Almog Lexer

Generated by Doxygen 1.9.1

Chapter 1

Class Index

1.1 Class List

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

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

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 216 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 221 of file Almog_Lexer.h.

Referenced by al_lexer_chop_char(), and test_helpers_direct().

3.1.2.2 content

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

Definition at line 217 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 218 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 219 of file Almog_Lexer.h.

Referenced by al_lexer_chop_char(), al_lexer_chop_while(), al_lexer_peek(), al_lexer_start_with(), al_lexer_trim_left(), and test_helpers_direct().

3.1.2.5 line_num

```
size_t Lexer::line_num
```

Definition at line 220 of file Almog_Lexer.h.

Referenced by al_lexer_chop_char(), and test_helpers_direct().

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

- Almog_Lexer.h

3.2 Literal_Token Struct Reference

`#include <Almog_Lexer.h>`

Public Attributes

- enum Token_Kind kind
- const char *const text

3.2.1 Detailed Description

Definition at line 113 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 682 of file Almog_Lexer.h.

3.2.2.2 text

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

Definition at line 115 of file Almog_Lexer.h.

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

- Almog_Lexer.h

3.3 Location Struct Reference

`#include <Almog_Lexer.h>`

Public Attributes

- size_t line_num
- size_t col

3.3.1 Detailed Description

Definition at line 187 of file Almog_Lexer.h.

3.3.2 Member Data Documentation

3.3.2.1 col

```
size_t Location::col
```

Definition at line 189 of file Almog_Lexer.h.

Referenced by al_token_print(), expect_tok(), and fail_token().

3.3.2.2 line_num

```
size_t Location::line_num
```

Definition at line 188 of file Almog_Lexer.h.

Referenced by al_token_print(), expect_tok(), and fail_token().

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

- Almog_Lexer.h

3.4 String Struct Reference

Public Attributes

- size_t length
- size_t capacity
- char * elements

3.4.1 Detailed Description

Definition at line 6 of file temp.c.

3.4.2 Member Data Documentation

3.4.2.1 capacity

```
size_t String::capacity
```

Definition at line 8 of file temp.c.

3.4.2.2 elements

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

Definition at line 9 of file temp.c.

3.4.2.3 length

```
size_t String::length
```

Definition at line 7 of file temp.c.

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

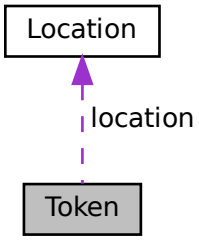
- temp.c

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 198 of file Almog_Lexer.h.

3.5.2 Member Data Documentation

3.5.2.1 kind

```
enum Token_Kind Token::kind
```

Definition at line 189 of file Almog_Lexer.h.

Referenced by al_token_print(), expect_tok(), fail_token(), and main().

3.5.2.2 location

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

Definition at line 201 of file Almog_Lexer.h.

Referenced by al_token_print(), expect_tok(), and fail_token().

3.5.2.3 text

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

Definition at line 200 of file Almog_Lexer.h.

Referenced by al_token_print(), expect_tok(), and fail_token().

3.5.2.4 text_len

```
size_t Token::text_len
```

Definition at line 201 of file Almog_Lexer.h.

Referenced by al_token_print(), expect_tok(), and fail_token().

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

- Almog_Lexer.h

Chapter 4

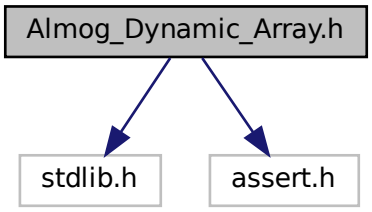
File Documentation

4.1 Almog_Dynamic_Array.h File Reference

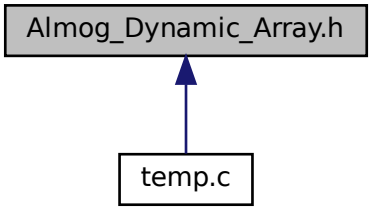
Header-only C macros that implement a simple dynamic array.

```
#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_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_append (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_append 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_append(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*1.5));   \
    }                                                           \
    header.elements[header.length] = value;                     \
    header.length++;                                            \
} while (0)
```

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

Parameters

type	Element type stored in the array.
header	Lvalue of the header struct.
value	Value to append.

Postcondition

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

Note

Growth factor is (int)(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 170 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 97 of file Almog_Dynamic_Array.h.

4.1.2.3 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

type	Element type stored in the array (e.g., int).
header	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 121 of file Almog_Dynamic_Array.h.

4.1.2.4 ADA_INIT_CAPACITY

```
#define ADA_INIT_CAPACITY 10
```

Default initial capacity used by ada_init_array.

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

Definition at line 62 of file Almog_Dynamic_Array.h.

4.1.2.5 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 (size_t ada_for_loop_index = header.length-2; ada_for_loop_index > (index); ada_for_loop_index--) {
       \
        header.elements[ada_for_loop_index] = header.elements [ada_for_loop_index-1];
       \
    }
       \
    header.elements[(index)] = value;
       \
} while (0)
```

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

Parameters

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

Precondition

0 <= index <= header.length.

header.length > 0 if index == 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 197 of file Almog_Dynamic_Array.h.

4.1.2.6 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_append(type, header, value);                            \
    } else {                                                        \
        ada_append(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

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

Precondition

0 <= index <= header.length.

Postcondition

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

Definition at line 223 of file Almog_Dynamic_Array.h.

4.1.2.7 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 73 of file Almog_Dynamic_Array.h.

4.1.2.8 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 85 of file Almog_Dynamic_Array.h.

4.1.2.9 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

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

Precondition

0 <= index < 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 247 of file Almog_Dynamic_Array.h.

4.1.2.10 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

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

Precondition

0 <= index < header.length and header.length > 0.

Postcondition

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

Definition at line 268 of file Almog_Dynamic_Array.h.

4.1.2.11 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) {
       \
            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

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

Precondition

new_capacity >= 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 exit(1).

Note

Reallocation uses ADA_REALLOC and is also checked via ADA_ASSERT.

Definition at line 144 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
00055
00062 #define ADA_INIT_CAPACITY 10
00063
00071 #ifndef ADA_MALLOC
00072 #include <stdlib.h>
00073 #define ADA_MALLOC malloc
00074 #endif /*ADA_MALLOC*/
00075
00083 #ifndef ADA_REALLOC
00084 #include <stdlib.h>
00085 #define ADA_REALLOC realloc
00086 #endif /*ADA_REALLOC*/
00087
00095 #ifndef ADA_ASSERT
00096 #include <assert.h>
00097 #define ADA_ASSERT assert
00098 #endif /*ADA_ASSERT*/
00099
00100 /* typedef struct {
00101     size_t length;
00102     size_t capacity;
00103     int* elements;
00104 } ada_int_array; */
00105
00121 #define ada_init_array(type, header) do {                                        \
00122         header.capacity = ADA_INIT_CAPACITY;                                        \
00123         header.length = 0;                                                       \
00124         header.elements = (type *)ADA_MALLOC(sizeof(type) * header.capacity);    \
00125         ADA_ASSERT(header.elements != NULL);                                     \
00126     } while (0)
00127
00144 #define ada_resize(type, header, new_capacity) do {
        \
00145         type *ada_temp_pointer = (type *)ADA_REALLOC((void *)(header.elements),
       new_capacity*sizeof(type)); \
00146         if (ada_temp_pointer == NULL) {
        \
00147             exit(1);
        \
00148         }
        \
00149         header.elements = ada_temp_pointer;
        \
00150         ADA_ASSERT(header.elements != NULL);
        \
00151         header.capacity = new_capacity;
        \
00152     } while (0)
00153
00170 #define ada_append(type, header, value) do {                          \
00171         if (header.length >= header.capacity) {                       \
00172             ada_resize(type, header, (int)(header.capacity*1.5));     \
00173         }                                                             \
00174         header.elements[header.length] = value;                       \
00175         header.length++;                                              \
00176     } while (0)
00177
00197 #define ada_insert(type, header, value, index) do {
        \
00198     ADA_ASSERT((int)(index) >= 0);
        \
00199     ADA_ASSERT((float)(index) - (int)(index) == 0);
        \
00200     ada_append(type, header, header.elements[header.length-1]);
        \
00201     for (size_t ada_for_loop_index = header.length-2; ada_for_loop_index > (index);
       ada_for_loop_index--) { \
00202         header.elements[ada_for_loop_index] = header.elements [ada_for_loop_index-1];
        \
00203     }
        \
00204     header.elements[(index)] = value;
        \
00205 } while (0)
00206
00207
00223 #define ada_insert_unordered(type, header, value, index) do {    \
00224     ADA_ASSERT((int)(index) >= 0);                               \
00225     ADA_ASSERT((float)(index) - (int)(index) == 0);              \
00226     if ((size_t)(index) == header.length) {                      \
```

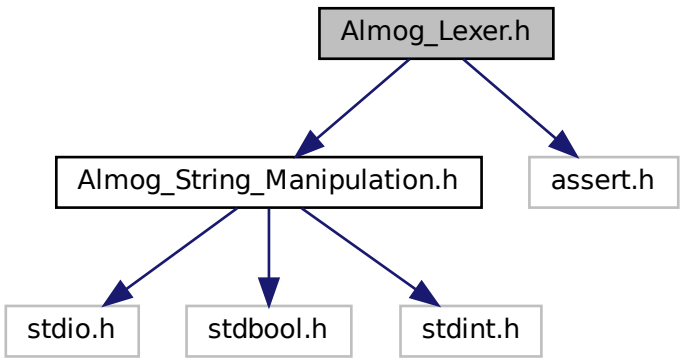
```
00227         ada_append(type, header, value);                          \
00228     } else {                                                      \
00229         ada_append(type, header, header.elements[(index)]);       \
00230         header.elements[(index)] = value;                         \
00231     }                                                             \
00232 } while (0)
00233
00247 #define ada_remove(type, header, index) do {
        \
00248     ADA_ASSERT((int)(index) >= 0);
        \
00249     ADA_ASSERT((float)(index) - (int)(index) == 0);
        \
00250     for (size_t ada_for_loop_index = (index); ada_for_loop_index < header.length-1;
      ada_for_loop_index++) { \
00251         header.elements[ada_for_loop_index] = header.elements[ada_for_loop_index+1];
        \
00252     }
        \
00253     header.length--;
        \
00254 } while (0)
00255
00268 #define ada_remove_unordered(type, header, index) do {            \
00269     ADA_ASSERT((int)(index) >= 0);                                \
00270     ADA_ASSERT((float)(index) - (int)(index) == 0);               \
00271     header.elements[index] = header.elements[header.length-1];    \
00272     header.length--;                                              \
00273 } while (0)
00274
00275
00276 #endif /*ALMOG_DYNAMIC_ARRAY_H_*/
```

4.3 Almog_Lexer.h File Reference

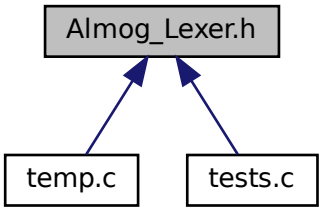
A small single-header lexer for C/C++-like source text.

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

Include dependency graph for Almog_Lexer.h:



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



Classes

- struct Literal_Token
- struct Location
- struct Token

  A token produced by the lexer.
- struct Lexer

  Lexer state over a caller-provided input buffer.

Macros

- #define AL_ASSERT assert
- #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
- #define ASM_NO_ERRORS

Enumerations

- enum Token_Kind {
  TOKEN_EOF , TOKEN_INVALID , TOKEN_PP_DIRECTIVE , TOKEN_COMMENT ,
  TOKEN_STRING_LIT , TOKEN_CHAR_LIT , TOKEN_NUMBER , 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 }

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 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.

Variables

- static struct Literal_Token literal_tokens [ ]
- static const char *const keywords [ ]

4.3.1 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 insapired by Tsoding's C-lexer implementation: `https://youtu.be/AqyZztKlSGQ`

Definition in file Almog_Lexer.h.

4.3.2 Macro Definition Documentation

4.3.2.1 AL_ASSERT

```
#define AL_ASSERT assert
```

Definition at line 23 of file Almog_Lexer.h.

4.3.2.2 AL_UNUSED

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

Definition at line 224 of file Almog_Lexer.h.

4.3.2.3 ASM_NO_ERRORS

```
#define ASM_NO_ERRORS
```

Definition at line 243 of file Almog_Lexer.h.

4.3.2.4 keywords_count

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

Definition at line 185 of file Almog_Lexer.h.

4.3.2.5 literal_tokens_count

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

Definition at line 168 of file Almog_Lexer.h.

4.3.3 Enumeration Type Documentation

4.3.3.1 Token_Kind

```
enum Token_Kind
```

Enumerator

TOKEN_EOF	
TOKEN_INVALID	
TOKEN_PP_DIRECTIVE	
TOKEN_COMMENT	
TOKEN_STRING_LIT	
TOKEN_CHAR_LIT	
TOKEN_NUMBER	
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 26 of file Almog_Lexer.h.

4.3.4 Function Documentation

4.3.4.1 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

c	Character to test.

Returns

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

Definition at line 254 of file Almog_Lexer.h.

References asm_isalnum().

Referenced by test_helpers_direct().

4.3.4.2 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

c	Character to test.

Returns

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

Definition at line 267 of file Almog_Lexer.h.

References asm_isalpha().

Referenced by test_helpers_direct().

4.3.4.3 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

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

Returns

A lexer initialized to the start of `content`.

Definition at line 267 of file Almog_Lexer.h.

Referenced by main(), test_basic_program(), test_comments(), test_hash_not_pp_directive_when_not_column1(), test_helpers_direct(), test_hex_float_variants(), test_invalid_single_char(), test_keyword_vs_identifier_prefix(), test_literal_operators_longest_match(), test_number_stops_on_invalid_digit_in_base(), test_numbers_valid_and_invalid(), test_pp_directive_and_locations(), test_string_and_char_literals(), test_unterminated_block_comment(), and test_whitespace_location_math().

4.3.4.4 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

l	Lexer to advance.

Returns

The consumed character.

Precondition

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

Definition at line 304 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(), al_lexer_trim_left(), and test_helpers_direct().

4.3.4.5 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

l	Lexer to advance.
pred	Predicate called with the next character to decide whether to consume it.

Definition at line 324 of file Almog_Lexer.h.

References al_lexer_chop_char(), Lexer::content, Lexer::content_len, and Lexer::cursor.

Referenced by test_helpers_direct().

4.3.4.6 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 "`</tt> and consumes until the next <tt>`" 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) => @c 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

l	Lexer to tokenize from.

Returns

The next token.

Definition at line 324 of file Almog_Lexer.h.

Referenced by expect_tok(), and main().

4.3.4.7 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

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

Returns

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

Definition at line 650 of file Almog_Lexer.h.

References Lexer::content, Lexer::content_len, and Lexer::cursor.

Referenced by test_helpers_direct().

4.3.4.8 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

l	Lexer whose input is tested.
prefix	Null-terminated prefix string to match.

Returns

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

Definition at line 608 of file Almog_Lexer.h.

References asm_length(), Lexer::content, Lexer::content_len, and Lexer::cursor.

Referenced by test_helpers_direct().

4.3.4.9 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 newlines update line/column bookkeeping.

Parameters

l	Lexer to advance.

Definition at line 633 of file Almog_Lexer.h.

References al_lexer_chop_char(), asm_isspace(), Lexer::content, Lexer::content_len, and Lexer::cursor.

4.3.4.10 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

kind	Token kind.

Returns

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

Definition at line 682 of file Almog_Lexer.h.

Referenced by al_token_print().

4.3.4.11 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

tok	Token to print.

Definition at line 668 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().

4.3.5 Variable Documentation

4.3.5.1 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", "consteval", "constexpr",
    "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",
}
```

Definition at line 170 of file Almog_Lexer.h.

4.3.5.2 literal_tokens

```
struct Literal_Token literal_tokens[]  [static]
```

Definition at line 1 of file Almog_Lexer.h.

4.4 Almog_Lexer.h

```
00001
00016 #ifndef ALMOG_LEXER_H_
00017 #define ALMOG_LEXER_H_
00018
00019 #include "Almog_String_Manipulation.h"
00020
00021 #ifndef AL_ASSERT
00022 #include <assert.h>
00023 #define AL_ASSERT assert
00024 #endif /* AL_ASSERT */
00025
00026 enum Token_Kind {
00027     /* Sentinel / unknown */
00028     TOKEN_EOF,
00029     TOKEN_INVALID,
00030
00031     /* High-level / multi-char / "word-like" */
00032     TOKEN_PP_DIRECTIVE,
00033     TOKEN_COMMENT,
00034     TOKEN_STRING_LIT,
00035     TOKEN_CHAR_LIT,
00036     TOKEN_NUMBER,
00037     TOKEN_KEYWORD,
00038     TOKEN_IDENTIFIER,
00039
00040
00041     /* Grouping / separators */
00042     TOKEN_LPAREN,
00043     TOKEN_RPAREN,
00044     TOKEN_LBRACKET,
00045     TOKEN_RBRACKET,
00046     TOKEN_LBRACE,
00047     TOKEN_RBRACE,
00048
00049     /* Punctuation */
00050     TOKEN_DOT,
00051     TOKEN_COMMA,
00052     TOKEN_SEMICOLON,
00053     TOKEN_BSLASH,
00054     TOKEN_HASH,
00055
00056     /* Ternary */
00057     TOKEN_QUESTION,
00058     TOKEN_COLON,
00059
00060     /* Assignment / equality */
00061     TOKEN_EQ,
00062     TOKEN_EQEQ,
00063     TOKEN_NE,
00064     TOKEN_BANG,
00065
00066     /* Relational */
00067     TOKEN_LT,
00068     TOKEN_GT,
00069     TOKEN_LE,
00070     TOKEN_GE,
00071
00072     /* Bitwise / boolean */
00073     TOKEN_BITAND,
00074     TOKEN_ANDAND,
00075     TOKEN_BITOR,
00076     TOKEN_OROR,
00077     /* Bitwise unary */
00078     TOKEN_CARET,
00079     TOKEN_TILDE,
00080
00081     /* Shifts */
00082     TOKEN_LSHIFT,
00083     TOKEN_RSHIFT,
00084
00085     /* Inc / dec */
00086     TOKEN_PLUSPLUS,
00087     TOKEN_MINUSMINUS,
00088
00089     /* Arithmetic */
00090     TOKEN_PLUS,
00091     TOKEN_MINUS,
00092     TOKEN_STAR,
00093     TOKEN_SLASH,
00094     TOKEN_PERCENT,
00095
00096     /* Compound assignment */
00097     TOKEN_PLUSEQ,
00098     TOKEN_MINUSEQ,
00099     TOKEN_STAREQ,
```

```
00100     TOKEN_SLASHEQ,
00101     TOKEN_PERCENTEQ,
00102     TOKEN_ANDEQ,
00103     TOKEN_OREQ,
00104     TOKEN_XOREQ,
00105     TOKEN_LSHIFTEQ,
00106     TOKEN_RSHIFTEQ,
00107 
00108     /* Member access / varargs */
00109     TOKEN_ARROW,
00110     TOKEN_ELLIPSIS,
00111 };
00112 
00113 struct Literal_Token {
00114     enum Token_Kind kind;
00115     const char * const text;
00116 };
00117 
00118 static struct Literal_Token literal_tokens[] = {
00119     {.text = "("  , .kind = TOKEN_LPAREN},
00120     {.text = ")"  , .kind = TOKEN_RPAREN},
00121     {.text = "["  , .kind = TOKEN_LBRACKET},
00122     {.text = "]"  , .kind = TOKEN_RBRACKET},
00123     {.text = "{"  , .kind = TOKEN_LBRACE},
00124     {.text = "}"  , .kind = TOKEN_RBRACE},
00125     {.text = "#"  , .kind = TOKEN_HASH},
00126     {.text = "...", .kind = TOKEN_ELLIPSIS},
00127     {.text = "."  , .kind = TOKEN_DOT},
00128     {.text = ","  , .kind = TOKEN_COMMA},
00129     {.text = "?"  , .kind = TOKEN_QUESTION},
00130     {.text = ":"  , .kind = TOKEN_COLON},
00131     {.text = "==" , .kind = TOKEN_EQEQ},
00132     {.text = "!=" , .kind = TOKEN_NE},
00133     {.text = "="  , .kind = TOKEN_EQ},
00134     {.text = "!"  , .kind = TOKEN_BANG},
00135     {.text = ";"  , .kind = TOKEN_SEMICOLON},
00136     {.text = "\\" , .kind = TOKEN_BSLASH},
00137     {.text = "->" , .kind = TOKEN_ARROW},
00138     {.text = ">"  , .kind = TOKEN_GT},
00139     {.text = ">=" , .kind = TOKEN_GE},
00140     {.text = "<"  , .kind = TOKEN_LT},
00141     {.text = "<=" , .kind = TOKEN_LE},
00142     {.text = "«=", .kind = TOKEN_LSHIFTEQ},
00143     {.text = "»=", .kind = TOKEN_RSHIFTEQ},
00144     {.text = "++" , .kind = TOKEN_PLUSPLUS},
00145     {.text = "--" , .kind = TOKEN_MINUSMINUS},
00146     {.text = "«" , .kind = TOKEN_LSHIFT},
00147     {.text = "»" , .kind = TOKEN_RSHIFT},
00148     {.text = "+=", .kind = TOKEN_PLUSEQ},
00149     {.text = "-=", .kind = TOKEN_MINUSEQ},
00150     {.text = "*=", .kind = TOKEN_STAREQ},
00151     {.text = "/=", .kind = TOKEN_SLASHEQ},
00152     {.text = "%=", .kind = TOKEN_PERCENTEQ},
00153     {.text = "&=", .kind = TOKEN_ANDEQ},
00154     {.text = "|=", .kind = TOKEN_OREQ},
00155     {.text = "^=", .kind = TOKEN_XOREQ},
00156     {.text = "||", .kind = TOKEN_OROR},
00157     {.text = "&&", .kind = TOKEN_ANDAND},
00158     {.text = "|" , .kind = TOKEN_BITOR},
00159     {.text = "&" , .kind = TOKEN_BITAND},
00160     {.text = "^" , .kind = TOKEN_CARET},
00161     {.text = "~" , .kind = TOKEN_TILDE},
00162     {.text = "+" , .kind = TOKEN_PLUS},
00163     {.text = "-" , .kind = TOKEN_MINUS},
00164     {.text = "*" , .kind = TOKEN_STAR},
00165     {.text = "/" , .kind = TOKEN_SLASH},
00166     {.text = "%" , .kind = TOKEN_PERCENT},
00167 };
00168 #define literal_tokens_count (sizeof(literal_tokens) / sizeof(literal_tokens[0]))
00169 
00170 static const char * const keywords[] = {
00171     "auto", "break", "case", "char", "const", "continue", "default", "do", "double",
00172     "else", "enum", "extern", "float", "for", "goto", "if", "int", "long", "register",
00173     "return", "short", "signed", "sizeof", "static", "struct", "switch", "typedef",
00174     "union", "unsigned", "void", "volatile", "while", "alignas", "alignof", "and",
00175     "and_eq", "asm", "atomic_cancel", "atomic_commit", "atomic_noexcept", "bitand",
00176     "bitor", "bool", "catch", "char16_t", "char32_t", "char8_t", "class", "co_await",
00177     "co_return", "co_yield", "compl", "concept", "const_cast", "consteval", "constexpr",
00178     "constinit", "decltype", "delete", "dynamic_cast", "explicit", "export", "false",
00179     "friend", "inline", "mutable", "namespace", "new", "noexcept", "not", "not_eq",
00180     "nullptr", "operator", "or", "or_eq", "private", "protected", "public", "reflexpr",
00181     "reinterpret_cast", "requires", "static_assert", "static_cast", "synchronized",
00182     "template", "this", "thread_local", "throw", "true", "try", "typeid", "typename",
00183     "using", "virtual", "wchar_t", "xor", "xor_eq",
00184 };
00185 #define keywords_count (sizeof(keywords) / sizeof(keywords[0]))
00186 
```

```
00187 struct Location {
00188     size_t line_num;
00189     size_t col;
00190 };
00191
00198 struct Token {
00199     enum Token_Kind kind;
00200     const char *text;
00201     size_t text_len;
00202     struct Location location;
00203 };
00204
00216 struct Lexer {
00217     const char * content;
00218     size_t content_len;
00219     size_t cursor;
00220     size_t line_num;
00221     size_t begining_of_line;
00222 };
00223
00224 #define AL_UNUSED(x) (void)x
00225
00226 bool            al_is_identifier(char c);
00227 bool            al_is_identifier_start(char c);
00228 struct Lexer    al_lexer_alloc(const char *content, size_t len);
00229 char            al_lexer_chop_char(struct Lexer *l);
00230 void            al_lexer_chop_while(struct Lexer *l, bool (*pred)(char));
00231 struct Token    al_lexer_next_token(struct Lexer *l);
00232 bool            al_lexer_start_with(struct Lexer *l, const char *prefix);
00233 void            al_lexer_trim_left(struct Lexer *l);
00234 char            al_lexer_peek(const struct Lexer *l, size_t off);
00235 void            al_token_print(struct Token tok);
00236 const char *    al_token_kind_name(enum Token_Kind kind);
00237
00238 #endif /*ALMOG_LEXER_H_*/
00239
00240 #ifdef ALMOG_LEXER_IMPLEMENTATION
00241 #undef ALMOG_LEXER_IMPLEMENTATION
00242
00243 #define ASM_NO_ERRORS
00244
00254 bool al_is_identifier(char c)
00255 {
00256     return asm_isalnum(c) || c == '_';
00257 }
00258
00267 bool al_is_identifier_start(char c)
00268 {
00269     return asm_isalpha(c) || c == '_';
00270 }
00271
00282 struct Lexer al_lexer_alloc(const char *content, size_t len)
00283 {
00284     struct Lexer l = {0};
00285     l.content = content;
00286     l.content_len = len;
00287     return l;
00288 }
00289
00304 char al_lexer_chop_char(struct Lexer *l)
00305 {
00306     AL_ASSERT(l->cursor < l->content_len);
00307     char c = l->content[l->cursor++];
00308     if (c == '\n') {
00309         l->line_num++;
00310         l->begining_of_line = l->cursor;
00311     }
00312     return c;
00313 }
00314
00324 void al_lexer_chop_while(struct Lexer *l, bool (*pred)(char))
00325 {
00326     while (l->cursor < l->content_len && pred(l->content[l->cursor])) {
00327         al_lexer_chop_char(l);
00328     }
00329 }
00330
00374 struct Token al_lexer_next_token(struct Lexer *l)
00375 {
00376     al_lexer_trim_left(l);
00377
00378     struct Token token = {
00379         .kind = TOKEN_INVALID,
00380         .text = &(l->content[l->cursor]),
00381         .text_len = 0,
00382         .location.line_num = l->line_num+1,
00383         .location.col = l->cursor - l->begining_of_line+1,
```

```
00384     };
00385     size_t start = l->cursor;
00386
00387     if (l->cursor >= l->content_len) {
00388         token.kind = TOKEN_EOF;
00389     } else if (l->content[l->cursor] == '#' && token.location.col == 1) {
00390         token.kind = TOKEN_PP_DIRECTIVE;
00391         for (;l->cursor < l->content_len && l->content[l->cursor] != '\n';) {
00392             al_lexer_chop_char(l);
00393         }
00394         if (l->cursor < l->content_len) {
00395             al_lexer_chop_char(l);
00396         }
00397     } else if (al_is_identifier_start(l->content[l->cursor])) {
00398         token.kind = TOKEN_IDENTIFIER;
00399         for ( ; l->cursor < l->content_len && al_is_identifier(l->content[l->cursor]); ) {
00400             al_lexer_chop_char(l);
00401         }
00402         {
00403             size_t ident_len = l->cursor - start;
00404             for (size_t i = 0; i < keywords_count; i++) {
00405                 size_t kw_len = asm_length(keywords[i]);
00406                 if (ident_len == kw_len && asm_strncmp(token.text, keywords[i], kw_len)) {
00407                     token.kind = TOKEN_KEYWORD;
00408                     break;
00409                 }
00410             }
00411         }
00412     } else if (l->content[l->cursor] == '"') {
00413         token.kind = TOKEN_STRING_LIT;
00414         al_lexer_chop_char(l);
00415         for ( ; (l->cursor < l->content_len) && (l->content[l->cursor] != '"') &&
      (l->content[l->cursor] != '\n'); ) {
00416             al_lexer_chop_char(l);
00417         }
00418         if ((l->cursor < l->content_len) && (l->content[l->cursor] == '"')) {
00419             al_lexer_chop_char(l);
00420         }
00421     } else if (l->content[l->cursor] == '\'') {
00422         token.kind = TOKEN_CHAR_LIT;
00423         al_lexer_chop_char(l);
00424         for ( ; (l->cursor < l->content_len) && (l->content[l->cursor] != '\'') &&
      (l->content[l->cursor] != '\n'); ) {
00425             al_lexer_chop_char(l);
00426         }
00427         if ((l->cursor < l->content_len) && (l->content[l->cursor] == '\'')) {
00428             al_lexer_chop_char(l);
00429         }
00430     } else if (al_lexer_start_with(l, "//")) {
00431         token.kind = TOKEN_COMMENT;
00432         for (;l->cursor < l->content_len && l->content[l->cursor] != '\n';) {
00433             al_lexer_chop_char(l);
00434         }
00435         if (l->cursor < l->content_len) {
00436             al_lexer_chop_char(l);
00437         }
00438     } else if (al_lexer_start_with(l, "/*")) {
00439         token.kind = TOKEN_COMMENT;
00440         al_lexer_chop_char(l);
00441         al_lexer_chop_char(l);
00442         for ( ; l->cursor < l->content_len; ) {
00443             if ((l->content[l->cursor-1] == '*') && (l->content[l->cursor] == '/')) {
00444                 al_lexer_chop_char(l);
00445                 break;
00446             }
00447             al_lexer_chop_char(l);
00448         }
00449     } else if (asm_isdigit(l->content[l->cursor]) || (l->content[l->cursor] == '.' &&
      asm_isdigit(al_lexer_peek(l, 1)))) {
00450         token.kind = TOKEN_NUMBER;
00451
00452         bool is_float = false;
00453         bool invalid = false;
00454
00455         /* decimal float starting with "." */
00456         if (l->content[l->cursor] == '.') {
00457             is_float = true;
00458             al_lexer_chop_char(l);
00459             al_lexer_chop_while(l, asm_isdigit);
00460
00461             /* optional exponent */
00462             if (al_lexer_peek(l, 0) == 'e' || al_lexer_peek(l, 0) == 'E') {
00463                 is_float = true;
00464                 al_lexer_chop_char(l);
00465                 if (al_lexer_peek(l, 0) == '+' || al_lexer_peek(l, 0) == '-') {
00466                     al_lexer_chop_char(l);
00467                 }
```

```
00468                 if (!asm_isdigit(al_lexer_peek(l, 0))) {
00469                     invalid = true; /* ".5e" or ".5e+" */
00470                 }
00471                 al_lexer_chop_while(l, asm_isdigit);
00472             }
00473         } else {
00474             /* starts with digit */
00475             if (al_lexer_peek(l, 0) == '0' && (al_lexer_peek(l, 1) == 'x' || al_lexer_peek(l, 1) ==
    'X')) {
00476                 /* hex int or hex float */
00477                 al_lexer_chop_char(l);
00478                 al_lexer_chop_char(l);
00479
00480                 size_t mantissa_digits = 0;
00481                 while (asm_isXdigit(al_lexer_peek(l, 0)) || asm_isxdigit(al_lexer_peek(l, 0))) {
00482                     mantissa_digits++;
00483                     al_lexer_chop_char(l);
00484                 }
00485                 if (al_lexer_peek(l, 0) == '.') {
00486                     is_float = true;
00487                     al_lexer_chop_char(l);
00488                     while (asm_isXdigit(al_lexer_peek(l, 0)) || asm_isxdigit(al_lexer_peek(l, 0))) {
00489                         mantissa_digits++;
00490                         al_lexer_chop_char(l);
00491                     }
00492                 }
00493                 if (mantissa_digits == 0) {
00494                     invalid = true; /* "0x" or "0x." */
00495                 }
00496
00497                 /* Hex float requires p/P exponent if it's a float form. */
00498                 if (al_lexer_peek(l, 0) == 'p' || al_lexer_peek(l, 0) == 'P') {
00499                     is_float = true;
00500                     al_lexer_chop_char(l);
00501                     if (al_lexer_peek(l, 0) == '+' || al_lexer_peek(l, 0) == '-') {
00502                         al_lexer_chop_char(l);
00503                     }
00504                     if (!asm_isdigit(al_lexer_peek(l, 0))) {
00505                         invalid = true; /* "0x1.fp" / "0x1p+" */
00506                     }
00507                     al_lexer_chop_while(l, asm_isdigit);
00508                 } else if (is_float) {
00509                     /* Had a '.' in hex mantissa but no p-exponent => invalid hex float */
00510                     invalid = true;
00511                 }
00512             } else if (al_lexer_peek(l, 0) == '0' && (al_lexer_peek(l, 1) == 'b' || al_lexer_peek(l,
    1) == 'B')) {
00513                 /* binary int */
00514                 al_lexer_chop_char(l);
00515                 al_lexer_chop_char(l);
00516                 if (!asm_isbdigit(al_lexer_peek(l, 0))) {
00517                     invalid = true; /* "0b" */
00518                 }
00519                 al_lexer_chop_while(l, asm_isbdigit);
00520             } else if (al_lexer_peek(l, 0) == '0' && (al_lexer_peek(l, 1) == 'o' || al_lexer_peek(l,
    1) == 'O')) {
00521                 /* explicit octal int */
00522                 al_lexer_chop_char(l);
00523                 al_lexer_chop_char(l);
00524                 if (!asm_isodigit(al_lexer_peek(l, 0))) {
00525                     invalid = true; /* "0o" */
00526                 }
00527                 while (asm_isodigit(al_lexer_peek(l, 0))) {
00528                     al_lexer_chop_char(l);
00529                 }
00530             } else {
00531                 /* decimal int or decimal float */
00532                 al_lexer_chop_while(l, asm_isdigit);
00533
00534                 if (al_lexer_peek(l, 0) == '.') {
00535                     is_float = true;
00536                     al_lexer_chop_char(l);
00537                     al_lexer_chop_while(l, asm_isdigit);
00538                 }
00539
00540                 if (al_lexer_peek(l, 0) == 'e' || al_lexer_peek(l, 0) == 'E') {
00541                     is_float = true;
00542                     al_lexer_chop_char(l);
00543                     if (al_lexer_peek(l, 0) == '+' || al_lexer_peek(l, 0) == '-') {
00544                         al_lexer_chop_char(l);
00545                     }
00546                     if (!asm_isdigit(al_lexer_peek(l, 0))) {
00547                         invalid = true; /* "1e" / "1e+" */
00548                     }
00549                     al_lexer_chop_while(l, asm_isdigit);
00550                 }
00551             }
```

```
00552         }
00553
00554         /* Suffix handling */
00555         if (is_float) {
00556             /* float suffixes: f/F/l/L (accept at most one, but we'll be permissive) */
00557             while (al_lexer_peek(l, 0) == 'f' || al_lexer_peek(l, 0) == 'F' ||
00558                    al_lexer_peek(l, 0) == 'l' || al_lexer_peek(l, 0) == 'L') {
00559                 al_lexer_chop_char(l);
00560             }
00561         } else {
00562             /* integer suffixes: u/U/l/L/z/Z (permissive) */
00563             while (al_lexer_peek(l, 0) == 'u' || al_lexer_peek(l, 0) == 'U' ||
00564                    al_lexer_peek(l, 0) == 'l' || al_lexer_peek(l, 0) == 'L' ||
00565                    al_lexer_peek(l, 0) == 'z' || al_lexer_peek(l, 0) == 'Z') {
00566                 al_lexer_chop_char(l);
00567             }
00568         }
00569
00570         if (invalid) token.kind = TOKEN_INVALID;
00571     } else {
00572         size_t longest_matching_token = 0;
00573         enum Token_Kind best_kind = TOKEN_INVALID;
00574         for (size_t i = 0; i < literal_tokens_count; i++) {
00575             if (al_lexer_start_with(l, literal_tokens[i].text)) {
00576                 /* NOTE: assumes that literal_tokens[i].text does not have any '\n' */
00577                 size_t text_len = asm_length(literal_tokens[i].text);
00578                 if (text_len > longest_matching_token) {
00579                     longest_matching_token = text_len;
00580                     best_kind = literal_tokens[i].kind;                }
00581             }
00582         }
00583         if (longest_matching_token > 0) {
00584             token.kind = best_kind;
00585             for (size_t i = 0; i < longest_matching_token; i++) {
00586                 al_lexer_chop_char(l);
00587             }
00588         } else {
00589             token.kind = TOKEN_INVALID;
00590             al_lexer_chop_char(l);
00591         }
00592     }
00593
00594     token.text_len = l->cursor - start;
00595
00596     return token;
00597 }
00598
00608 bool al_lexer_start_with(struct Lexer *l, const char *prefix)
00609 {
00610     size_t prefix_len = asm_length(prefix);
00611     if (prefix_len == 0) {
00612         return true;
00613     }
00614     if (l->cursor + prefix_len > l->content_len) {
00615         return false;
00616     }
00617     for (size_t i = 0; i < prefix_len; i++) {
00618         if (prefix[i] != l->content[l->cursor + i]) {
00619             return false;
00620         }
00621     }
00622     return true;
00623 }
00624
00633 void al_lexer_trim_left(struct Lexer *l)
00634 {
00635     for (;l->cursor < l->content_len;) {
00636         if (!asm_isspace(l->content[l->cursor])) {
00637             break;
00638         }
00639         al_lexer_chop_char(l);
00640     }
00641 }
00642
00650 char al_lexer_peek(const struct Lexer *l, size_t off)
00651 {
00652     size_t i = l->cursor + off;
00653     if (i >= l->content_len) return '\0';
00654     return l->content[i];
00655 }
00656
00668 void al_token_print(struct Token tok)
00669 {
00670     printf("%4zu:%-3zu:(%-18s) -> \"%.*s\"\n", tok.location.line_num, tok.location.col,
       al_token_kind_name(tok.kind), (int)tok.text_len, tok.text);
00671 }
00672
```

```
00682 const char *al_token_kind_name(enum Token_Kind kind)
00683 {
00684     switch (kind) {
00685         case TOKEN_EOF:
00686             return ("TOKEN_EOF");
00687         case TOKEN_INVALID:
00688             return ("TOKEN_INVALID");
00689         case TOKEN_PP_DIRECTIVE:
00690             return ("TOKEN_PP_DIRECTIVE");
00691         case TOKEN_IDENTIFIER:
00692             return ("TOKEN_IDENTIFIER");
00693         case TOKEN_LPAREN:
00694             return ("TOKEN_LPAREN");
00695         case TOKEN_RPAREN:
00696             return ("TOKEN_RPAREN");
00697         case TOKEN_LBRACKET:
00698             return ("TOKEN_LBRACKET");
00699         case TOKEN_RBRACKET:
00700             return ("TOKEN_RBRACKET");
00701         case TOKEN_LBRACE:
00702             return ("TOKEN_LBRACE");
00703         case TOKEN_RBRACE:
00704             return ("TOKEN_RBRACE");
00705         case TOKEN_DOT:
00706             return ("TOKEN_DOT");
00707         case TOKEN_COMMA:
00708             return ("TOKEN_COMMA");
00709         case TOKEN_SEMICOLON:
00710             return ("TOKEN_SEMICOLON");
00711         case TOKEN_BSLASH:
00712             return ("TOKEN_BSLASH");
00713         case TOKEN_QUESTION:
00714             return ("TOKEN_QUESTION");
00715         case TOKEN_COLON:
00716             return ("TOKEN_COLON");
00717         case TOKEN_LT:
00718             return ("TOKEN_LT");
00719         case TOKEN_GT:
00720             return ("TOKEN_GT");
00721         case TOKEN_GE:
00722             return ("TOKEN_GE");
00723         case TOKEN_LE:
00724             return ("TOKEN_LE");
00725         case TOKEN_KEYWORD:
00726             return ("TOKEN_KEYWORD");
00727         case TOKEN_NUMBER:
00728             return ("TOKEN_NUMBER");
00729         case TOKEN_COMMENT:
00730             return ("TOKEN_COMMENT");
00731         case TOKEN_STRING_LIT:
00732             return ("TOKEN_STRING_LIT");
00733         case TOKEN_CHAR_LIT:
00734             return ("TOKEN_CHAR_LIT");
00735         case TOKEN_EQ:
00736             return ("TOKEN_EQ");
00737         case TOKEN_EQEQ:
00738             return ("TOKEN_EQEQ");
00739         case TOKEN_NE:
00740             return ("TOKEN_NE");
00741         case TOKEN_BANG:
00742             return ("TOKEN_BANG");
00743         case TOKEN_BITAND:
00744             return ("TOKEN_BITAND");
00745         case TOKEN_ANDAND:
00746             return ("TOKEN_ANDAND");
00747         case TOKEN_BITOR:
00748             return ("TOKEN_BITOR");
00749         case TOKEN_OROR:
00750             return ("TOKEN_OROR");
00751         case TOKEN_CARET:
00752             return ("TOKEN_CARET");
00753         case TOKEN_TILDE:
00754             return ("TOKEN_TILDE");
00755         case TOKEN_PLUSPLUS:
00756             return ("TOKEN_PLUSPLUS");
00757         case TOKEN_MINUSMINUS:
00758             return ("TOKEN_MINUSMINUS");
00759         case TOKEN_LSHIFT:
00760             return ("TOKEN_LSHIFT");
00761         case TOKEN_RSHIFT:
00762             return ("TOKEN_RSHIFT");
00763         case TOKEN_PLUS:
00764             return ("TOKEN_PLUS");
00765         case TOKEN_MINUS:
00766             return ("TOKEN_MINUS");
00767         case TOKEN_STAR:
00768             return ("TOKEN_STAR");
```

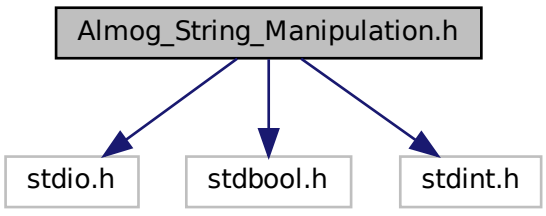
```
00769         case TOKEN_SLASH:
00770             return ("TOKEN_SLASH");
00771         case TOKEN_HASH:
00772             return ("TOKEN_HASH");
00773         case TOKEN_PERCENT:
00774             return ("TOKEN_PERCENT");
00775         case TOKEN_PLUSEQ:
00776             return ("TOKEN_PLUSEQ");
00777         case TOKEN_MINUSEQ:
00778             return ("TOKEN_MINUSEQ");
00779         case TOKEN_STAREQ:
00780             return ("TOKEN_STAREQ");
00781         case TOKEN_SLASHEQ:
00782             return ("TOKEN_SLASHEQ");
00783         case TOKEN_PERCENTEQ:
00784             return ("TOKEN_PERCENTEQ");
00785         case TOKEN_ANDEQ:
00786             return ("TOKEN_ANDEQ");
00787         case TOKEN_OREQ:
00788             return ("TOKEN_OREQ");
00789         case TOKEN_XOREQ:
00790             return ("TOKEN_XOREQ");
00791         case TOKEN_LSHIFTEQ:
00792             return ("TOKEN_LSHIFTEQ");
00793         case TOKEN_RSHIFTEQ:
00794             return ("TOKEN_RSHIFTEQ");
00795         case TOKEN_ARROW:
00796             return ("TOKEN_ARROW");
00797         case TOKEN_ELLIPSIS:
00798             return ("TOKEN_ELLIPSIS");
00799         default:
00800             AL_ASSERT(0 && "Unknown kind");
00801     }
00802     return NULL;
00803 }
00804
00805 #endif /*ALMOG_LEXER_IMPLEMENTATION*/
00806
```

4.5 Almog_String_Manipulation.h File Reference

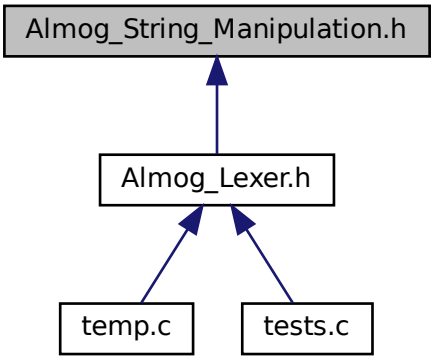
Lightweight string and line manipulation helpers.

```
#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.

4.5.1 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.

4.5.2 Macro Definition Documentation

4.5.2.1 asm_dprintCHAR

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

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

Parameters

expr	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.

4.5.2.2 asm_dprintDOUBLE

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

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

Parameters

expr	An expression that yields a double. The expression is evaluated exactly once.

Definition at line 110 of file Almog_String_Manipulation.h.

4.5.2.3 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.

4.5.2.4 asm_dprintFLOAT

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

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

Parameters

expr	An expression that yields a float. The expression is evaluated exactly once.

Definition at line 101 of file Almog_String_Manipulation.h.

4.5.2.5 asm_dprintINT

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

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

Parameters

expr	An expression that yields an int. The expression is evaluated exactly once.

Definition at line 92 of file Almog_String_Manipulation.h.

4.5.2.6 asm_dprintSIZE_T

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

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

Parameters

expr	An expression that yields a size_t. The expression is evaluated exactly once.

Definition at line 119 of file Almog_String_Manipulation.h.

4.5.2.7 asm_dprintSTRING

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

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

Parameters

expr	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.

4.5.2.8 asm_max

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

Return the larger of two values (macro).

Parameters

a	First value.
b	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.

4.5.2.9 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.

4.5.2.10 asm_min

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

Return the smaller of two values (macro).

Parameters

a	First value.
b	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.

4.5.3 Function Documentation

4.5.3.1 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

c	Character to test (e.g., '0'–'9', 'a'–'z', 'A'–'Z').
base	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().

4.5.3.2 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

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

4.5.3.3 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

c	Digit character ('0'–'9', 'a'–'z', 'A'–'Z').
base	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().

4.5.3.4 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

fp	Input stream (must be non-NULL).
dst	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.

Referenced by main().

4.5.3.5 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

dst	Destination buffer for the extracted token. Must be large enough to hold the token plus the null terminator.
src	Source C string to parse (not modified by this function).
delimiter	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 ('
') 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().

4.5.3.6 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

dst	Destination buffer for the extracted token (must be large enough for the token plus the null terminator).
src	Source buffer, modified in-place by this function.
delimiter	Delimiter character used to stop token extraction.
leave_delimiter	If true, do not remove the delimiter from `src`; 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().

4.5.3.7 asm_isalnum()

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

Test for an alphanumeric character (ASCII).

Parameters

c	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().

4.5.3.8 asm_isalpha()

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

Test for an alphabetic character (ASCII).

Parameters

c	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(), asm_isalnum(), and test_helpers_direct().

4.5.3.9 asm_isbdigit()

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

Test for a binary digit (ASCII).

Parameters

c	Character to test.

Returns

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

Definition at line 430 of file Almog_String_Manipulation.h.

4.5.3.10 asm_iscntrl()

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

Test for a control character (ASCII).

Parameters

c	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.

4.5.3.11 asm_isdigit()

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

Test for a decimal digit (ASCII).

Parameters

c	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().

4.5.3.12 asm_isgraph()

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

Test for any printable character except space (ASCII).

Parameters

c	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().

4.5.3.13 asm_islower()

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

Test for a lowercase letter (ASCII).

Parameters

c	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().

4.5.3.14 asm_isodigit()

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

Test for an octal digit (ASCII).

Parameters

c	Character to test.

Returns

true if c is '0'–'7'; false otherwise.

Definition at line 505 of file Almog_String_Manipulation.h.

4.5.3.15 asm_isprint()

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

Test for any printable character including space (ASCII).

Parameters

c	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().

4.5.3.16 asm_ispunct()

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

Test for a punctuation character (ASCII).

Parameters

c	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.

4.5.3.17 asm_isspace()

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

Test for a whitespace character (ASCII).

Parameters

c	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().

4.5.3.18 asm_isupper()

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

Test for an uppercase letter (ASCII).

Parameters

c	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().

4.5.3.19 asm_isxdigit()

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

Test for a hexadecimal digit (lowercase or decimal).

Parameters

c	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().

4.5.3.20 asm_isXdigit()

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

Test for a hexadecimal digit (uppercase or decimal).

Parameters

c	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().

4.5.3.21 asm_length()

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

Compute the length of a null-terminated C string.

Parameters

str	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().

4.5.3.22 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

des	Destination memory block to modify. Must point to a valid buffer of at least `n` bytes.
value	Unsigned byte value to store repeatedly.
n	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.

4.5.3.23 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

s	String to pad. Modified in-place.
padding	Number of leading spaces to insert.
pad	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().

4.5.3.24 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

str	String to print (as-is with printf("%s", ...)).
n	Number of times to print `str`.

Definition at line 689 of file Almog_String_Manipulation.h.

4.5.3.25 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

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

4.5.3.26 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

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

4.5.3.27 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

s	String to convert. Leading ASCII whitespace is skipped.
end	If non-NULL, ∗end is set to point to the first character not used in the conversion.
base	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().

4.5.3.28 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

s	String to convert. Leading ASCII whitespace is skipped.
end	If non-NULL, ∗end is set to point to the first character not used in the conversion.
base	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 ∗ 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 0.0f 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().

4.5.3.29 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

s	String to convert. Leading ASCII whitespace is skipped.
end	If non-NULL, *end is set to point to the first character not used in the conversion.
base	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().

4.5.3.30 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

s	String to convert. Leading ASCII whitespace is skipped.
end	If non-NULL, *end is set to point to the first character not used in the conversion.
base	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().

4.5.3.31 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

src	The string to search in (must be null-terminated).
word_to_search	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().

4.5.3.32 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().

4.5.3.33 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().

4.5.3.34 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

s1	Destination string buffer (must be null-terminated).
s2	Source string buffer (must be null-terminated).
N	Maximum number of characters to append. If N == 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.

4.5.3.35 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

s1	First string (may be shorter than `N`).
s2	Second string (may be shorter than `N`).
N	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().

4.5.3.36 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, len(s2)) characters from `s2` into `s1` and then writes a terminating '\0'.

Parameters

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

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().

4.5.3.37 asm_tolower()

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

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

Parameters

s	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().

4.5.3.38 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().

4.5.3.39 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().

4.6 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
00352     dst[j] = '\0';
00353
00354     return j;
00355 }
00356
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_dprintERROR("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) /
      (double)divider;
00845             divider *= base;
00846         }
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
00867     return res;
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_dprintERROR("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
```

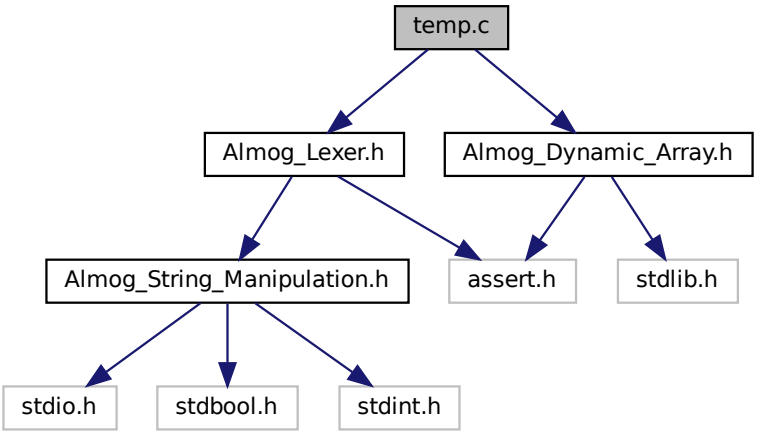
```
01036         asm_dprintERROR("Supported bases are [2...36]. Input: %zu", base);
01037         #endif
01038         if (end) *end = s+num_of_whitespace;
01039         return 0;
01040     }
01041
01042     size_t n = 0, i = 0;
01043     if (s[0+num_of_whitespace] == '+') {
01044         i++;
01045     }
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 
```

4.7 temp.c File Reference

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

Include dependency graph for temp.c:



Classes

- struct String

Macros

- #define ALMOG_STRING_MANIPULATION_IMPLEMENTATION
- #define ALMOG_LEXER_IMPLEMENTATION

Functions

- int main (void)

4.7.1 Macro Definition Documentation

4.7.1.1 ALMOG_LEXER_IMPLEMENTATION

```
#define ALMOG_LEXER_IMPLEMENTATION
```

Definition at line 2 of file temp.c.

4.7.1.2 ALMOG_STRING_MANIPULATION_IMPLEMENTATION

```
#define ALMOG_STRING_MANIPULATION_IMPLEMENTATION
```

Definition at line 1 of file temp.c.

4.7.2 Function Documentation

4.7.2.1 main()

```
int main (
            void  )
```

Definition at line 12 of file temp.c.

References ada_append, ada_init_array, al_lexer_alloc(), al_lexer_next_token(), al_token_print(), asm_dprintINT, asm_get_line(), ASM_MAX_LEN, Token::kind, and TOKEN_EOF.

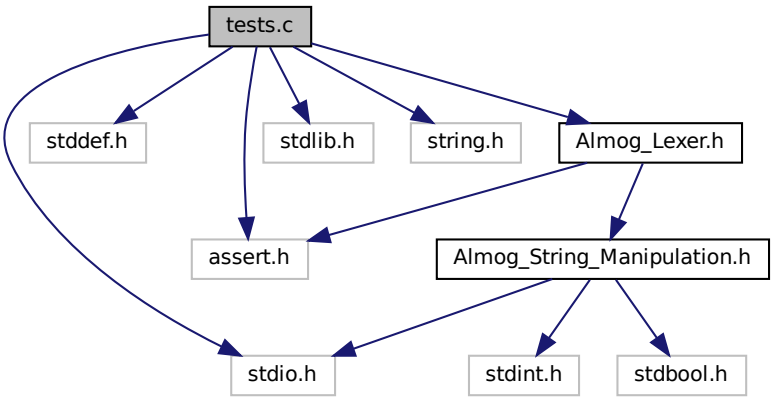
4.8 temp.c

```
00001 #define ALMOG_STRING_MANIPULATION_IMPLEMENTATION
00002 #define ALMOG_LEXER_IMPLEMENTATION
00003 #include "Almog_Lexer.h"
00004 #include "Almog_Dynamic_Array.h"
00005
00006 struct String {
00007   size_t length;
00008   size_t capacity;
00009   char* elements;
00010 };
00011
00012 int main(void)
00013 {
00014     FILE *fp = fopen("./temp.c", "r");
00015
00016     #
00017
00018     struct String string = {0};
00019     ada_init_array(char, string);
00020     char temp_str[ASM_MAX_LEN];
00021     int len = 0;
00022     while ((len = asm_get_line(fp, temp_str)) != EOF) {
00023         for (int i = 0; i < len; i++) {
00024             ada_append(char, string, temp_str[i]);
00025         }
00026         ada_append(char, string, '\n');
00027     }
00028
00029     struct Lexer l = al_lexer_alloc(string.elements, string.length);
00030
00031     struct Token t = al_lexer_next_token(&l);
00032     int num_of_tokens = 0;
00033     while (t.kind != TOKEN_EOF) {
00034         al_token_print(t);
00035         t = al_lexer_next_token(&l);
00036         num_of_tokens++;
00037     }
00038     al_token_print(al_lexer_next_token(&l));
00039
00040     asm_dprintINT(num_of_tokens);
00041
00042     free(string.elements);
00043
00044     return 0;
00045 }
```

4.9 tests.c File Reference

```
#include <assert.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Almog_Lexer.h"
```

Include dependency graph for tests.c:



Macros

- #define ALMOG_STRING_MANIPULATION_IMPLEMENTATION
- #define ALMOG_LEXER_IMPLEMENTATION

Functions

- static const char * kind_name (enum Token_Kind k)
- static void fail_token (const char *test_name, struct Token got, enum Token_Kind exp_kind, const char *exp↩ _text, size_t exp_line, size_t exp_col)
- static void expect_tok (const char *test_name, struct Lexer *l, enum Token_Kind exp_kind, const char *exp↩ _text, size_t exp_line, size_t exp_col)
- static void test_basic_program (void)
- static void test_pp_directive_and_locations (void)
- static void test_whitespace_location_math (void)
- static void test_comments (void)
- static void test_string_and_char_literals (void)
- static void test_literal_operators_longest_match (void)
- static void test_numbers_valid_and_invalid (void)
- static void test_invalid_single_char (void)
- static void test_keyword_vs_identifier_prefix (void)
- static void test_hash_not_pp_directive_when_not_column1 (void)
- static void test_unterminated_block_comment (void)
- static void test_hex_float_variants (void)
- static void test_number_stops_on_invalid_digit_in_base (void)
- static void test_helpers_direct (void)
- int main (void)

4.9.1 Macro Definition Documentation

4.9.1.1 ALMOG_LEXER_IMPLEMENTATION

```
#define ALMOG_LEXER_IMPLEMENTATION
```

Definition at line 16 of file tests.c.

4.9.1.2 ALMOG_STRING_MANIPULATION_IMPLEMENTATION

```
#define ALMOG_STRING_MANIPULATION_IMPLEMENTATION
```

Written by AI

test_almog_lexer.c Simple, self-contained tests for Almog_Lexer.h (single-header).

Definition at line 15 of file tests.c.

4.9.2 Function Documentation

4.9.2.1 expect_tok()

```
static void expect_tok (
            const char * test_name,
            struct Lexer * l,
            enum Token_Kind exp_kind,
            const char * exp_text,
            size_t exp_line,
            size_t exp_col )  [static]
```

Definition at line 123 of file tests.c.

References al_lexer_next_token(), Location::col, fail_token(), Token::kind, Location::line_num, Token::location, Token::text, and Token::text_len.

Referenced by test_basic_program(), test_comments(), test_hash_not_pp_directive_when_not_column1(), test_hex_float_variants(), test_invalid_single_char(), test_keyword_vs_identifier_prefix(), test_literal_operators_longest_match(), test_number_stops_on_invalid_digit_in_base(), test_numbers_valid_and_invalid(), test_pp_directive_and_locations(), test_string_and_char_literals(), test_unterminated_block_comment(), and test_whitespace_location_math().

4.9.2.2 fail_token()

```
static void fail_token (
            const char * test_name,
            struct Token got,
            enum Token_Kind exp_kind,
            const char * exp_text,
            size_t exp_line,
            size_t exp_col )  [static]
```

Definition at line 94 of file tests.c.

References Location::col, Token::kind, kind_name(), Location::line_num, Token::location, Token::text, and Token::text_len.

Referenced by expect_tok().

4.9.2.3 kind_name()

```
static const char* kind_name (
            enum Token_Kind k )  [static]
```

Definition at line 19 of file tests.c.

References 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_GE, TOKEN_GT, TOKEN_HASH, TOKEN_IDENTIFIER, 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_NUMBER, 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 fail_token().

4.9.2.4 main()

```
int main (
            void  )
```

Definition at line 507 of file tests.c.

References test_basic_program(), test_comments(), test_hash_not_pp_directive_when_not_column1(), test_helpers_direct(), test_hex_float_variants(), test_invalid_single_char(), test_keyword_vs_identifier_prefix(), test_literal_operators_longest_match(), test_number_stops_on_invalid_digit_in_base(), test_numbers_valid_and_invalid(), test_pp_directive_and_locations(), test_string_and_char_literals(), test_unterminated_block_comment(), and test_whitespace_location_math().

4.9.2.5 test_basic_program()

```
static void test_basic_program (
            void  )  [static]
```

Definition at line 154 of file tests.c.

References al_lexer_alloc(), expect_tok(), TOKEN_EOF, TOKEN_IDENTIFIER, TOKEN_KEYWORD, TOKEN_LBRACE, TOKEN_LPAREN, TOKEN_NUMBER, TOKEN_RBRACE, TOKEN_RPAREN, and TOKEN_SEMICOLON.

Referenced by main().

4.9.2.6 test_comments()

```
static void test_comments (
            void  )  [static]
```

Definition at line 199 of file tests.c.

References al_lexer_alloc(), expect_tok(), TOKEN_COMMENT, TOKEN_EOF, and TOKEN_IDENTIFIER.

Referenced by main().

4.9.2.7 test_hash_not_pp_directive_when_not_column1()

```
static void test_hash_not_pp_directive_when_not_column1 (
            void  )  [static]
```

Definition at line 392 of file tests.c.

References al_lexer_alloc(), expect_tok(), TOKEN_EOF, TOKEN_HASH, TOKEN_IDENTIFIER, TOKEN_NUMBER, and TOKEN_PP_DIRECTIVE.

Referenced by main().

4.9.2.8 test_helpers_direct()

```
static void test_helpers_direct (
            void  )  [static]
```

Definition at line 465 of file tests.c.

References al_is_identifier(), al_is_identifier_start(), al_lexer_alloc(), al_lexer_chop_char(), al_lexer_chop_while(), al_lexer_peek(), al_lexer_start_with(), AL_UNUSED, asm_isalpha(), Lexer::begining_of_line, Lexer::cursor, and Lexer::line_num.

Referenced by main().

4.9.2.9 test_hex_float_variants()

```
static void test_hex_float_variants (
            void  )  [static]
```

Definition at line 420 of file tests.c.

References al_lexer_alloc(), expect_tok(), TOKEN_EOF, TOKEN_INVALID, and TOKEN_NUMBER.

Referenced by main().

4.9.2.10 test_invalid_single_char()

```
static void test_invalid_single_char (
            void  )  [static]
```

Definition at line 356 of file tests.c.

References al_lexer_alloc(), expect_tok(), TOKEN_EOF, and TOKEN_INVALID.

Referenced by main().

4.9.2.11 test_keyword_vs_identifier_prefix()

```
static void test_keyword_vs_identifier_prefix (
            void  )  [static]
```

Definition at line 366 of file tests.c.

References al_lexer_alloc(), expect_tok(), TOKEN_EOF, TOKEN_IDENTIFIER, and TOKEN_KEYWORD.

Referenced by main().

4.9.2.12 test_literal_operators_longest_match()

```
static void test_literal_operators_longest_match (
            void  )  [static]
```

Definition at line 237 of file tests.c.

References al_lexer_alloc(), expect_tok(), TOKEN_ANDAND, TOKEN_ANDEQ, TOKEN_ARROW, TOKEN_BANG, TOKEN_BITAND, TOKEN_BITOR, TOKEN_BSLASH, TOKEN_CARET, TOKEN_COLON, TOKEN_COMMA, TOKEN_DOT, TOKEN_ELLIPSIS, TOKEN_EOF, TOKEN_EQ, TOKEN_EQEQ, TOKEN_GE, TOKEN_GT, 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_QUESTION, TOKEN_RBRACE, TOKEN_RBRACKET, TOKEN_RPAREN, TOKEN_RSHIFT, TOKEN_RSHIFTEQ, TOKEN_SEMICOLON, TOKEN_SLASH, TOKEN_SLASHEQ, TOKEN_STAR, TOKEN_STAREQ, TOKEN_TILDE, and TOKEN_XOREQ.

Referenced by main().

4.9.2.13 test_number_stops_on_invalid_digit_in_base()

```
static void test_number_stops_on_invalid_digit_in_base (
            void  )  [static]
```

Definition at line 446 of file tests.c.

References al_lexer_alloc(), expect_tok(), TOKEN_EOF, and TOKEN_NUMBER.

Referenced by main().

4.9.2.14 test_numbers_valid_and_invalid()

```
static void test_numbers_valid_and_invalid (
            void  )  [static]
```

Definition at line 306 of file tests.c.

References al_lexer_alloc(), expect_tok(), TOKEN_DOT, TOKEN_EOF, TOKEN_INVALID, and TOKEN_NUMBER.

Referenced by main().

4.9.2.15 test_pp_directive_and_locations()

```
static void test_pp_directive_and_locations (
            void  )  [static]
```

Definition at line 172 of file tests.c.

References al_lexer_alloc(), expect_tok(), TOKEN_EOF, TOKEN_IDENTIFIER, TOKEN_KEYWORD, TOKEN_PP_DIRECTIVE, and TOKEN_SEMICOLON.

Referenced by main().

4.9.2.16 test_string_and_char_literals()

```
static void test_string_and_char_literals (
            void  )  [static]
```

Definition at line 223 of file tests.c.

References al_lexer_alloc(), expect_tok(), TOKEN_CHAR_LIT, TOKEN_EOF, and TOKEN_STRING_LIT.

Referenced by main().

4.9.2.17 test_unterminated_block_comment()

```
static void test_unterminated_block_comment (
            void  )  [static]
```

Definition at line 409 of file tests.c.

References al_lexer_alloc(), expect_tok(), TOKEN_COMMENT, and TOKEN_EOF.

Referenced by main().

4.9.2.18 test_whitespace_location_math()

```
static void test_whitespace_location_math (
            void  )  [static]
```

Definition at line 188 of file tests.c.

References al_lexer_alloc(), expect_tok(), TOKEN_EOF, and TOKEN_IDENTIFIER.

Referenced by main().

4.10 tests.c

```
00001
00008 #include <assert.h>
00009 #include <stddef.h>
00010 #include <stdio.h>
00011 #include <stdlib.h>
00012 #include <string.h>
00013
00014 /* Compile implementations in THIS translation unit. */
00015 #define ALMOG_STRING_MANIPULATION_IMPLEMENTATION
00016 #define ALMOG_LEXER_IMPLEMENTATION
00017 #include "Almog_Lexer.h"
00018
00019 static const char *kind_name(enum Token_Kind k)
00020 {
00021     switch (k) {
00022         case TOKEN_EOF: return "TOKEN_EOF";
00023         case TOKEN_INVALID: return "TOKEN_INVALID";
00024         case TOKEN_PP_DIRECTIVE: return "TOKEN_PP_DIRECTIVE";
00025         case TOKEN_COMMENT: return "TOKEN_COMMENT";
00026         case TOKEN_STRING_LIT: return "TOKEN_STRING_LIT";
00027         case TOKEN_CHAR_LIT: return "TOKEN_CHAR_LIT";
00028         case TOKEN_NUMBER: return "TOKEN_NUMBER";
00029         case TOKEN_KEYWORD: return "TOKEN_KEYWORD";
00030         case TOKEN_IDENTIFIER: return "TOKEN_IDENTIFIER";
00031
00032         case TOKEN_LPAREN: return "TOKEN_LPAREN";
00033         case TOKEN_RPAREN: return "TOKEN_RPAREN";
00034         case TOKEN_LBRACKET: return "TOKEN_LBRACKET";
00035         case TOKEN_RBRACKET: return "TOKEN_RBRACKET";
00036         case TOKEN_LBRACE: return "TOKEN_LBRACE";
00037         case TOKEN_RBRACE: return "TOKEN_RBRACE";
00038
00039         case TOKEN_DOT: return "TOKEN_DOT";
00040         case TOKEN_COMMA: return "TOKEN_COMMA";
00041         case TOKEN_SEMICOLON: return "TOKEN_SEMICOLON";
00042         case TOKEN_BSLASH: return "TOKEN_BSLASH";
00043         case TOKEN_HASH: return "TOKEN_HASH";
00044
00045         case TOKEN_QUESTION: return "TOKEN_QUESTION";
00046         case TOKEN_COLON: return "TOKEN_COLON";
00047
00048         case TOKEN_EQ: return "TOKEN_EQ";
00049         case TOKEN_EQEQ: return "TOKEN_EQEQ";
```

```
00050         case TOKEN_NE: return "TOKEN_NE";
00051         case TOKEN_BANG: return "TOKEN_BANG";
00052
00053         case TOKEN_LT: return "TOKEN_LT";
00054         case TOKEN_GT: return "TOKEN_GT";
00055         case TOKEN_LE: return "TOKEN_LE";
00056         case TOKEN_GE: return "TOKEN_GE";
00057
00058         case TOKEN_BITAND: return "TOKEN_BITAND";
00059         case TOKEN_ANDAND: return "TOKEN_ANDAND";
00060         case TOKEN_BITOR: return "TOKEN_BITOR";
00061         case TOKEN_OROR: return "TOKEN_OROR";
00062         case TOKEN_CARET: return "TOKEN_CARET";
00063         case TOKEN_TILDE: return "TOKEN_TILDE";
00064
00065         case TOKEN_LSHIFT: return "TOKEN_LSHIFT";
00066         case TOKEN_RSHIFT: return "TOKEN_RSHIFT";
00067
00068         case TOKEN_PLUSPLUS: return "TOKEN_PLUSPLUS";
00069         case TOKEN_MINUSMINUS: return "TOKEN_MINUSMINUS";
00070
00071         case TOKEN_PLUS: return "TOKEN_PLUS";
00072         case TOKEN_MINUS: return "TOKEN_MINUS";
00073         case TOKEN_STAR: return "TOKEN_STAR";
00074         case TOKEN_SLASH: return "TOKEN_SLASH";
00075         case TOKEN_PERCENT: return "TOKEN_PERCENT";
00076
00077         case TOKEN_PLUSEQ: return "TOKEN_PLUSEQ";
00078         case TOKEN_MINUSEQ: return "TOKEN_MINUSEQ";
00079         case TOKEN_STAREQ: return "TOKEN_STAREQ";
00080         case TOKEN_SLASHEQ: return "TOKEN_SLASHEQ";
00081         case TOKEN_PERCENTEQ: return "TOKEN_PERCENTEQ";
00082         case TOKEN_ANDEQ: return "TOKEN_ANDEQ";
00083         case TOKEN_OREQ: return "TOKEN_OREQ";
00084         case TOKEN_XOREQ: return "TOKEN_XOREQ";
00085         case TOKEN_LSHIFTEQ: return "TOKEN_LSHIFTEQ";
00086         case TOKEN_RSHIFTEQ: return "TOKEN_RSHIFTEQ";
00087
00088         case TOKEN_ARROW: return "TOKEN_ARROW";
00089         case TOKEN_ELLIPSIS: return "TOKEN_ELLIPSIS";
00090     }
00091     return "TOKEN_<unknown>";
00092 }
00093
00094 static void fail_token(
00095     const char *test_name,
00096     struct Token got,
00097     enum Token_Kind exp_kind,
00098     const char *exp_text,
00099     size_t exp_line,
00100     size_t exp_col
00101 )
00102 {
00103     fprintf(stderr, "\n[FAIL] %s\n", test_name);
00104     fprintf(stderr, "  expected: kind=%s", kind_name(exp_kind));
00105     if (exp_text) {
00106         fprintf(stderr, ", text=\"%s\" (len=%zu)", exp_text, strlen(exp_text));
00107     }
00108     if (exp_line) fprintf(stderr, ", line=%zu", exp_line);
00109     if (exp_col) fprintf(stderr, ", col=%zu", exp_col);
00110     fprintf(stderr, "\n");
00111
00112     fprintf(stderr, "  got:      kind=%s, text_len=%zu, line=%zu, col=%zu, text=\"%.*s\"\n",
00113             kind_name(got.kind),
00114             got.text_len,
00115             got.location.line_num,
00116             got.location.col,
00117             (int)got.text_len,
00118             got.text ? got.text : "");
00119     exit(1);
00120 }
00121
00122 /* If exp_text == NULL => don't check text. If exp_line/col == 0 => don't check. */
00123 static void expect_tok(
00124     const char *test_name,
00125     struct Lexer *l,
00126     enum Token_Kind exp_kind,
00127     const char *exp_text,
00128     size_t exp_line,
00129     size_t exp_col
00130 )
00131 {
00132     struct Token t = al_lexer_next_token(l);
00133
00134     if (t.kind != exp_kind) {
00135         fail_token(test_name, t, exp_kind, exp_text, exp_line, exp_col);
00136     }
```

```
00137
00138     if (exp_text) {
00139         size_t n = strlen(exp_text);
00140         if (t.text_len != n || memcmp(t.text, exp_text, n) != 0) {
00141             fail_token(test_name, t, exp_kind, exp_text, exp_line, exp_col);
00142         }
00143     }
00144
00145     if (exp_line && t.location.line_num != exp_line) {
00146         fail_token(test_name, t, exp_kind, exp_text, exp_line, exp_col);
00147     }
00148
00149     if (exp_col && t.location.col != exp_col) {
00150         fail_token(test_name, t, exp_kind, exp_text, exp_line, exp_col);
00151     }
00152 }
00153
00154 static void test_basic_program(void)
00155 {
00156     const char *name = "basic_program";
00157     const char *src = "int main() { return 0; }";
00158     struct Lexer l = al_lexer_alloc(src, strlen(src));
00159
00160     expect_tok(name, &l, TOKEN_KEYWORD, "int", 0, 0);
00161     expect_tok(name, &l, TOKEN_IDENTIFIER, "main", 0, 0);
00162     expect_tok(name, &l, TOKEN_LPAREN, "(", 0, 0);
00163     expect_tok(name, &l, TOKEN_RPAREN, ")", 0, 0);
00164     expect_tok(name, &l, TOKEN_LBRACE, "{", 0, 0);
00165     expect_tok(name, &l, TOKEN_KEYWORD, "return", 0, 0);
00166     expect_tok(name, &l, TOKEN_NUMBER, "0", 0, 0);
00167     expect_tok(name, &l, TOKEN_SEMICOLON, ";", 0, 0);
00168     expect_tok(name, &l, TOKEN_RBRACE, "}", 0, 0);
00169     expect_tok(name, &l, TOKEN_EOF, NULL, 0, 0);
00170 }
00171
00172 static void test_pp_directive_and_locations(void)
00173 {
00174     const char *name = "pp_directive_and_locations";
00175     const char *src = "#include <stdio.h>\nint x;\n";
00176     struct Lexer l = al_lexer_alloc(src, strlen(src));
00177
00178     /* PP directive is only recognized at col==1 and includes the newline. */
00179     expect_tok(name, &l, TOKEN_PP_DIRECTIVE, "#include <stdio.h>\n", 1, 1);
00180
00181     expect_tok(name, &l, TOKEN_KEYWORD, "int", 2, 1);
00182     expect_tok(name, &l, TOKEN_IDENTIFIER, "x", 2, 5);
00183     expect_tok(name, &l, TOKEN_SEMICOLON, ";", 2, 6);
00184
00185     expect_tok(name, &l, TOKEN_EOF, NULL, 0, 0);
00186 }
00187
00188 static void test_whitespace_location_math(void)
00189 {
00190     const char *name = "whitespace_location";
00191     const char *src = "a\n  b";
00192     struct Lexer l = al_lexer_alloc(src, strlen(src));
00193
00194     expect_tok(name, &l, TOKEN_IDENTIFIER, "a", 1, 1);
00195     expect_tok(name, &l, TOKEN_IDENTIFIER, "b", 2, 3);
00196     expect_tok(name, &l, TOKEN_EOF, NULL, 0, 0);
00197 }
00198
00199 static void test_comments(void)
00200 {
00201     {
00202         const char *name = "line_comment_includes_newline";
00203         const char *src = "// hello\nx";
00204         struct Lexer l = al_lexer_alloc(src, strlen(src));
00205
00206         expect_tok(name, &l, TOKEN_COMMENT, "// hello\n", 1, 1);
00207         expect_tok(name, &l, TOKEN_IDENTIFIER, "x", 2, 1);
00208         expect_tok(name, &l, TOKEN_EOF, NULL, 0, 0);
00209     }
00210
00211     {
00212         const char *name = "block_comment_updates_line_col";
00213         const char *src = "/*x\ny*/z";
00214         struct Lexer l = al_lexer_alloc(src, strlen(src));
00215
00216         expect_tok(name, &l, TOKEN_COMMENT, "/*x\ny*/", 1, 1);
00217         /* After the newline inside the comment, 'z' should be on line 2. */
00218         expect_tok(name, &l, TOKEN_IDENTIFIER, "z", 2, 4);
00219         expect_tok(name, &l, TOKEN_EOF, NULL, 0, 0);
00220     }
00221 }
00222
00223 static void test_string_and_char_literals(void)
```

```
00224 {
00225     const char *name = "string_and_char_literals";
00226     const char *src = "\"abc\" 'x' \"unterminated\n";
00227     struct Lexer l = al_lexer_alloc(src, strlen(src));
00228
00229     expect_tok(name, &l, TOKEN_STRING_LIT, "\"abc\"", 0, 0);
00230     expect_tok(name, &l, TOKEN_CHAR_LIT, "'x'", 0, 0);
00231
00232     /* Lexer stops string literal on '\n' if not closed. Still TOKEN_STRING_LIT. */
00233     expect_tok(name, &l, TOKEN_STRING_LIT, "\"unterminated", 0, 0);
00234     expect_tok(name, &l, TOKEN_EOF, NULL, 0, 0);
00235 }
00236
00237 static void test_literal_operators_longest_match(void)
00238 {
00239     const char *name = "literal_operators_longest_match";
00240     const char *src =
00241         "( ) [ ] { } ... . , ? : == != = ! ; \\ -> "
00242         "> >= < <= «= »= ++ -- « » "
00243         "+= -= *= /= %= &= |= ^= || && | & ^ ~ "
00244         "+ - * / %";
00245     struct Lexer l = al_lexer_alloc(src, strlen(src));
00246
00247     expect_tok(name, &l, TOKEN_LPAREN, "(", 0, 0);
00248     expect_tok(name, &l, TOKEN_RPAREN, ")", 0, 0);
00249     expect_tok(name, &l, TOKEN_LBRACKET, "[", 0, 0);
00250     expect_tok(name, &l, TOKEN_RBRACKET, "]", 0, 0);
00251     expect_tok(name, &l, TOKEN_LBRACE, "{", 0, 0);
00252     expect_tok(name, &l, TOKEN_RBRACE, "}", 0, 0);
00253
00254     expect_tok(name, &l, TOKEN_ELLIPSIS, "...", 0, 0);
00255     expect_tok(name, &l, TOKEN_DOT, ".", 0, 0);
00256     expect_tok(name, &l, TOKEN_COMMA, ",", 0, 0);
00257     expect_tok(name, &l, TOKEN_QUESTION, "?", 0, 0);
00258     expect_tok(name, &l, TOKEN_COLON, ":", 0, 0);
00259
00260     expect_tok(name, &l, TOKEN_EQEQ, "==", 0, 0);
00261     expect_tok(name, &l, TOKEN_NE, "!=", 0, 0);
00262     expect_tok(name, &l, TOKEN_EQ, "=", 0, 0);
00263     expect_tok(name, &l, TOKEN_BANG, "!", 0, 0);
00264     expect_tok(name, &l, TOKEN_SEMICOLON, ";", 0, 0);
00265     expect_tok(name, &l, TOKEN_BSLASH, "\\", 0, 0);
00266     expect_tok(name, &l, TOKEN_ARROW, "->", 0, 0);
00267
00268     expect_tok(name, &l, TOKEN_GT, ">", 0, 0);
00269     expect_tok(name, &l, TOKEN_GE, ">=", 0, 0);
00270     expect_tok(name, &l, TOKEN_LT, "<", 0, 0);
00271     expect_tok(name, &l, TOKEN_LE, "<=", 0, 0);
00272
00273     expect_tok(name, &l, TOKEN_LSHIFTEQ, "«=", 0, 0);
00274     expect_tok(name, &l, TOKEN_RSHIFTEQ, "»=", 0, 0);
00275
00276     expect_tok(name, &l, TOKEN_PLUSPLUS, "++", 0, 0);
00277     expect_tok(name, &l, TOKEN_MINUSMINUS, "--", 0, 0);
00278     expect_tok(name, &l, TOKEN_LSHIFT, "«", 0, 0);
00279     expect_tok(name, &l, TOKEN_RSHIFT, "»", 0, 0);
00280
00281     expect_tok(name, &l, TOKEN_PLUSEQ, "+=", 0, 0);
00282     expect_tok(name, &l, TOKEN_MINUSEQ, "-=", 0, 0);
00283     expect_tok(name, &l, TOKEN_STAREQ, "*=", 0, 0);
00284     expect_tok(name, &l, TOKEN_SLASHEQ, "/=", 0, 0);
00285     expect_tok(name, &l, TOKEN_PERCENTEQ, "%=", 0, 0);
00286     expect_tok(name, &l, TOKEN_ANDEQ, "&=", 0, 0);
00287     expect_tok(name, &l, TOKEN_OREQ, "|=", 0, 0);
00288     expect_tok(name, &l, TOKEN_XOREQ, "^=", 0, 0);
00289
00290     expect_tok(name, &l, TOKEN_OROR, "||", 0, 0);
00291     expect_tok(name, &l, TOKEN_ANDAND, "&&", 0, 0);
00292     expect_tok(name, &l, TOKEN_BITOR, "|", 0, 0);
00293     expect_tok(name, &l, TOKEN_BITAND, "&", 0, 0);
00294     expect_tok(name, &l, TOKEN_CARET, "^", 0, 0);
00295     expect_tok(name, &l, TOKEN_TILDE, "~", 0, 0);
00296
00297     expect_tok(name, &l, TOKEN_PLUS, "+", 0, 0);
00298     expect_tok(name, &l, TOKEN_MINUS, "-", 0, 0);
00299     expect_tok(name, &l, TOKEN_STAR, "*", 0, 0);
00300     expect_tok(name, &l, TOKEN_SLASH, "/", 0, 0);
00301     expect_tok(name, &l, TOKEN_PERCENT, "%", 0, 0);
00302
00303     expect_tok(name, &l, TOKEN_EOF, NULL, 0, 0);
00304 }
00305
00306 static void test_numbers_valid_and_invalid(void)
00307 {
00308     const char *name = "numbers_valid_and_invalid";
00309     const char *src =
00310         "0 123 1. .5 1.5 "
```

```
00311         "1e3 1e+3 1e-3 1e 1e+ "
00312         "0xFF 0x1.fp3 0x1.fp 0x "
00313         "0b1011 0b "
00314         "0o77 0o "
00315         "42u 42ULL "
00316         "3.14f 2.0L "
00317         ". .0";
00318     struct Lexer l = al_lexer_alloc(src, strlen(src));
00319
00320     expect_tok(name, &l, TOKEN_NUMBER, "0", 0, 0);
00321     expect_tok(name, &l, TOKEN_NUMBER, "123", 0, 0);
00322     expect_tok(name, &l, TOKEN_NUMBER, "1.", 0, 0);
00323     expect_tok(name, &l, TOKEN_NUMBER, ".5", 0, 0);
00324     expect_tok(name, &l, TOKEN_NUMBER, "1.5", 0, 0);
00325
00326     expect_tok(name, &l, TOKEN_NUMBER, "1e3", 0, 0);
00327     expect_tok(name, &l, TOKEN_NUMBER, "1e+3", 0, 0);
00328     expect_tok(name, &l, TOKEN_NUMBER, "1e-3", 0, 0);
00329     expect_tok(name, &l, TOKEN_INVALID, "1e", 0, 0);
00330     expect_tok(name, &l, TOKEN_INVALID, "1e+", 0, 0);
00331
00332     expect_tok(name, &l, TOKEN_NUMBER, "0xFF", 0, 0);
00333     expect_tok(name, &l, TOKEN_NUMBER, "0x1.fp3", 0, 0);
00334     expect_tok(name, &l, TOKEN_INVALID, "0x1.fp", 0, 0);
00335     expect_tok(name, &l, TOKEN_INVALID, "0x", 0, 0);
00336
00337     expect_tok(name, &l, TOKEN_NUMBER, "0b1011", 0, 0);
00338     expect_tok(name, &l, TOKEN_INVALID, "0b", 0, 0);
00339
00340     expect_tok(name, &l, TOKEN_NUMBER, "0o77", 0, 0);
00341     expect_tok(name, &l, TOKEN_INVALID, "0o", 0, 0);
00342
00343     expect_tok(name, &l, TOKEN_NUMBER, "42u", 0, 0);
00344     expect_tok(name, &l, TOKEN_NUMBER, "42ULL", 0, 0);
00345
00346     expect_tok(name, &l, TOKEN_NUMBER, "3.14f", 0, 0);
00347     expect_tok(name, &l, TOKEN_NUMBER, "2.0L", 0, 0);
00348
00349     /* '.' alone should be DOT, but '.0' should be NUMBER. */
00350     expect_tok(name, &l, TOKEN_DOT, ".", 0, 0);
00351     expect_tok(name, &l, TOKEN_NUMBER, ".0", 0, 0);
00352
00353     expect_tok(name, &l, TOKEN_EOF, NULL, 0, 0);
00354 }
00355
00356 static void test_invalid_single_char(void)
00357 {
00358     const char *name = "invalid_single_char";
00359     const char *src = "@";
00360     struct Lexer l = al_lexer_alloc(src, strlen(src));
00361
00362     expect_tok(name, &l, TOKEN_INVALID, "@", 1, 1);
00363     expect_tok(name, &l, TOKEN_EOF, NULL, 0, 0);
00364 }
00365
00366 static void test_keyword_vs_identifier_prefix(void)
00367 {
00368     const char *name = "keyword_vs_identifier_prefix";
00369     const char *src =
00370         "int intensity return return_ goto goto1 _x x_1 __ __9 a9 _9";
00371     struct Lexer l = al_lexer_alloc(src, strlen(src));
00372
00373     expect_tok(name, &l, TOKEN_KEYWORD, "int", 0, 0);
00374     expect_tok(name, &l, TOKEN_IDENTIFIER, "intensity", 0, 0);
00375
00376     expect_tok(name, &l, TOKEN_KEYWORD, "return", 0, 0);
00377     expect_tok(name, &l, TOKEN_IDENTIFIER, "return_", 0, 0);
00378
00379     expect_tok(name, &l, TOKEN_KEYWORD, "goto", 0, 0);
00380     expect_tok(name, &l, TOKEN_IDENTIFIER, "goto1", 0, 0);
00381
00382     expect_tok(name, &l, TOKEN_IDENTIFIER, "_x", 0, 0);
00383     expect_tok(name, &l, TOKEN_IDENTIFIER, "x_1", 0, 0);
00384     expect_tok(name, &l, TOKEN_IDENTIFIER, "__", 0, 0);
00385     expect_tok(name, &l, TOKEN_IDENTIFIER, "__9", 0, 0);
00386     expect_tok(name, &l, TOKEN_IDENTIFIER, "a9", 0, 0);
00387     expect_tok(name, &l, TOKEN_IDENTIFIER, "_9", 0, 0);
00388
00389     expect_tok(name, &l, TOKEN_EOF, NULL, 0, 0);
00390 }
00391
00392 static void test_hash_not_pp_directive_when_not_column1(void)
00393 {
00394     const char *name = "hash_not_pp_directive_when_not_column1";
00395     const char *src = "  #define X 1\n#define Y 2\n";
00396     struct Lexer l = al_lexer_alloc(src, strlen(src));
00397
```

```
00398     /* Because of leading spaces, '#' is not at col 1 => NOT a PP directive. */
00399     expect_tok(name, &l, TOKEN_HASH, "#", 1, 3);
00400     expect_tok(name, &l, TOKEN_IDENTIFIER, "define", 1, 4);
00401     expect_tok(name, &l, TOKEN_IDENTIFIER, "X", 1, 11);
00402     expect_tok(name, &l, TOKEN_NUMBER, "1", 1, 13);
00403
00404     /* This one is at col 1 and should be treated as a directive (includes '\n'). */
00405     expect_tok(name, &l, TOKEN_PP_DIRECTIVE, "#define Y 2\n", 2, 1);
00406     expect_tok(name, &l, TOKEN_EOF, NULL, 0, 0);
00407 }
00408
00409 static void test_unterminated_block_comment(void)
00410 {
00411     const char *name = "unterminated_block_comment";
00412     const char *src = "/* unterminated";
00413     struct Lexer l = al_lexer_alloc(src, strlen(src));
00414
00415     /* Lexer consumes to EOF and still returns TOKEN_COMMENT. */
00416     expect_tok(name, &l, TOKEN_COMMENT, "/* unterminated", 1, 1);
00417     expect_tok(name, &l, TOKEN_EOF, NULL, 0, 0);
00418 }
00419
00420 static void test_hex_float_variants(void)
00421 {
00422     const char *name = "hex_float_variants";
00423     const char *src =
00424         "0x1p2 0x1p+2 0x1p-2 0x.1p1 0x.p1 0xp1 0x1.0p0 0x1.0 0x1";
00425     struct Lexer l = al_lexer_alloc(src, strlen(src));
00426
00427     expect_tok(name, &l, TOKEN_NUMBER, "0x1p2", 0, 0);
00428     expect_tok(name, &l, TOKEN_NUMBER, "0x1p+2", 0, 0);
00429     expect_tok(name, &l, TOKEN_NUMBER, "0x1p-2", 0, 0);
00430     expect_tok(name, &l, TOKEN_NUMBER, "0x.1p1", 0, 0);
00431
00432     /* Invalid: dot in hex mantissa but no digits before/after the dot */
00433     expect_tok(name, &l, TOKEN_INVALID, "0x.p1", 0, 0);
00434     /* Invalid: no mantissa digits (even though exponent is present) */
00435     expect_tok(name, &l, TOKEN_INVALID, "0xp1", 0, 0);
00436
00437     expect_tok(name, &l, TOKEN_NUMBER, "0x1.0p0", 0, 0);
00438     /* Invalid: '.' in hex mantissa requires p/P exponent in this lexer */
00439     expect_tok(name, &l, TOKEN_INVALID, "0x1.0", 0, 0);
00440     /* Plain hex integer is valid */
00441     expect_tok(name, &l, TOKEN_NUMBER, "0x1", 0, 0);
00442
00443     expect_tok(name, &l, TOKEN_EOF, NULL, 0, 0);
00444 }
00445
00446 static void test_number_stops_on_invalid_digit_in_base(void)
00447 {
00448     const char *name = "number_stops_on_invalid_digit_in_base";
00449     const char *src = "0b102 0o78";
00450     struct Lexer l = al_lexer_alloc(src, strlen(src));
00451
00452     /*
00453         Current behavior: it tokenizes the longest valid prefix for the base.
00454         This test documents that behavior (rather than forcing it to be invalid).
00455     */
00456     expect_tok(name, &l, TOKEN_NUMBER, "0b10", 0, 0);
00457     expect_tok(name, &l, TOKEN_NUMBER, "2", 0, 0);
00458
00459     expect_tok(name, &l, TOKEN_NUMBER, "0o7", 0, 0);
00460     expect_tok(name, &l, TOKEN_NUMBER, "8", 0, 0);
00461
00462     expect_tok(name, &l, TOKEN_EOF, NULL, 0, 0);
00463 }
00464
00465 static void test_helpers_direct(void)
00466 {
00467     const char *name = "helpers_direct";
00468     AL_UNUSED(name);
00469
00470     /* al_is_identifier / al_is_identifier_start */
00471     assert(al_is_identifier_start('_'));
00472     assert(al_is_identifier('_'));
00473     assert(al_is_identifier('a'));
00474     assert(al_is_identifier('Z'));
00475     assert(al_is_identifier('9'));
00476     assert(!al_is_identifier_start('9'));
00477
00478     /* al_lexer_start_with: empty prefix path */
00479     {
00480         struct Lexer l = al_lexer_alloc("abc123", 6);
00481         assert(al_lexer_start_with(&l, ""));
00482         assert(al_lexer_start_with(&l, "ab"));
00483         assert(!al_lexer_start_with(&l, "abcd"));
00484     }
```

```
00485
00486     /* al_lexer_chop_while + al_lexer_peek */
00487     {
00488         struct Lexer l = al_lexer_alloc("abc123", 6);
00489         al_lexer_chop_while(&l, asm_isalpha);
00490         assert(l.cursor == 3);
00491         assert(al_lexer_peek(&l, 0) == '1');
00492         assert(al_lexer_peek(&l, 100) == '\0');
00493     }
00494
00495     /* al_lexer_chop_char newline bookkeeping */
00496     {
00497         struct Lexer l = al_lexer_alloc("x\ny", 3);
00498         assert(l.line_num == 0);
00499         assert(l.begining_of_line == 0);
00500         (void)al_lexer_chop_char(&l); /* 'x' */
00501         (void)al_lexer_chop_char(&l); /* '\n' */
00502         assert(l.line_num == 1);
00503         assert(l.begining_of_line == 2);
00504     }
00505 }
00506
00507 int main(void)
00508 {
00509     test_basic_program();
00510     test_pp_directive_and_locations();
00511     test_whitespace_location_math();
00512     test_comments();
00513     test_string_and_char_literals();
00514     test_literal_operators_longest_match();
00515     test_numbers_valid_and_invalid();
00516     test_keyword_vs_identifier_prefix();
00517     test_hash_not_pp_directive_when_not_column1();
00518     test_unterminated_block_comment();
00519     test_hex_float_variants();
00520     test_number_stops_on_invalid_digit_in_base();
00521     test_helpers_direct();
00522     test_invalid_single_char();
00523
00524     printf("All lexer tests passed.\n");
00525     return 0;
00526 }
```

Index