251     }
00252 }
00253
00264 int asm_get_char_value_in_base(const char c, const size_t base)
00265 {
00266     if (!asm_check_char_belong_to_base(c, base)) return -1;
00267     if (asm_isdigit(c)) {
00268         return c - '0';
00269     } else if (asm_isupper(c)) {
00270         return c - 'A' + 10;
00271     } else {
00272         return c - 'a' + 10;
00273     }
00274 }
00275
00297 int asm_get_line(FILE *fp, char * const dst)
00298 {
00299     int i = 0;
00300     int c;
00301     while ((c = fgetc(fp)) != '\n' && c != EOF) {
00302         dst[i++] = c;
00303         if (i >= ASM_MAX_LEN) {
00304             #ifndef NO_ERRORS
00305                 asm_dprintERROR("%s", "index exceeds ASM_MAX_LEN. Line in file is too long.");
00306             #endif
00307             dst[i-1] = '\0';
00308             return -1;
00309         }
00310     }
00311     dst[i] = '\0';
00312     if (c == EOF && i == 0) {
00313         return -1;
00314     }
00315     return i;
00316 }
00317
00344 int asm_get_next_token_from_str(char * const dst, const char * const src, const char delimiter)
00345 {
00346     int i = 0, j = 0;
00347     char c;
00348     while ((c = src[i]) != delimiter && c != '\0') {
00349         dst[j++] = src[i++];
00350     }
00351     dst[j] = '\0';
00352     return j;
00353 }
00354
00387 int asm_get_token_and_cut(char * const dst, char *src, const char delimiter, const bool
    leave_delimiter)
00388 {
00389     int new_src_start_index = asm_get_next_token_from_str(dst, src, delimiter);
00390     bool delimiter_at_start = src[new_src_start_index] == delimiter;
00391
00392     if (leave_delimiter) {
00393         asm_shift_left(src, new_src_start_index);
00394     } else if (delimiter_at_start) {
00395         asm_shift_left(src, new_src_start_index + 1);
00396     } else {
00397         src[0] = '\0';
00398     }
00399     return new_src_start_index ? 1 : 0;
00400 }
00401
00408 bool asm_isalnum(char c)
00409 {
00410     return asm_isalpha(c) || asm_isdigit(c);
00411 }
00412
00419 bool asm_isalpha(char c)
00420 {
00421     return asm_isupper(c) || asm_islower(c);
00422 }
00423
00430 bool asm_isbdigit(const char c)
00431 {
00432     if (c == '0' || c == '1') {
00433         return true;
00434     } else {
00435         return false;
00436     }
00437 }
00438
00445 bool asm_iscntrl(char c)
00446 {

```

```
00447     if ((c >= 0 && c <= 31) || c == 127) {
00448         return true;
00449     } else {
00450         return false;
00451     }
00452 }
00453
00460 bool asm_isdigit(char c)
00461 {
00462     if (c >= '0' && c <= '9') {
00463         return true;
00464     } else {
00465         return false;
00466     }
00467 }
00468
00475 bool asm_isgraph(char c)
00476 {
00477     if (c >= 33 && c <= 126) {
00478         return true;
00479     } else {
00480         return false;
00481     }
00482 }
00483
00490 bool asm_islower(char c)
00491 {
00492     if (c >= 'a' && c <= 'z') {
00493         return true;
00494     } else {
00495         return false;
00496     }
00497 }
00498
00505 bool asm_isodigit(const char c)
00506 {
00507     if ((c >= '0' && c <= '7')) {
00508         return true;
00509     } else {
00510         return false;
00511     }
00512 }
00513
00521 bool asm_isprint(char c)
00522 {
00523     return asm_isgraph(c) || c == ' ';
00524 }
00525
00533 bool asm_ispunct(char c)
00534 {
00535     if ((c >= 33 && c <= 47) || (c >= 58 && c <= 64) || (c >= 91 && c <= 96) || (c >= 123 && c <=
126)) {
00536         return true;
00537     } else {
00538         return false;
00539     }
00540 }
00541
00549 bool asm_isspace(char c)
00550 {
00551     if (c == ' ' || c == '\n' || c == '\t' ||
00552         c == '\v' || c == '\f' || c == '\r') {
00553         return true;
00554     } else {
00555         return false;
00556     }
00557 }
00558
00565 bool asm_isupper(char c)
00566 {
00567     if (c >= 'A' && c <= 'Z') {
00568         return true;
00569     } else {
00570         return false;
00571     }
00572 }
00573
00580 bool asm_isxdigit(char c)
00581 {
00582     if ((c >= 'a' && c <= 'f') || asm_isdigit(c)) {
00583         return true;
00584     } else {
00585         return false;
00586     }
00587 }
00588
00595 bool asm_isXdigit(char c)
```

```

00596 {
00597     if ((c >= 'A' && c <= 'F') || asm_isdigit(c)) {
00598         return true;
00599     } else {
00600         return false;
00601     }
00602 }
00603
00614 size_t asm_length(const char * const str)
00615 {
00616     char c;
00617     size_t i = 0;
00618
00619     while ((c = str[i++]) != '\0') {
00620         if (i > ASM_MAX_LEN) {
00621             #ifndef NO_ERRORS
00622                 asm_dprintERROR("%s", "index exceeds ASM_MAX_LEN. Probably no NULL termination.");
00623             #endif
00624             return SIZE_MAX;
00625         }
00626     }
00627     return --i;
00628 }
00629
00649 void * asm_memset(void * const des, const unsigned char value, const size_t n)
00650 {
00651     unsigned char *ptr = (unsigned char *)des;
00652     for (size_t i = n; i-- > 0;) {
00653         *ptr++ = value;
00654     }
00655     return des;
00656 }
00657
00672 void asm_pad_left(char * const s, const size_t padding, const char pad)
00673 {
00674     int len = (int)asm_length(s);
00675     for (int i = len; i >= 0; i--) {
00676         s[i+(int)padding] = s[i];
00677     }
00678     for (int i = 0; i < (int)padding; i++) {
00679         s[i] = pad;
00680     }
00681 }
00682
00689 void asm_print_many_times(const char * const str, const size_t n)
00690 {
00691     for (size_t i = 0; i < n; i++) {
00692         printf("%s", str);
00693     }
00694     printf("\n");
00695 }
00696
00709 void asm_remove_char_from_string(char * const s, const size_t index)
00710 {
00711     size_t len = asm_length(s);
00712     if (len == 0) return;
00713     if (index >= len) {
00714         #ifndef NO_ERRORS
00715             asm_dprintERROR("%s", "index exceeds array length.");
00716         #endif
00717         return;
00718     }
00719
00720     for (size_t i = index; i < len; i++) {
00721         s[i] = s[i+1];
00722     }
00723 }
00724
00738 void asm_shift_left(char * const s, const size_t shift)
00739 {
00740     size_t len = asm_length(s);
00741
00742     if (shift == 0) return;
00743     if (len <= shift) {
00744         s[0] = '\0';
00745         return;
00746     }
00747
00748     size_t i;
00749     for (i = shift; i < len; i++) {
00750         s[i-shift] = s[i];
00751     }
00752     s[i-shift] = '\0';
00753 }
00754
00769 int asm_str_in_str(const char * const src, const char * const word_to_search)
00770 {

```

```

00771     int i = 0, num_of_accur = 0;
00772     while (src[i] != '\0') {
00773         if (asm_strncmp(src+i, word_to_search, asm_length(word_to_search))) {
00774             num_of_accur++;
00775         }
00776         i++;
00777     }
00778     return num_of_accur;
00779 }
00780
00812 double asm_str2double(const char * const s, const char ** const end, const size_t base)
00813 {
00814     if (base < 2 || base > 36) {
00815         #ifndef NO_ERRORS
00816             asm_dprintfERROR("Supported bases are [2...36]. Input: %zu", base);
00817         #endif
00818         if (end) *end = s;
00819         return 0.0;
00820     }
00821     int num_of_whitespace = 0;
00822     while (asm_isspace(s[num_of_whitespace])) {
00823         num_of_whitespace++;
00824     }
00825
00826     int i = 0;
00827     if (s[0+num_of_whitespace] == '-' || s[0+num_of_whitespace] == '+') {
00828         i++;
00829     }
00830     int sign = s[0+num_of_whitespace] == '-' ? -1 : 1;
00831
00832     size_t left = 0;
00833     double right = 0.0;
00834     int expo = 0;
00835     for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
00836         left = base * left + asm_get_char_value_in_base(s[i+num_of_whitespace], base);
00837     }
00838
00839     if (s[i+num_of_whitespace] == '.') {
00840         i++; /* skip the point */
00841
00842         size_t divider = base;
00843         for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
00844             right = right + asm_get_char_value_in_base(s[i+num_of_whitespace], base) /
00845 (double)divider;
00846             divider *= base;
00847         }
00848
00849         if ((s[i+num_of_whitespace] == 'e') || (s[i+num_of_whitespace] == 'E')) {
00850             expo = asm_str2int(&s[i+num_of_whitespace+1], end, 10);
00851         } else {
00852             if (end) *end = s + i + num_of_whitespace;
00853         }
00854
00855         double res = sign * (left + right);
00856
00857         if (expo > 0) {
00858             for (int index = 0; index < expo; index++) {
00859                 res *= (double)base;
00860             }
00861         } else {
00862             for (int index = 0; index > expo; index--) {
00863                 res /= (double)base;
00864             }
00865         }
00866         return res;
00867     }
00868 }
00869
00899 float asm_str2float(const char * const s, const char ** const end, const size_t base)
00900 {
00901     if (base < 2 || base > 36) {
00902         #ifndef NO_ERRORS
00903             asm_dprintfERROR("Supported bases are [2...36]. Input: %zu", base);
00904         #endif
00905         if (end) *end = s;
00906         return 0.0f;
00907     }
00908     int num_of_whitespace = 0;
00909     while (asm_isspace(s[num_of_whitespace])) {
00910         num_of_whitespace++;
00911     }
00912
00913     int i = 0;
00914     if (s[0+num_of_whitespace] == '-' || s[0+num_of_whitespace] == '+') {
00915         i++;
00916     }

```



```

00917     int sign = s[0+num_of_whitespace] == '-' ? -1 : 1;
00918
00919     int left = 0;
00920     float right = 0.0f;
00921     int expo = 0;
00922     for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
00923         left = base * left + asm_get_char_value_in_base(s[i+num_of_whitespace], base);
00924     }
00925
00926     if (s[i+num_of_whitespace] == '.') {
00927         i++; /* skip the point */
00928
00929         size_t divider = base;
00930         for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
00931             right = right + asm_get_char_value_in_base(s[i+num_of_whitespace], base) / (float)divider;
00932             divider *= base;
00933         }
00934     }
00935
00936     if ((s[i+num_of_whitespace] == 'e') || (s[i+num_of_whitespace] == 'E')) {
00937         expo = asm_str2int(&(s[i+num_of_whitespace+1]), end, 10);
00938     } else {
00939         if (end) *end = s + i + num_of_whitespace;
00940     }
00941
00942     float res = sign * (left + right);
00943
00944     if (expo > 0) {
00945         for (int index = 0; index < expo; index++) {
00946             res *= (float)base;
00947         }
00948     } else {
00949         for (int index = 0; index > expo; index--) {
00950             res /= (float)base;
00951         }
00952     }
00953
00954     return res;
00955 }
00956
00973 int asm_str2int(const char * const s, const char ** const end, const size_t base)
00974 {
00975     if (base < 2 || base > 36) {
00976         #ifndef NO_ERRORS
00977             asm_dprintERROR("Supported bases are [2...36]. Input: %zu", base);
00978         #endif
00979         if (end) *end = s;
00980         return 0;
00981     }
00982     int num_of_whitespace = 0;
00983     while (asm_isspace(s[num_of_whitespace])) {
00984         num_of_whitespace++;
00985     }
00986
00987     int n = 0, i = 0;
00988     if (s[0+num_of_whitespace] == '-' || s[0+num_of_whitespace] == '+') {
00989         i++;
00990     }
00991     int sign = s[0+num_of_whitespace] == '-' ? -1 : 1;
00992
00993     for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
00994         n = base * n + asm_get_char_value_in_base(s[i+num_of_whitespace], base);
00995     }
00996
00997     if (end) *end = s + i+num_of_whitespace;
00998
00999     return n * sign;
01000 }
01001
01018 size_t asm_str2size_t(const char * const s, const char ** const end, const size_t base)
01019 {
01020     if (end) *end = s;
01021
01022     int num_of_whitespace = 0;
01023     while (asm_isspace(s[num_of_whitespace])) {
01024         num_of_whitespace++;
01025     }
01026
01027     if (s[0+num_of_whitespace] == '-') {
01028         #ifndef NO_ERRORS
01029             asm_dprintERROR("%s", "Unable to convert a negative number to size_t.");
01030         #endif
01031         return 0;
01032     }
01033
01034     if (base < 2 || base > 36) {
01035         #ifndef NO_ERRORS

```

```

01036         asm_dprintERROR("Supported bases are [2...36]. Input: %zu", base);
01037     #endif
01038     if (end) *end = s+num_of_whitespace;
01039     return 0;
01040 }
01041
01042 size_t n = 0, i = 0;
01043 if (s[0+num_of_whitespace] == '+') {
01044     i++;
01045 }
01046
01047 for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
01048     n = base * n + asm_get_char_value_in_base(s[i+num_of_whitespace], base);
01049 }
01050
01051 if (end) *end = s + i+num_of_whitespace;
01052
01053 return n;
01054 }
01055
01065 void asm_strip_whitespace(char * const s)
01066 {
01067     size_t len = asm_length(s);
01068     size_t i;
01069     for (i = 0; i < len; i++) {
01070         if (asm_isspace(s[i])) {
01071             asm_remove_char_from_string(s, i);
01072             len--;
01073             i--;
01074         }
01075     }
01076     s[i] = '\0';
01077 }
01078
01086 bool asm_str_is_whitespace(const char * const s)
01087 {
01088     size_t len = asm_length(s);
01089     for (size_t i = 0; i < len; i++) {
01090         if (!asm_isspace(s[i])) {
01091             return false;
01092         }
01093     }
01094
01095     return true;
01096 }
01097
01118 int asm_strncat(char * const s1, const char * const s2, const size_t N)
01119 {
01120     size_t len_s1 = asm_length(s1);
01121
01122     int limit = N;
01123     if (limit == 0) {
01124         limit = ASM_MAX_LEN;
01125     }
01126
01127     int i = 0;
01128     while (i < limit && s2[i] != '\0') {
01129         if (len_s1 + (size_t)i >= ASM_MAX_LEN-1) {
01130             #ifndef NO_ERRORS
01131                 asm_dprintERROR("s2 or the first N=%zu digit of s2 does not fit into s1.", N);
01132             #endif
01133             return i;
01134         }
01135
01136         s1[len_s1+(size_t)i] = s2[i];
01137         i++;
01138     }
01139     s1[len_s1+(size_t)i] = '\0';
01140
01141     return i;
01142 }
01143
01160 int asm_strncmp(const char *s1, const char *s2, const size_t N)
01161 {
01162     size_t i = 0;
01163     while (i < N) {
01164         if (s1[i] == '\0' && s2[i] == '\0') {
01165             break;
01166         }
01167         if (s1[i] != s2[i] || (s1[i] == '\0') || (s2[i] == '\0')) {
01168             return 0;
01169         }
01170         i++;
01171     }
01172     return 1;
01173 }
01174

```

```

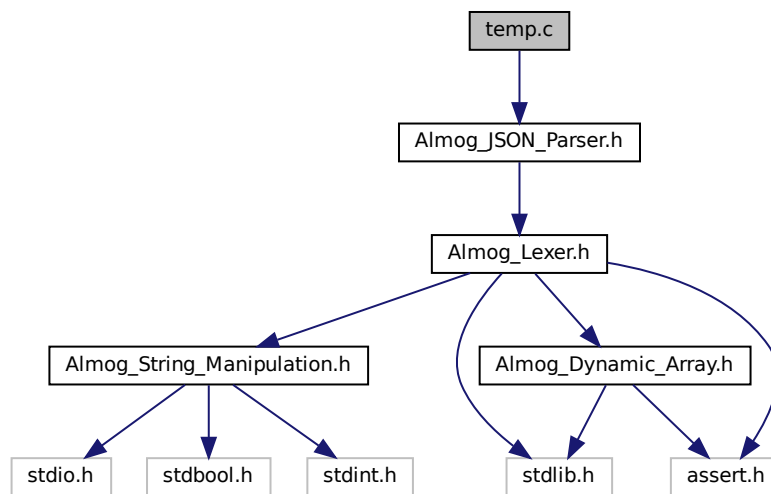
01192 int asm_strncpy(char * const s1, const char * const s2, const size_t N)
01193 {
01194     size_t len1 = asm_length(s1);
01195     size_t len2 = asm_length(s2);
01196
01197     size_t n = N < len2 ? N : len2;
01198
01199     if (n > len1) {
01200         #ifndef NO_ERRORS
01201             asm_dprintERROR("%s", "min(N, len(s2)) is bigger then len(s1)");
01202         #endif
01203         return 0;
01204     }
01205
01206     size_t i;
01207     for (i = 0; i < n; i++) {
01208         s1[i] = s2[i];
01209     }
01210     s1[i] = '\0';
01211
01212     return i;
01213 }
01214
01220 void asm_tolower(char * const s)
01221 {
01222     size_t len = asm_length(s);
01223     for (size_t i = 0; i < len; i++) {
01224         if (asm_isupper(s[i])) {
01225             s[i] += 'a' - 'A';
01226         }
01227     }
01228 }
01229
01235 void asm_toupper(char * const s)
01236 {
01237     size_t len = asm_length(s);
01238     for (size_t i = 0; i < len; i++) {
01239         if (asm_islower(s[i])) {
01240             s[i] += 'A' - 'a';
01241         }
01242     }
01243 }
01244
01253 void asm_trim_left_whitespace(char * const s)
01254 {
01255     size_t len = asm_length(s);
01256
01257     if (len == 0) return;
01258     size_t i;
01259     for (i = 0; i < len; i++) {
01260         if (!asm_isspace(s[i])) {
01261             break;
01262         }
01263     }
01264     asm_shift_left(s, i);
01265 }
01266
01267 #ifdef NO_ERRORS
01268 #undef NO_ERRORS
01269 #endif
01270
01271 #endif /*ALMOG_STRING_MANIPULATION_IMPLEMENTATION*/
01272

```

4.9 temp.c File Reference

```
#include "Almog_JSON_Parser.h"
```

Include dependency graph for temp.c:



Macros

- `#define` [ALMOG_STRING_MANIPULATION_IMPLEMENTATION](#)
- `#define` [ALMOG_LEXER_IMPLEMENTATION](#)
- `#define` [ALMOG_JSON_PARSER_IMPLEMENTATION](#)

Functions

- `int` [main](#) (`void`)

4.9.1 Macro Definition Documentation

4.9.1.1 ALMOG_JSON_PARSER_IMPLEMENTATION

```
#define ALMOG_JSON_PARSER_IMPLEMENTATION
```

Definition at line 3 of file [temp.c](#).

4.9.1.2 ALMOG_LEXER_IMPLEMENTATION

```
#define ALMOG_LEXER_IMPLEMENTATION
```

Definition at line 2 of file [temp.c](#).

4.9.1.3 ALMOG_STRING_MANIPULATION_IMPLEMENTATION

```
#define ALMOG_STRING_MANIPULATION_IMPLEMENTATION
```

Definition at line 1 of file [temp.c](#).

4.9.2 Function Documentation

4.9.2.1 main()

```
int main (  
    void )
```

Definition at line 6 of file [temp.c](#).

References [al_lex_entire_file\(\)](#), [al_token_print\(\)](#), [al_tokens_free\(\)](#), [asm_dprintSIZE_T](#), [Tokens::elements](#), and [Tokens::length](#).

4.10 temp.c

```
00001 #define ALMOG_STRING_MANIPULATION_IMPLEMENTATION  
00002 #define ALMOG_LEXER_IMPLEMENTATION  
00003 #define ALMOG_JSON_PARSER_IMPLEMENTATION  
00004 #include "Almog_JSON_Parser.h"  
00005  
00006 int main(void)  
00007 {  
00008     FILE *fp = fopen("./temp.c", "r");  
00009  
00010     struct Tokens tokens = al_lex_entire_file(fp);  
00011  
00012     for (size_t i = 0; i < tokens.length; i++) {  
00013         al_token_print(tokens.elements[i]);  
00014     }  
00015     asm_dprintSIZE_T(tokens.length);  
00016  
00017     al_tokens_free(tokens);  
00018  
00019  
00020     return 0;  
00021 }
```


Index

- ada_append
 - Almog_Dynamic_Array.h, [17](#)
- ADA_ASSERT
 - Almog_Dynamic_Array.h, [18](#)
- ADA_EXIT
 - Almog_Dynamic_Array.h, [18](#)
- ada_init_array
 - Almog_Dynamic_Array.h, [18](#)
- ADA_INIT_CAPACITY
 - Almog_Dynamic_Array.h, [19](#)
- ada_insert
 - Almog_Dynamic_Array.h, [19](#)
- ada_insert_unordered
 - Almog_Dynamic_Array.h, [20](#)
- ADA_MALLOC
 - Almog_Dynamic_Array.h, [21](#)
- ADA_REALLOC
 - Almog_Dynamic_Array.h, [21](#)
- ada_remove
 - Almog_Dynamic_Array.h, [22](#)
- ada_remove_unordered
 - Almog_Dynamic_Array.h, [22](#)
- ada_resize
 - Almog_Dynamic_Array.h, [23](#)
- AL_ASSERT
 - Almog_Lexer.h, [30](#)
- AL_FREE
 - Almog_Lexer.h, [30](#)
- al_is_identifier
 - Almog_Lexer.h, [33](#)
- al_is_identifier_start
 - Almog_Lexer.h, [33](#)
- al_lex_entire_file
 - Almog_Lexer.h, [34](#)
- al_lexer_alloc
 - Almog_Lexer.h, [34](#)
- al_lexer_chop_char
 - Almog_Lexer.h, [34](#)
- al_lexer_chop_while
 - Almog_Lexer.h, [35](#)
- al_lexer_next_token
 - Almog_Lexer.h, [35](#)
- al_lexer_peek
 - Almog_Lexer.h, [37](#)
- al_lexer_start_with
 - Almog_Lexer.h, [37](#)
- al_lexer_trim_left
 - Almog_Lexer.h, [38](#)
- al_token_kind_name
 - Almog_Lexer.h, [38](#)
- Almog_Lexer.h, [38](#)
- al_token_print
 - Almog_Lexer.h, [39](#)
- al_tokens_free
 - Almog_Lexer.h, [39](#)
- al_tokens_init
 - Almog_Lexer.h, [39](#)
- AL_UNUSED
 - Almog_Lexer.h, [30](#)
- Almog_Dynamic_Array.h, [15](#)
 - ada_append, [17](#)
 - ADA_ASSERT, [18](#)
 - ADA_EXIT, [18](#)
 - ada_init_array, [18](#)
 - ADA_INIT_CAPACITY, [19](#)
 - ada_insert, [19](#)
 - ada_insert_unordered, [20](#)
 - ADA_MALLOC, [21](#)
 - ADA_REALLOC, [21](#)
 - ada_remove, [22](#)
 - ada_remove_unordered, [22](#)
 - ada_resize, [23](#)
- Almog_JSON_Parser.h, [26](#)
- ALMOG_JSON_PARSER_IMPLEMENTATION
 - temp.c, [86](#)
- Almog_Lexer.h, [27](#)
 - AL_ASSERT, [30](#)
 - AL_FREE, [30](#)
 - al_is_identifier, [33](#)
 - al_is_identifier_start, [33](#)
 - al_lex_entire_file, [34](#)
 - al_lexer_alloc, [34](#)
 - al_lexer_chop_char, [34](#)
 - al_lexer_chop_while, [35](#)
 - al_lexer_next_token, [35](#)
 - al_lexer_peek, [37](#)
 - al_lexer_start_with, [37](#)
 - al_lexer_trim_left, [38](#)
 - al_token_kind_name, [38](#)
 - al_token_print, [39](#)
 - al_tokens_free, [39](#)
 - al_tokens_init, [39](#)
 - AL_UNUSED, [30](#)
 - keywords, [40](#)
 - keywords_count, [31](#)
 - literal_tokens, [40](#)
 - literal_tokens_count, [31](#)
 - TOKEN_ANDAND, [32](#)
 - TOKEN_ANDEQ, [33](#)

- TOKEN_ARROW, [33](#)
- TOKEN_BANG, [32](#)
- TOKEN_BITAND, [32](#)
- TOKEN_BITOR, [32](#)
- TOKEN_BSLASH, [32](#)
- TOKEN_CARET, [32](#)
- TOKEN_CHAR_LIT, [32](#)
- TOKEN_COLON, [32](#)
- TOKEN_COMMA, [32](#)
- TOKEN_COMMENT, [31](#)
- TOKEN_DOT, [32](#)
- TOKEN_ELLIPSIS, [33](#)
- TOKEN_EOF, [31](#)
- TOKEN_EQ, [32](#)
- TOKEN_EQEQ, [32](#)
- TOKEN_FLOAT_LIT_DEC, [32](#)
- TOKEN_FLOAT_LIT_HEX, [32](#)
- TOKEN_GE, [32](#)
- TOKEN_GT, [32](#)
- TOKEN_HASH, [32](#)
- TOKEN_IDENTIFIER, [32](#)
- TOKEN_INT_LIT_BIN, [32](#)
- TOKEN_INT_LIT_DEC, [32](#)
- TOKEN_INT_LIT_HEX, [32](#)
- TOKEN_INT_LIT_OCT, [32](#)
- TOKEN_INVALID, [31](#)
- TOKEN_KEYWORD, [32](#)
- Token_Kind, [31](#)
- TOKEN_LBRACE, [32](#)
- TOKEN_LBRACKET, [32](#)
- TOKEN_LE, [32](#)
- TOKEN_LPAREN, [32](#)
- TOKEN_LSHIFT, [32](#)
- TOKEN_LSHIFTEQ, [33](#)
- TOKEN_LT, [32](#)
- TOKEN_MINUS, [32](#)
- TOKEN_MINUSEQ, [32](#)
- TOKEN_MINUSMINUS, [32](#)
- TOKEN_NE, [32](#)
- TOKEN_OREQ, [33](#)
- TOKEN_OROR, [32](#)
- TOKEN_PERCENT, [32](#)
- TOKEN_PERCENTEQ, [33](#)
- TOKEN_PLUS, [32](#)
- TOKEN_PLUSEQ, [32](#)
- TOKEN_PLUSPLUS, [32](#)
- TOKEN_PP_DIRECTIVE, [31](#)
- TOKEN_QUESTION, [32](#)
- TOKEN_RBRACE, [32](#)
- TOKEN_RBRACKET, [32](#)
- TOKEN_LPAREN, [32](#)
- TOKEN_RSHIFT, [32](#)
- TOKEN_RSHIFTEQ, [33](#)
- TOKEN_SEMICOLON, [32](#)
- TOKEN_SLASH, [32](#)
- TOKEN_SLASHEQ, [32](#)
- TOKEN_STAR, [32](#)
- TOKEN_STAREQ, [32](#)
- TOKEN_STRING_LIT, [31](#)
- TOKEN_TILDE, [32](#)
- TOKEN_XOREQ, [33](#)
- ALMOG_LEXER_IMPLEMENTATION
 - temp.c, [86](#)
- Almog_String_Manipulation.h, [49](#)
 - asm_check_char_belong_to_base, [57](#)
 - asm_copy_array_by_indexes, [57](#)
 - asm_dprintCHAR, [53](#)
 - asm_dprintDOUBLE, [53](#)
 - asm_dprintERROR, [54](#)
 - asm_dprintFLOAT, [54](#)
 - asm_dprintINT, [54](#)
 - asm_dprintSIZE_T, [55](#)
 - asm_dprintSTRING, [55](#)
 - asm_get_char_value_in_base, [58](#)
 - asm_get_line, [58](#)
 - asm_get_next_token_from_str, [59](#)
 - asm_get_token_and_cut, [60](#)
 - asm_isalnum, [61](#)
 - asm_isalpha, [61](#)
 - asm_isbdigit, [62](#)
 - asm_iscntrl, [62](#)
 - asm_isdigit, [62](#)
 - asm_isgraph, [63](#)
 - asm_islower, [63](#)
 - asm_isodigit, [64](#)
 - asm_isprint, [64](#)
 - asm_isspace, [65](#)
 - asm_isupper, [65](#)
 - asm_isXdigit, [66](#)
 - asm_isxdigit, [66](#)
 - asm_length, [67](#)
 - asm_max, [55](#)
 - ASM_MAX_LEN, [56](#)
 - asm_memset, [67](#)
 - asm_min, [56](#)
 - asm_pad_left, [68](#)
 - asm_print_many_times, [68](#)
 - asm_remove_char_from_string, [69](#)
 - asm_shift_left, [69](#)
 - asm_str2double, [70](#)
 - asm_str2float, [71](#)
 - asm_str2int, [71](#)
 - asm_str2size_t, [72](#)
 - asm_str_in_str, [73](#)
 - asm_str_is_whitespace, [73](#)
 - asm_strip_whitespace, [74](#)
 - asm_strncat, [74](#)
 - asm_strncmp, [75](#)
 - asm_strncpy, [75](#)
 - asm_tolower, [76](#)
 - asm_toupper, [76](#)
 - asm_trim_left_whitespace, [77](#)
- ALMOG_STRING_MANIPULATION_IMPLEMENTATION
 - temp.c, [86](#)
- asm_check_char_belong_to_base

- Almog_String_Manipulation.h, [57](#)
- asm_copy_array_by_indexes
 - Almog_String_Manipulation.h, [57](#)
- asm_dprintCHAR
 - Almog_String_Manipulation.h, [53](#)
- asm_dprintDOUBLE
 - Almog_String_Manipulation.h, [53](#)
- asm_dprintERROR
 - Almog_String_Manipulation.h, [54](#)
- asm_dprintFLOAT
 - Almog_String_Manipulation.h, [54](#)
- asm_dprintINT
 - Almog_String_Manipulation.h, [54](#)
- asm_dprintSIZE_T
 - Almog_String_Manipulation.h, [55](#)
- asm_dprintSTRING
 - Almog_String_Manipulation.h, [55](#)
- asm_get_char_value_in_base
 - Almog_String_Manipulation.h, [58](#)
- asm_get_line
 - Almog_String_Manipulation.h, [58](#)
- asm_get_next_token_from_str
 - Almog_String_Manipulation.h, [59](#)
- asm_get_token_and_cut
 - Almog_String_Manipulation.h, [60](#)
- asm_isalnum
 - Almog_String_Manipulation.h, [61](#)
- asm_isalpha
 - Almog_String_Manipulation.h, [61](#)
- asm_isbdigit
 - Almog_String_Manipulation.h, [62](#)
- asm_iscntrl
 - Almog_String_Manipulation.h, [62](#)
- asm_isdigit
 - Almog_String_Manipulation.h, [62](#)
- asm_isgraph
 - Almog_String_Manipulation.h, [63](#)
- asm_islower
 - Almog_String_Manipulation.h, [63](#)
- asm_isodigit
 - Almog_String_Manipulation.h, [64](#)
- asm_isprint
 - Almog_String_Manipulation.h, [64](#)
- asm_ispunct
 - Almog_String_Manipulation.h, [64](#)
- asm_isspace
 - Almog_String_Manipulation.h, [65](#)
- asm_isupper
 - Almog_String_Manipulation.h, [65](#)
- asm_isXdigit
 - Almog_String_Manipulation.h, [66](#)
- asm_isxdigit
 - Almog_String_Manipulation.h, [66](#)
- asm_length
 - Almog_String_Manipulation.h, [67](#)
- asm_max
 - Almog_String_Manipulation.h, [55](#)
- ASM_MAX_LEN
- Almog_String_Manipulation.h, [56](#)
- asm_memset
 - Almog_String_Manipulation.h, [67](#)
- asm_min
 - Almog_String_Manipulation.h, [56](#)
- asm_pad_left
 - Almog_String_Manipulation.h, [68](#)
- asm_print_many_times
 - Almog_String_Manipulation.h, [68](#)
- asm_remove_char_from_string
 - Almog_String_Manipulation.h, [69](#)
- asm_shift_left
 - Almog_String_Manipulation.h, [69](#)
- asm_str2double
 - Almog_String_Manipulation.h, [70](#)
- asm_str2float
 - Almog_String_Manipulation.h, [71](#)
- asm_str2int
 - Almog_String_Manipulation.h, [71](#)
- asm_str2size_t
 - Almog_String_Manipulation.h, [72](#)
- asm_str_in_str
 - Almog_String_Manipulation.h, [73](#)
- asm_str_is_whitespace
 - Almog_String_Manipulation.h, [73](#)
- asm_strip_whitespace
 - Almog_String_Manipulation.h, [74](#)
- asm_strncat
 - Almog_String_Manipulation.h, [74](#)
- asm_strncmp
 - Almog_String_Manipulation.h, [75](#)
- asm_strncpy
 - Almog_String_Manipulation.h, [75](#)
- asm_tolower
 - Almog_String_Manipulation.h, [76](#)
- asm_toupper
 - Almog_String_Manipulation.h, [76](#)
- asm_trim_left_whitespace
 - Almog_String_Manipulation.h, [77](#)
- beginning_of_line
 - Lexer, [5](#)
- capacity
 - String, [9](#)
 - Tokens, [12](#)
- col
 - Location, [8](#)
- content
 - Lexer, [6](#)
 - Tokens, [12](#)
- content_len
 - Lexer, [6](#)
- cursor
 - Lexer, [6](#)
- elements
 - String, [9](#)
 - Tokens, [12](#)

- keywords
 - Almog_Lexer.h, 40
- keywords_count
 - Almog_Lexer.h, 31
- kind
 - Literal_Token, 7
 - Token, 10
- length
 - String, 9
 - Tokens, 13
- Lexer, 5
 - begining_of_line, 5
 - content, 6
 - content_len, 6
 - cursor, 6
 - line_num, 6
- line_num
 - Lexer, 6
 - Location, 8
- Literal_Token, 7
 - kind, 7
 - text, 7
- literal_tokens
 - Almog_Lexer.h, 40
- literal_tokens_count
 - Almog_Lexer.h, 31
- Location, 8
 - col, 8
 - line_num, 8
- location
 - Token, 10
- main
 - temp.c, 87
- String, 9
 - capacity, 9
 - elements, 9
 - length, 9
- temp.c, 85
 - ALMOG_JSON_PARSER_IMPLEMENTATION, 86
 - ALMOG_LEXER_IMPLEMENTATION, 86
 - ALMOG_STRING_MANIPULATION_IMPLEMENTATION, 86
 - main, 87
- text
 - Literal_Token, 7
 - Token, 11
- text_len
 - Token, 11
- Token, 10
 - kind, 10
 - location, 10
 - text, 11
 - text_len, 11
- TOKEN_ANDAND
 - Almog_Lexer.h, 32
- TOKEN_ANDEQ
 - Almog_Lexer.h, 33
- TOKEN_ARROW
 - Almog_Lexer.h, 33
- TOKEN_BANG
 - Almog_Lexer.h, 32
- TOKEN_BITAND
 - Almog_Lexer.h, 32
- TOKEN_BITOR
 - Almog_Lexer.h, 32
- TOKEN_BSLASH
 - Almog_Lexer.h, 32
- TOKEN_CARET
 - Almog_Lexer.h, 32
- TOKEN_CHAR_LIT
 - Almog_Lexer.h, 32
- TOKEN_COLON
 - Almog_Lexer.h, 32
- TOKEN_COMMA
 - Almog_Lexer.h, 32
- TOKEN_COMMENT
 - Almog_Lexer.h, 31
- TOKEN_DOT
 - Almog_Lexer.h, 32
- TOKEN_ELLIPSIS
 - Almog_Lexer.h, 33
- TOKEN_EOF
 - Almog_Lexer.h, 31
- TOKEN_EQ
 - Almog_Lexer.h, 32
- TOKEN_EQEQ
 - Almog_Lexer.h, 32
- TOKEN_FLOAT_LIT_DEC
 - Almog_Lexer.h, 32
- TOKEN_FLOAT_LIT_HEX
 - Almog_Lexer.h, 32
- TOKEN_GE
 - Almog_Lexer.h, 32
- TOKEN_GT
 - Almog_Lexer.h, 32
- TOKEN_HASH
 - Almog_Lexer.h, 32
- TOKEN_IDENTIFIER
 - Almog_Lexer.h, 32
- TOKEN_INT_LIT_BIN
 - Almog_Lexer.h, 32
- TOKEN_INT_LIT_DEC
 - Almog_Lexer.h, 32
- TOKEN_INT_LIT_HEX
 - Almog_Lexer.h, 32
- TOKEN_INT_LIT_OCT
 - Almog_Lexer.h, 32
- TOKEN_INVALID
 - Almog_Lexer.h, 31
- TOKEN_KEYWORD
 - Almog_Lexer.h, 32
- Token_Kind
 - Almog_Lexer.h, 31

- TOKEN_LBRACE
 - Almog_Lexer.h, [32](#)
- TOKEN_LBRACKET
 - Almog_Lexer.h, [32](#)
- TOKEN_LE
 - Almog_Lexer.h, [32](#)
- TOKEN_LPAREN
 - Almog_Lexer.h, [32](#)
- TOKEN_LSHIFT
 - Almog_Lexer.h, [32](#)
- TOKEN_LSHIFTEQ
 - Almog_Lexer.h, [33](#)
- TOKEN_LT
 - Almog_Lexer.h, [32](#)
- TOKEN_MINUS
 - Almog_Lexer.h, [32](#)
- TOKEN_MINUSEQ
 - Almog_Lexer.h, [32](#)
- TOKEN_MINUSMINUS
 - Almog_Lexer.h, [32](#)
- TOKEN_NE
 - Almog_Lexer.h, [32](#)
- TOKEN_OREQ
 - Almog_Lexer.h, [33](#)
- TOKEN_OROR
 - Almog_Lexer.h, [32](#)
- TOKEN_PERCENT
 - Almog_Lexer.h, [32](#)
- TOKEN_PERCENTEQ
 - Almog_Lexer.h, [33](#)
- TOKEN_PLUS
 - Almog_Lexer.h, [32](#)
- TOKEN_PLUSEQ
 - Almog_Lexer.h, [32](#)
- TOKEN_PLUSPLUS
 - Almog_Lexer.h, [32](#)
- TOKEN_PP_DIRECTIVE
 - Almog_Lexer.h, [31](#)
- TOKEN_QUESTION
 - Almog_Lexer.h, [32](#)
- TOKEN_RBRACE
 - Almog_Lexer.h, [32](#)
- TOKEN_RBRACKET
 - Almog_Lexer.h, [32](#)
- TOKEN_RPAREN
 - Almog_Lexer.h, [32](#)
- TOKEN_RSHIFT
 - Almog_Lexer.h, [32](#)
- TOKEN_RSHIFTEQ
 - Almog_Lexer.h, [33](#)
- TOKEN_SEMICOLON
 - Almog_Lexer.h, [32](#)
- TOKEN_SLASH
 - Almog_Lexer.h, [32](#)
- TOKEN_SLASHEQ
 - Almog_Lexer.h, [32](#)
- TOKEN_STAR
 - Almog_Lexer.h, [32](#)
- TOKEN_STAREQ
 - Almog_Lexer.h, [32](#)
- TOKEN_STRING_LIT
 - Almog_Lexer.h, [31](#)
- TOKEN_TILDE
 - Almog_Lexer.h, [32](#)
- TOKEN_XOREQ
 - Almog_Lexer.h, [33](#)
- Tokens, [11](#)
 - capacity, [12](#)
 - content, [12](#)
 - elements, [12](#)
 - length, [13](#)