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:

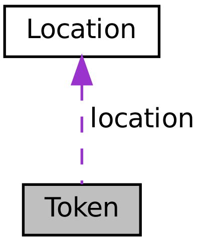
- temp.c

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

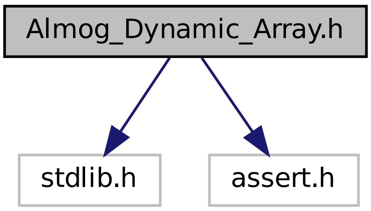
File Documentation

4.1 Almog_Dynamic_Array.h File Reference

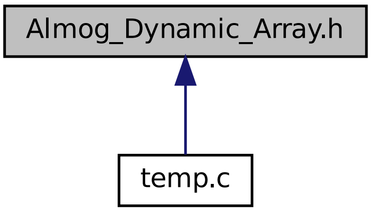
Header-only C macros that implement a simple dynamic array.

```
#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

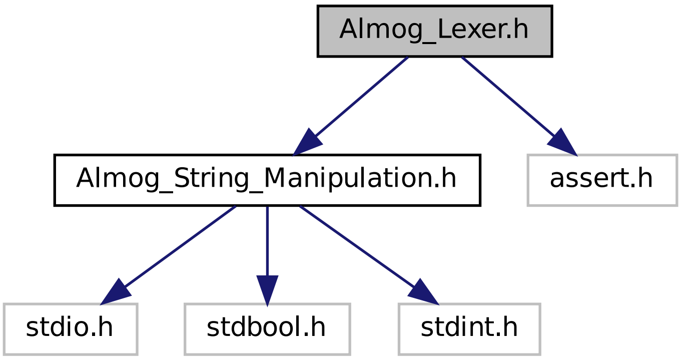
```
#ifndef ALMOG_DYNAMIC_ARRAY_H_
#define ALMOG_DYNAMIC_ARRAY_H_



#define ADA_INIT_CAPACITY 10

#ifndef ADA_MALLOC
#include <stdlib.h>
#define ADA_MALLOC malloc
#endif /*ADA_MALLOC*/

#ifndef ADA_REALLOC
#include <stdlib.h>
#define ADA_REALLOC realloc
#endif /*ADA_REALLOC*/

#ifndef ADA_ASSERT
#include <assert.h>
#define ADA_ASSERT assert
#endif /*ADA_ASSERT*/

/* typedef struct {
    size_t length;
    size_t capacity;
    int* elements;
} ada_int_array; */

#define ada_init_array(type, header) 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)

#define ada_resize(type, header, new_capacity) 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)

#define ada_append(type, header, 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)

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


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

#define ada_remove(type, header, index) 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)

#define ada_remove_unordered(type, header, index) 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)


#endif /*ALMOG_DYNAMIC_ARRAY_H_*/
```

4.3 Almog_Lexer.h File Reference

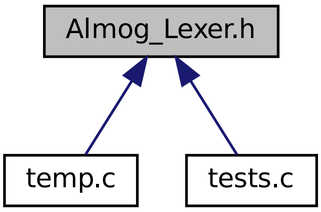
A small single-header lexer for C/C++-like source text.

```
#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

```

#ifndef ALMOG_LEXER_H_
#define ALMOG_LEXER_H_

#include "Almog_String_Manipulation.h"

#ifndef AL_ASSERT
#include <assert.h>
#define AL_ASSERT assert
#endif /* AL_ASSERT */

enum Token_Kind {
    /* Sentinel / unknown */
    TOKEN_EOF,
    TOKEN_INVALID,

    /* High-level / multi-char / "word-like" */
    TOKEN_PP_DIRECTIVE,
    TOKEN_COMMENT,
    TOKEN_STRING_LIT,
    TOKEN_CHAR_LIT,
    TOKEN_NUMBER,
    TOKEN_KEYWORD,
    TOKEN_IDENTIFIER,


    /* Grouping / separators */
    TOKEN_LPAREN,
    TOKEN_RPAREN,
    TOKEN_LBRACKET,
    TOKEN_RBRACKET,
    TOKEN_LBRACE,
    TOKEN_RBRACE,

    /* Punctuation */
    TOKEN_DOT,
    TOKEN_COMMA,
    TOKEN_SEMICOLON,
    TOKEN_BSLASH,
    TOKEN_HASH,

    /* Ternary */
    TOKEN_QUESTION,
    TOKEN_COLON,

    /* Assignment / equality */
    TOKEN_EQ,
    TOKEN_EQEQ,
    TOKEN_NE,
    TOKEN_BANG,

    /* Relational */
    TOKEN_LT,
    TOKEN_GT,
    TOKEN_LE,
    TOKEN_GE,

    /* Bitwise / boolean */
    TOKEN_BITAND,
    TOKEN_ANDAND,
    TOKEN_BITOR,
    TOKEN_OROR,
    /* Bitwise unary */
    TOKEN_CARET,
    TOKEN_TILDE,

    /* Shifts */
    TOKEN_LSHIFT,
    TOKEN_RSHIFT,

    /* Inc / dec */
    TOKEN_PLUSPLUS,
    TOKEN_MINUSMINUS,

    /* Arithmetic */
    TOKEN_PLUS,
    TOKEN_MINUS,
    TOKEN_STAR,
    TOKEN_SLASH,
    TOKEN_PERCENT,

    /* Compound assignment */
    TOKEN_PLUSEQ,
    TOKEN_MINUSEQ,
    TOKEN_STAREQ,
    TOKEN_SLASHEQ,
    TOKEN_PERCENTEQ,
    TOKEN_ANDEQ,
    TOKEN_OREQ,
    TOKEN_XOREQ,
    TOKEN_LSHIFTEQ,
    TOKEN_RSHIFTEQ,

    /* Member access / varargs */
    TOKEN_ARROW,
    TOKEN_ELLIPSIS,
};

struct Literal_Token {
    enum Token_Kind kind;
    const char * const text;
};

static struct Literal_Token literal_tokens[] = {
    {.text = "("  , .kind = TOKEN_LPAREN},
    {.text = ")"  , .kind = TOKEN_RPAREN},
    {.text = "["  , .kind = TOKEN_LBRACKET},
    {.text = "]"  , .kind = TOKEN_RBRACKET},
    {.text = "{"  , .kind = TOKEN_LBRACE},
    {.text = "}"  , .kind = TOKEN_RBRACE},
    {.text = "#"  , .kind = TOKEN_HASH},
    {.text = "...", .kind = TOKEN_ELLIPSIS},
    {.text = "."  , .kind = TOKEN_DOT},
    {.text = ","  , .kind = TOKEN_COMMA},
    {.text = "?"  , .kind = TOKEN_QUESTION},
    {.text = ":"  , .kind = TOKEN_COLON},
    {.text = "==" , .kind = TOKEN_EQEQ},
    {.text = "!=" , .kind = TOKEN_NE},
    {.text = "="  , .kind = TOKEN_EQ},
    {.text = "!"  , .kind = TOKEN_BANG},
    {.text = ";"  , .kind = TOKEN_SEMICOLON},
    {.text = "\\" , .kind = TOKEN_BSLASH},
    {.text = "->" , .kind = TOKEN_ARROW},
    {.text = ">"  , .kind = TOKEN_GT},
    {.text = ">=" , .kind = TOKEN_GE},
    {.text = "<"  , .kind = TOKEN_LT},
    {.text = "<=" , .kind = TOKEN_LE},
    {.text = "«=", .kind = TOKEN_LSHIFTEQ},
    {.text = "»=", .kind = TOKEN_RSHIFTEQ},
    {.text = "++" , .kind = TOKEN_PLUSPLUS},
    {.text = "--" , .kind = TOKEN_MINUSMINUS},
    {.text = "«" , .kind = TOKEN_LSHIFT},
    {.text = "»" , .kind = TOKEN_RSHIFT},
    {.text = "+=", .kind = TOKEN_PLUSEQ},
    {.text = "-=", .kind = TOKEN_MINUSEQ},
    {.text = "*=", .kind = TOKEN_STAREQ},
    {.text = "/=", .kind = TOKEN_SLASHEQ},
    {.text = "%=", .kind = TOKEN_PERCENTEQ},
    {.text = "&=", .kind = TOKEN_ANDEQ},
    {.text = "|=", .kind = TOKEN_OREQ},
    {.text = "^=", .kind = TOKEN_XOREQ},
    {.text = "||", .kind = TOKEN_OROR},
    {.text = "&&", .kind = TOKEN_ANDAND},
    {.text = "|" , .kind = TOKEN_BITOR},
    {.text = "&" , .kind = TOKEN_BITAND},
    {.text = "^" , .kind = TOKEN_CARET},
    {.text = "~" , .kind = TOKEN_TILDE},
    {.text = "+" , .kind = TOKEN_PLUS},
    {.text = "-" , .kind = TOKEN_MINUS},
    {.text = "*" , .kind = TOKEN_STAR},
    {.text = "/" , .kind = TOKEN_SLASH},
    {.text = "%" , .kind = TOKEN_PERCENT},
};
#define literal_tokens_count (sizeof(literal_tokens) / sizeof(literal_tokens[0]))

static const char * const keywords[] = {
    "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",
};
#define keywords_count (sizeof(keywords) / sizeof(keywords[0]))

```

```c
struct Location {
    size_t line_num;
    size_t col;
};

struct Token {
    enum Token_Kind kind;
    const char *text;
    size_t text_len;
    struct Location location;
};

struct Lexer {
    const char * content;
    size_t content_len;
    size_t cursor;
    size_t line_num;
    size_t begining_of_line;
};

#define AL_UNUSED(x) (void)x

bool            al_is_identifier(char c);
bool            al_is_identifier_start(char c);
struct Lexer    al_lexer_alloc(const char *content, size_t len);
char            al_lexer_chop_char(struct Lexer *l);
void            al_lexer_chop_while(struct Lexer *l, bool (*pred)(char));
struct Token    al_lexer_next_token(struct Lexer *l);
bool            al_lexer_start_with(struct Lexer *l, const char *prefix);
void            al_lexer_trim_left(struct Lexer *l);
char            al_lexer_peek(const struct Lexer *l, size_t off);
void            al_token_print(struct Token tok);
const char *    al_token_kind_name(enum Token_Kind kind);

#endif /*ALMOG_LEXER_H_*/

#ifdef ALMOG_LEXER_IMPLEMENTATION
#undef ALMOG_LEXER_IMPLEMENTATION

#define ASM_NO_ERRORS

bool al_is_identifier(char c)
{
    return asm_isalnum(c) || c == '_';
}

bool al_is_identifier_start(char c)
{
    return asm_isalpha(c) || c == '_';
}

struct Lexer al_lexer_alloc(const char *content, size_t len)
{
    struct Lexer l = {0};
    l.content = content;
    l.content_len = len;
    return l;
}

char al_lexer_chop_char(struct Lexer *l)
{
    AL_ASSERT(l->cursor < l->content_len);
    char c = l->content[l->cursor++];
    if (c == '\n') {
        l->line_num++;
        l->begining_of_line = l->cursor;
    }
    return c;
}

void al_lexer_chop_while(struct Lexer *l, bool (*pred)(char))
{
    while (l->cursor < l->content_len && pred(l->content[l->cursor])) {
        al_lexer_chop_char(l);
    }
}

struct Token al_lexer_next_token(struct Lexer *l)
{
    al_lexer_trim_left(l);

    struct Token token = {
        .kind = TOKEN_INVALID,
        .text = &(l->content[l->cursor]),
        .text_len = 0,
        .location.line_num = l->line_num+1,
        .location.col = l->cursor - l->begining_of_line+1,
    };
    size_t start = l->cursor;

    if (l->cursor >= l->content_len) {
        token.kind = TOKEN_EOF;
    } else if (l->content[l->cursor] == '#' && token.location.col == 1) {
        token.kind = TOKEN_PP_DIRECTIVE;
        for (;l->cursor < l->content_len && l->content[l->cursor] != '\n';) {
            al_lexer_chop_char(l);
        }
        if (l->cursor < l->content_len) {
            al_lexer_chop_char(l);
        }
    } else if (al_is_identifier_start(l->content[l->cursor])) {
        token.kind = TOKEN_IDENTIFIER;
        for ( ; l->cursor < l->content_len && al_is_identifier(l->content[l->cursor]); ) {
            al_lexer_chop_char(l);
        }
        {
            size_t ident_len = l->cursor - start;
            for (size_t i = 0; i < keywords_count; i++) {
                size_t kw_len = asm_length(keywords[i]);
                if (ident_len == kw_len && asm_strncmp(token.text, keywords[i], kw_len)) {
                    token.kind = TOKEN_KEYWORD;
                    break;
                }
            }
        }
    } else if (l->content[l->cursor] == '"') {
        token.kind = TOKEN_STRING_LIT;
        al_lexer_chop_char(l);
        for ( ; (l->cursor < l->content_len) && (l->content[l->cursor] != '"') && (l->content[l->cursor] != '\n'); ) {
            al_lexer_chop_char(l);
        }
        if ((l->cursor < l->content_len) && (l->content[l->cursor] == '"')) {
            al_lexer_chop_char(l);
        }
    } else if (l->content[l->cursor] == '\'') {
        token.kind = TOKEN_CHAR_LIT;
        al_lexer_chop_char(l);
        for ( ; (l->cursor < l->content_len) && (l->content[l->cursor] != '\'') && (l->content[l->cursor] != '\n'); ) {
            al_lexer_chop_char(l);
        }
        if ((l->cursor < l->content_len) && (l->content[l->cursor] == '\'')) {
            al_lexer_chop_char(l);
        }
    } else if (al_lexer_start_with(l, "//")) {
        token.kind = TOKEN_COMMENT;
        for (;l->cursor < l->content_len && l->content[l->cursor] != '\n';) {
            al_lexer_chop_char(l);
        }
        if (l->cursor < l->content_len) {
            al_lexer_chop_char(l);
        }
    } else if (al_lexer_start_with(l, "/*")) {
        token.kind = TOKEN_COMMENT;
        al_lexer_chop_char(l);
        al_lexer_chop_char(l);
        for ( ; l->cursor < l->content_len; ) {
            if ((l->content[l->cursor-1] == '*') && (l->content[l->cursor] == '/')) {
                al_lexer_chop_char(l);
                break;
            }
            al_lexer_chop_char(l);
        }
    } else if (asm_isdigit(l->content[l->cursor]) || (l->content[l->cursor] == '.' && asm_isdigit(al_lexer_peek(l, 1)))) {
        token.kind = TOKEN_NUMBER;

        bool is_float = false;
        bool invalid = false;

        /* decimal float starting with "." */
        if (l->content[l->cursor] == '.') {
            is_float = true;
            al_lexer_chop_char(l);
            al_lexer_chop_while(l, asm_isdigit);

            /* optional exponent */
            if (al_lexer_peek(l, 0) == 'e' || al_lexer_peek(l, 0) == 'E') {
                is_float = true;
                al_lexer_chop_char(l);
                if (al_lexer_peek(l, 0) == '+' || al_lexer_peek(l, 0) == '-') {
                    al_lexer_chop_char(l);
                }
                if (!asm_isdigit(al_lexer_peek(l, 0))) {
                    invalid = true; /* ".5e" or ".5e+" */
                }
                al_lexer_chop_while(l, asm_isdigit);
            }
        } else {
            /* starts with digit */
            if (al_lexer_peek(l, 0) == '0' && (al_lexer_peek(l, 1) == 'x' || al_lexer_peek(l, 1) == 'X')) {
                /* hex int or hex float */
                al_lexer_chop_char(l);
                al_lexer_chop_char(l);

                size_t mantissa_digits = 0;
                while (asm_isXdigit(al_lexer_peek(l, 0)) || asm_isxdigit(al_lexer_peek(l, 0))) {
                    mantissa_digits++;
                    al_lexer_chop_char(l);
                }
                if (al_lexer_peek(l, 0) == '.') {
                    is_float = true;
                    al_lexer_chop_char(l);
                    while (asm_isXdigit(al_lexer_peek(l, 0)) || asm_isxdigit(al_lexer_peek(l, 0))) {
                        mantissa_digits++;
                        al_lexer_chop_char(l);
                    }
                }
                if (mantissa_digits == 0) {
                    invalid = true; /* "0x" or "0x." */
                }

                /* Hex float requires p/P exponent if it's a float form. */
                if (al_lexer_peek(l, 0) == 'p' || al_lexer_peek(l, 0) == 'P') {
                    is_float = true;
                    al_lexer_chop_char(l);
                    if (al_lexer_peek(l, 0) == '+' || al_lexer_peek(l, 0) == '-') {
                        al_lexer_chop_char(l);
                    }
                    if (!asm_isdigit(al_lexer_peek(l, 0))) {
                        invalid = true; /* "0x1.fp" / "0x1p+" */
                    }
                    al_lexer_chop_while(l, asm_isdigit);
                } else if (is_float) {
                    /* Had a '.' in hex mantissa but no p-exponent => invalid hex float */
                    invalid = true;
                }
            } else if (al_lexer_peek(l, 0) == '0' && (al_lexer_peek(l, 1) == 'b' || al_lexer_peek(l, 1) == 'B')) {
                /* binary int */
                al_lexer_chop_char(l);
                al_lexer_chop_char(l);
                if (!asm_isbdigit(al_lexer_peek(l, 0))) {
                    invalid = true; /* "0b" */
                }
                al_lexer_chop_while(l, asm_isbdigit);
            } else if (al_lexer_peek(l, 0) == '0' && (al_lexer_peek(l, 1) == 'o' || al_lexer_peek(l, 1) == 'O')) {
                /* explicit octal int */
                al_lexer_chop_char(l);
                al_lexer_chop_char(l);
                if (!asm_isodigit(al_lexer_peek(l, 0))) {
                    invalid = true; /* "0o" */
                }
                while (asm_isodigit(al_lexer_peek(l, 0))) {
                    al_lexer_chop_char(l);
                }
            } else {
                /* decimal int or decimal float */
                al_lexer_chop_while(l, asm_isdigit);

                if (al_lexer_peek(l, 0) == '.') {
                    is_float = true;
                    al_lexer_chop_char(l);
                    al_lexer_chop_while(l, asm_isdigit);
                }

                if (al_lexer_peek(l, 0) == 'e' || al_lexer_peek(l, 0) == 'E') {
                    is_float = true;
                    al_lexer_chop_char(l);
                    if (al_lexer_peek(l, 0) == '+' || al_lexer_peek(l, 0) == '-') {
                        al_lexer_chop_char(l);
                    }
                    if (!asm_isdigit(al_lexer_peek(l, 0))) {
                        invalid = true; /* "1e" / "1e+" */
                    }
                    al_lexer_chop_while(l, asm_isdigit);
                }
            }
        }

        /* Suffix handling */
        if (is_float) {
            /* float suffixes: f/F/l/L (accept at most one, but we'll be permissive) */
            while (al_lexer_peek(l, 0) == 'f' || al_lexer_peek(l, 0) == 'F' ||
                   al_lexer_peek(l, 0) == 'l' || al_lexer_peek(l, 0) == 'L') {
                al_lexer_chop_char(l);
            }
        } else {
            /* integer suffixes: u/U/l/L/z/Z (permissive) */
            while (al_lexer_peek(l, 0) == 'u' || al_lexer_peek(l, 0) == 'U' ||
                   al_lexer_peek(l, 0) == 'l' || al_lexer_peek(l, 0) == 'L' ||
                   al_lexer_peek(l, 0) == 'z' || al_lexer_peek(l, 0) == 'Z') {
                al_lexer_chop_char(l);
            }
        }

        if (invalid) token.kind = TOKEN_INVALID;
    } else {
        size_t longest_matching_token = 0;
        enum Token_Kind best_kind = TOKEN_INVALID;
        for (size_t i = 0; i < literal_tokens_count; i++) {
            if (al_lexer_start_with(l, literal_tokens[i].text)) {
                /* NOTE: assumes that literal_tokens[i].text does not have any '\n' */
                size_t text_len = asm_length(literal_tokens[i].text);
                if (text_len > longest_matching_token) {
                    longest_matching_token = text_len;
                    best_kind = literal_tokens[i].kind;                }
            }
        }
        if (longest_matching_token > 0) {
            token.kind = best_kind;
            for (size_t i = 0; i < longest_matching_token; i++) {
                al_lexer_chop_char(l);
            }
        } else {
            token.kind = TOKEN_INVALID;
            al_lexer_chop_char(l);
        }
    }

    token.text_len = l->cursor - start;

    return token;
}

bool al_lexer_start_with(struct Lexer *l, const char *prefix)
{
    size_t prefix_len = asm_length(prefix);
    if (prefix_len == 0) {
        return true;
    }
    if (l->cursor + prefix_len > l->content_len) {
        return false;
    }
    for (size_t i = 0; i < prefix_len; i++) {
        if (prefix[i] != l->content[l->cursor + i]) {
            return false;
        }
    }
    return true;
}

void al_lexer_trim_left(struct Lexer *l)
{
    for (;l->cursor < l->content_len;) {
        if (!asm_isspace(l->content[l->cursor])) {
            break;
        }
        al_lexer_chop_char(l);
    }
}

char al_lexer_peek(const struct Lexer *l, size_t off)
{
    size_t i = l->cursor + off;
    if (i >= l->content_len) return '\0';
    return l->content[i];
}

void al_token_print(struct Token tok)
{
    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);
}

```

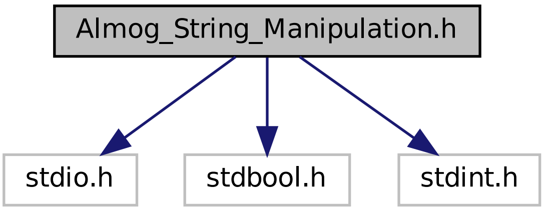
```
const char *al_token_kind_name(enum Token_Kind kind)
{
    switch (kind) {
        case TOKEN_EOF:
            return ("TOKEN_EOF");
        case TOKEN_INVALID:
            return ("TOKEN_INVALID");
        case TOKEN_PP_DIRECTIVE:
            return ("TOKEN_PP_DIRECTIVE");
        case TOKEN_IDENTIFIER:
            return ("TOKEN_IDENTIFIER");
        case TOKEN_LPAREN:
            return ("TOKEN_LPAREN");
        case TOKEN_RPAREN:
            return ("TOKEN_RPAREN");
        case TOKEN_LBRACKET:
            return ("TOKEN_LBRACKET");
        case TOKEN_RBRACKET:
            return ("TOKEN_RBRACKET");
        case TOKEN_LBRACE:
            return ("TOKEN_LBRACE");
        case TOKEN_RBRACE:
            return ("TOKEN_RBRACE");
        case TOKEN_DOT:
            return ("TOKEN_DOT");
        case TOKEN_COMMA:
            return ("TOKEN_COMMA");
        case TOKEN_SEMICOLON:
            return ("TOKEN_SEMICOLON");
        case TOKEN_BSLASH:
            return ("TOKEN_BSLASH");
        case TOKEN_QUESTION:
            return ("TOKEN_QUESTION");
        case TOKEN_COLON:
            return ("TOKEN_COLON");
        case TOKEN_LT:
            return ("TOKEN_LT");
        case TOKEN_GT:
            return ("TOKEN_GT");
        case TOKEN_GE:
            return ("TOKEN_GE");
        case TOKEN_LE:
            return ("TOKEN_LE");
        case TOKEN_KEYWORD:
            return ("TOKEN_KEYWORD");
        case TOKEN_NUMBER:
            return ("TOKEN_NUMBER");
        case TOKEN_COMMENT:
            return ("TOKEN_COMMENT");
        case TOKEN_STRING_LIT:
            return ("TOKEN_STRING_LIT");
        case TOKEN_CHAR_LIT:
            return ("TOKEN_CHAR_LIT");
        case TOKEN_EQ:
            return ("TOKEN_EQ");
        case TOKEN_EQEQ:
            return ("TOKEN_EQEQ");
        case TOKEN_NE:
            return ("TOKEN_NE");
        case TOKEN_BANG:
            return ("TOKEN_BANG");
        case TOKEN_BITAND:
            return ("TOKEN_BITAND");
        case TOKEN_ANDAND:
            return ("TOKEN_ANDAND");
        case TOKEN_BITOR:
            return ("TOKEN_BITOR");
        case TOKEN_OROR:
            return ("TOKEN_OROR");
        case TOKEN_CARET:
            return ("TOKEN_CARET");
        case TOKEN_TILDE:
            return ("TOKEN_TILDE");
        case TOKEN_PLUSPLUS:
            return ("TOKEN_PLUSPLUS");
        case TOKEN_MINUSMINUS:
            return ("TOKEN_MINUSMINUS");
        case TOKEN_LSHIFT:
            return ("TOKEN_LSHIFT");
        case TOKEN_RSHIFT:
            return ("TOKEN_RSHIFT");
        case TOKEN_PLUS:
            return ("TOKEN_PLUS");
        case TOKEN_MINUS:
            return ("TOKEN_MINUS");
        case TOKEN_STAR:
            return ("TOKEN_STAR");
        case TOKEN_SLASH:
            return ("TOKEN_SLASH");
        case TOKEN_HASH:
            return ("TOKEN_HASH");
        case TOKEN_PERCENT:
            return ("TOKEN_PERCENT");
        case TOKEN_PLUSEQ:
            return ("TOKEN_PLUSEQ");
        case TOKEN_MINUSEQ:
            return ("TOKEN_MINUSEQ");
        case TOKEN_STAREQ:
            return ("TOKEN_STAREQ");
        case TOKEN_SLASHEQ:
            return ("TOKEN_SLASHEQ");
        case TOKEN_PERCENTEQ:
            return ("TOKEN_PERCENTEQ");
        case TOKEN_ANDEQ:
            return ("TOKEN_ANDEQ");
        case TOKEN_OREQ:
            return ("TOKEN_OREQ");
        case TOKEN_XOREQ:
            return ("TOKEN_XOREQ");
        case TOKEN_LSHIFTEQ:
            return ("TOKEN_LSHIFTEQ");
        case TOKEN_RSHIFTEQ:
            return ("TOKEN_RSHIFTEQ");
        case TOKEN_ARROW:
            return ("TOKEN_ARROW");
        case TOKEN_ELLIPSIS:
            return ("TOKEN_ELLIPSIS");
        default:
            AL_ASSERT(0 && "Unknown kind");
    }
    return NULL;
}

#endif /*ALMOG_LEXER_IMPLEMENTATION*/

```

4.5 Almog_String_Manipulation.h File Reference

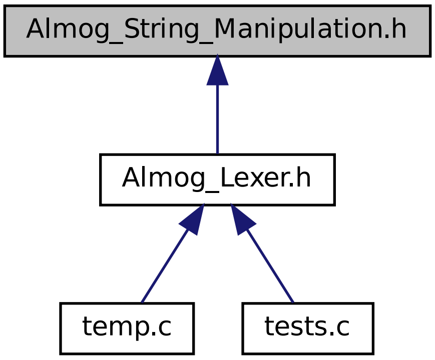
Lightweight string and line manipulation helpers.

```
#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

```

#ifndef ALMOG_STRING_MANIPULATION_H_
#define ALMOG_STRING_MANIPULATION_H_

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

#ifndef ASM_MAX_LEN
#define ASM_MAX_LEN (int)1e3
#endif

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

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

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

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

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

```c

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

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

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

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

bool    asm_check_char_belong_to_base(const char c, const size_t base);
void    asm_copy_array_by_indexes(char * const target, const int start, const int end, const char * const src);
int     asm_get_char_value_in_base(const char c, const size_t base);
int     asm_get_line(FILE *fp, char * const dst);
int     asm_get_next_token_from_str(char * const dst, const char * const src, const char delimiter);
int     asm_get_token_and_cut(char * const dst, char *src, const char delimiter, const bool leave_delimiter);
bool    asm_isalnum(const char c);
bool    asm_isalpha(const char c);
bool    asm_isbdigit(const char c);
bool    asm_iscntrl(const char c);
bool    asm_isdigit(const char c);
bool    asm_isgraph(const char c);
bool    asm_islower(const char c);
bool    asm_isodigit(const char c);
bool    asm_isprint(const char c);
bool    asm_ispunct(const char c);
bool    asm_isspace(const char c);
bool    asm_isupper(const char c);
bool    asm_isxdigit(const char c);
bool    asm_isXdigit(const char c);
size_t  asm_length(const char * const str);
void *  asm_memset(void * const des, const unsigned char value, const size_t n);
void    asm_pad_left(char * const s, const size_t padding, const char pad);
void    asm_print_many_times(const char * const str, const size_t n);
void    asm_remove_char_from_string(char * const s, const size_t index);
void    asm_shift_left(char * const s, const size_t shift);
int     asm_str_in_str(const char * const src, const char * const word_to_search);
double  asm_str2double(const char * const s, const char ** const end, const size_t base);
float   asm_str2float(const char * const s, const char ** const end, const size_t base);
int     asm_str2int(const char * const s, const char ** const end, const size_t base);
size_t  asm_str2size_t(const char * const s, const char ** const end, const size_t base);
void    asm_strip_whitespace(char * const s);
bool    asm_str_is_whitespace(const char * const s);
int     asm_strncat(char * const s1, const char * const s2, const size_t N);
int     asm_strncmp(const char * const s1, const char * const s2, const size_t N);
int     asm_strncpy(char * const s1, const char * const s2, const size_t N);
void    asm_tolower(char * const s);
void    asm_toupper(char * const s);
void    asm_trim_left_whitespace(char *s);

#endif /*ALMOG_STRING_MANIPULATION_H_*/

#ifdef ALMOG_STRING_MANIPULATION_IMPLEMENTATION
#undef ALMOG_STRING_MANIPULATION_IMPLEMENTATION

bool asm_check_char_belong_to_base(const char c, const size_t base)
{
    if (base > 36 || base < 2) {
        #ifndef NO_ERRORS
        asm_dprintERROR("Supported bases are [2...36]. Inputted: %zu", base);
        #endif
        return false;
    }
    if (base <= 10) {
        return c >= '0' && c <= '9'+(char)base-10;
    }
    if (base > 10) {
        return asm_isdigit(c) || (c >= 'A' && c <= ('A'+(char)base-11)) || (c >= 'a' && c <= ('a'+(char)base-11));
    }

    return false;
}

void asm_copy_array_by_indexes(char * const target, const int start, const int end, const char * const src)
{
    if (start > end) return;
    int j = 0;
    for (int i = start; i <= end; i++) {
        target[j] = src[i];
        j++;
    }
    if (target[j-1] != '\0') {
            target[j] = ’\0’;
        }
}

int asm_get_char_value_in_base(const char c, const size_t base)
{
    if (!asm_check_char_belong_to_base(c, base)) return -1;
    if (asm_isdigit(c)) {
        return c - ’0’;
    } else if (asm_isupper(c)) {
        return c - ’A’ + 10;
    } else {
        return c - ’a’ + 10;
    }
}

int asm_get_line(FILE *fp, char * const dst)
{
    int i = 0;
    int c;
    while ((c = fgetc(fp)) != ’\n’ && c != EOF) {
        dst[i++] = c;
        if (i >= ASM_MAX_LEN) {
            #ifndef NO_ERRORS
            asm_dprintERROR("%s", "index exceeds ASM_MAX_LEN. Line in file is too long.");
            #endif
            dst[i-1] = ’\0’;
            return -1;
        }
    }
    dst[i] = ’\0’;
    if (c == EOF && i == 0) {
        return -1;
    }
    return i;
}

int asm_get_next_token_from_str(char * const dst, const char * const src, const char delimiter)
{
    int i = 0, j = 0;
    char c;
    while ((c = src[i]) != delimiter && c != ’\0’) {
        dst[j++] = src[i++];
    }

    dst[j] = ’\0’;

    return j;
}

int asm_get_token_and_cut(char * const dst, char *src, const char delimiter, const bool
 leave_delimiter)
{
    int new_src_start_index = asm_get_next_token_from_str(dst, src, delimiter);
    bool delimiter_at_start = src[new_src_start_index] == delimiter;

    if (leave_delimiter) {
        asm_shift_left(src, new_src_start_index);
    } else if (delimiter_at_start) {
        asm_shift_left(src, new_src_start_index + 1);
    } else {
        src[0] = ’\0’;
    }
    return new_src_start_index ? 1 : 0;
}

bool asm_isalnum(char c)
{
    return asm_isalpha(c) || asm_isdigit(c);
}

bool asm_isalpha(char c)
{
    return asm_isupper(c) || asm_islower(c);
}

bool asm_isbdigit(const char c)
{
    if (c == ’0’ || c == ’1’) {
        return true;
    } else {
        return false;
    }
}

bool asm_iscntrl(char c)
{
    if ((c >= 0 && c <= 31) || c == 127) {
        return true;
    } else {
        return false;
    }
}

bool asm_isdigit(char c)
{
    if (c >= '0' && c <= '9') {
        return true;
    } else {
        return false;
    }
}

bool asm_isgraph(char c)
{
    if (c >= 33 && c <= 126) {
        return true;
    } else {
        return false;
    }
}

bool asm_islower(char c)
{
    if (c >= 'a' && c <= 'z') {
        return true;
    } else {
        return false;
    }
}

bool asm_isodigit(const char c)
{
    if ((c >= '0' && c <= '7')) {
        return true;
    } else {
        return false;
    }
}

bool asm_isprint(char c)
{
    return asm_isgraph(c) || c == ' ';
}

bool asm_ispunct(char c)
{
    if ((c >= 33 && c <= 47) || (c >= 58 && c <= 64) || (c >= 91 && c <= 96) || (c >= 123 && c <= 126)) {
        return true;
    } else {
        return false;
    }
}

bool asm_isspace(char c)
{
    if (c == ' ' || c == '\n' || c == '\t' ||
        c == '\v'|| c == '\f' || c == '\r') {
        return true;
    } else {
        return false;
    }
}

bool asm_isupper(char c)
{
    if (c >= 'A' && c <= 'Z') {
        return true;
    } else {
        return false;
    }
}

bool asm_isxdigit(char c)
{
    if ((c >= 'a' && c <= 'f') || asm_isdigit(c)) {
        return true;
    } else {
        return false;
    }
}

bool asm_isXdigit(char c)
```

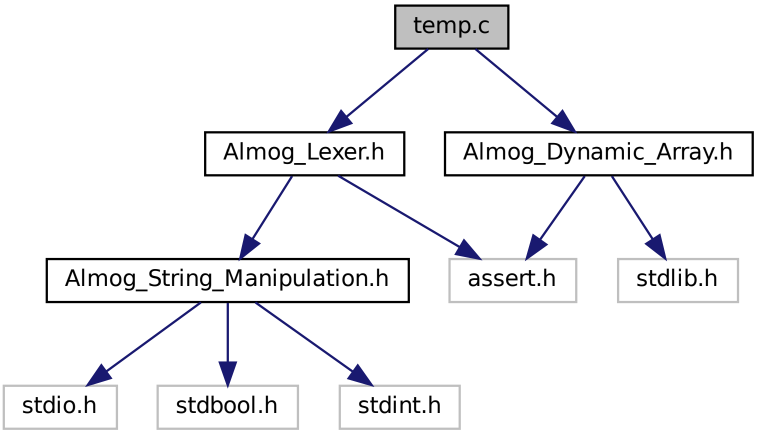
```
{
    if ((c >= 'A' && c <= 'F') || asm_isdigit(c)) {
        return true;
    } else {
        return false;
    }
}

size_t asm_length(const char * const str)
{
    char c;
    size_t i = 0;

    while ((c = str[i++]) != '\0') {
        if (i > ASM_MAX_LEN) {
            #ifndef NO_ERRORS
            asm_dprintERROR("%s", "index exceeds ASM_MAX_LEN. Probably no NULL termination.");
            #endif
            return SIZE_MAX;
        }
    }
    return --i;
}

void * asm_memset(void * const des, const unsigned char value, const size_t n)
{
    unsigned char *ptr = (unsigned char *)des;
    for (size_t i = n; i-- > 0;) {
        *ptr++ = value;
    }
    return des;
}

void asm_pad_left(char * const s, const size_t padding, const char pad)
{
    int len = (int)asm_length(s);
    for (int i = len; i >= 0; i--) {
        s[i+(int)padding] = s[i];
    }
    for (int i = 0; i < (int)padding; i++) {
        s[i] = pad;
    }
}

void asm_print_many_times(const char * const str, const size_t n)
{
    for (size_t i = 0; i < n; i++) {
        printf("%s", str);
    }
    printf("\n");
}

void asm_remove_char_from_string(char * const s, const size_t index)
{
    size_t len = asm_length(s);
    if (len == 0) return;
    if (index >= len) {
        #ifndef NO_ERRORS
        asm_dprintERROR("%s", "index exceeds array length.");
        #endif
        return;
    }

    for (size_t i = index; i < len; i++) {
        s[i] = s[i+1];
    }
}

void asm_shift_left(char * const s, const size_t shift)
{
    size_t len = asm_length(s);

    if (shift == 0) return;
    if (len <= shift) {
        s[0] = '\0';
        return;
    }

    size_t i;
    for (i = shift; i < len; i++) {
        s[i-shift] = s[i];
    }
    s[i-shift] = '\0';
}

int asm_str_in_str(const char * const src, const char * const word_to_search)
{
    int i = 0, num_of_accur = 0;
    while (src[i] != ’\0’) {
        if (asm_strncmp(src+i, word_to_search, asm_length(word_to_search))) {
            num_of_accur++;
        }
        i++;
    }
    return num_of_accur;
}

double asm_str2double(const char * const s, const char ** const end, const size_t base)
{
    if (base < 2 || base > 36) {
        #ifndef NO_ERRORS
        asm_dprintERROR("Supported bases are [2...36]. Input: %zu", base);
        #endif
        if (end) *end = s;
        return 0.0;
    }
    int num_of_whitespace = 0;
    while (asm_isspace(s[num_of_whitespace])) {
        num_of_whitespace++;
    }

    int i = 0;
    if (s[0+num_of_whitespace] == ’-’ || s[0+num_of_whitespace] == ’+’) {
        i++;
    }
    int sign = s[0+num_of_whitespace] == ’-’ ? -1 : 1;

    size_t left = 0;
    double right = 0.0;
    int expo = 0;
    for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
        left = base * left + asm_get_char_value_in_base(s[i+num_of_whitespace], base);
    }

    if (s[i+num_of_whitespace] == ’.’) {
        i++; /* skip the point */

        size_t divider = base;
        for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
            right = right + asm_get_char_value_in_base(s[i+num_of_whitespace], base) / (double)divider;
            divider *= base;
        }
    }

    if ((s[i+num_of_whitespace] == ’e’) || (s[i+num_of_whitespace] == ’E’)) {
        expo = asm_str2int(&(s[i+num_of_whitespace+1]), end, 10);
    } else {
        if (end) *end = s + i + num_of_whitespace;
    }

    double res = sign * (left + right);

    if (expo > 0) {
        for (int index = 0; index < expo; index++) {
            res *= (double)base;
        }
    } else {
        for (int index = 0; index > expo; index--) {
            res /= (double)base;
        }
    }

    return res;
}

float asm_str2float(const char * const s, const char ** const end, const size_t base)
{
    if (base < 2 || base > 36) {
        #ifndef NO_ERRORS
        asm_dprintERROR("Supported bases are [2...36]. Input: %zu", base);
        #endif
        if (end) *end = s;
        return 0.0f;
    }
    int num_of_whitespace = 0;
    while (asm_isspace(s[num_of_whitespace])) {
        num_of_whitespace++;
    }

    int i = 0;
    if (s[0+num_of_whitespace] == ’-’ || s[0+num_of_whitespace] == ’+’) {
        i++;
    }
    int sign = s[0+num_of_whitespace] == ’-’ ? -1 : 1;

    int left = 0;
    float right = 0.0f;
    int expo = 0;
    for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
        left = base * left + asm_get_char_value_in_base(s[i+num_of_whitespace], base);
    }

    if (s[i+num_of_whitespace] == ’.’) {
        i++; /* skip the point */

        size_t divider = base;
        for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
            right = right + asm_get_char_value_in_base(s[i+num_of_whitespace], base) / (float)divider;
            divider *= base;
        }
    }

    if ((s[i+num_of_whitespace] == ’e’) || (s[i+num_of_whitespace] == ’E’)) {
        expo = asm_str2int(&(s[i+num_of_whitespace+1]), end, 10);
    } else {
        if (end) *end = s + i + num_of_whitespace;
    }

    float res = sign * (left + right);

    if (expo > 0) {
        for (int index = 0; index < expo; index++) {
            res *= (float)base;
        }
    } else {
        for (int index = 0; index > expo; index--) {
            res /= (float)base;
        }
    }

    return res;
}

int asm_str2int(const char * const s, const char ** const end, const size_t base)
{
    if (base < 2 || base > 36) {
        #ifndef NO_ERRORS
        asm_dprintERROR("Supported bases are [2...36]. Input: %zu", base);
        #endif
        if (end) *end = s;
        return 0;
    }
    int num_of_whitespace = 0;
    while (asm_isspace(s[num_of_whitespace])) {
        num_of_whitespace++;
    }

    int n = 0, i = 0;
    if (s[0+num_of_whitespace] == ’-’ || s[0+num_of_whitespace] == ’+’) {
        i++;
    }
    int sign = s[0+num_of_whitespace] == ’-’ ? -1 : 1;

    for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
        n = base * n + asm_get_char_value_in_base(s[i+num_of_whitespace], base);
    }

    if (end) *end = s + i+num_of_whitespace;

    return n * sign;
}

size_t asm_str2size_t(const char * const s, const char ** const end, const size_t base)
{
    if (end) *end = s;

    int num_of_whitespace = 0;
    while (asm_isspace(s[num_of_whitespace])) {
        num_of_whitespace++;
    }

    if (s[0+num_of_whitespace] == ’-’) {
        #ifndef NO_ERRORS
        asm_dprintERROR("%s", "Unable to convert a negative number to size_t.");
        #endif
        return 0;
    }

    if (base < 2 || base > 36) {
        #ifndef NO_ERRORS
        asm_dprintERROR("Supported bases are [2...36]. Input: %zu", base);
        #endif
        if (end) *end = s+num_of_whitespace;
        return 0;
    }

    size_t n = 0, i = 0;
    if (s[0+num_of_whitespace] == ’+’) {
        i++;
    }

    for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
        n = base * n + asm_get_char_value_in_base(s[i+num_of_whitespace], base);
    }

    if (end) *end = s + i+num_of_whitespace;

    return n;
}

void asm_strip_whitespace(char * const s)
{
    size_t len = asm_length(s);
    size_t i;
    for (i = 0; i < len; i++) {
        if (asm_isspace(s[i])) {
            asm_remove_char_from_string(s, i);
            len--;
            i--;
        }
    }
    s[i] = ’\0’;
}

bool asm_str_is_whitespace(const char * const s)
{
    size_t len = asm_length(s);
    for (size_t i = 0; i < len; i++) {
        if (!asm_isspace(s[i])) {
            return false;
        }
    }

    return true;
}

int asm_strncat(char * const s1, const char * const s2, const size_t N)
{
    size_t len_s1 = asm_length(s1);

    int limit = N;
    if (limit == 0) {
        limit = ASM_MAX_LEN;
    }

    int i = 0;
    while (i < limit && s2[i] != ’\0’) {
        if (len_s1 + (size_t)i >= ASM_MAX_LEN-1) {
            #ifndef NO_ERRORS
            asm_dprintERROR("s2 or the first N=%zu digit of s2 does not fit into s1.", N);
            #endif
            return i;
        }

        s1[len_s1+(size_t)i] = s2[i];
        i++;
    }
    s1[len_s1+(size_t)i] = ’\0’;

    return i;
}

int asm_strncmp(const char *s1, const char *s2, const size_t N)
{
    size_t i = 0;
    while (i < N) {
        if (s1[i] == ’\0’ && s2[i] == ’\0’) {
            break;
        }
        if (s1[i] != s2[i] || (s1[i] == ’\0’) || (s2[i] == ’\0’)) {
            return 0;
        }
        i++;
    }
    return 1;
}

```

```
int asm_strncpy(char * const s1, const char * const s2, const size_t N)
{
    size_t len1 = asm_length(s1);
    size_t len2 = asm_length(s2);

    size_t n = N < len2 ? N : len2;

    if (n > len1) {
        #ifndef NO_ERRORS
        asm_dprintERROR("%s", "min(N, len(s2)) is bigger then len(s1)");
        #endif
        return 0;
    }

    size_t i;
    for (i = 0; i < n; i++) {
        s1[i] = s2[i];
    }
    s1[i] = ’\0’;

    return i;
}

void asm_tolower(char * const s)
{
    size_t len = asm_length(s);
    for (size_t i = 0; i < len; i++) {
        if (asm_isupper(s[i])) {
            s[i] += ’a’ - ’A’;
        }
    }
}

void asm_toupper(char * const s)
{
    size_t len = asm_length(s);
    for (size_t i = 0; i < len; i++) {
        if (asm_islower(s[i])) {
            s[i] += ’A’ - ’a’;
        }
    }
}

void asm_trim_left_whitespace(char * const s)
{
    size_t len = asm_length(s);

    if (len == 0) return;
    size_t i;
    for (i = 0; i < len; i++) {
        if (!asm_isspace(s[i])) {
            break;
        }
    }
    asm_shift_left(s, i);
}

#ifdef NO_ERRORS
#undef NO_ERRORS
#endif

#endif /*ALMOG_STRING_MANIPULATION_IMPLEMENTATION*/

```

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.

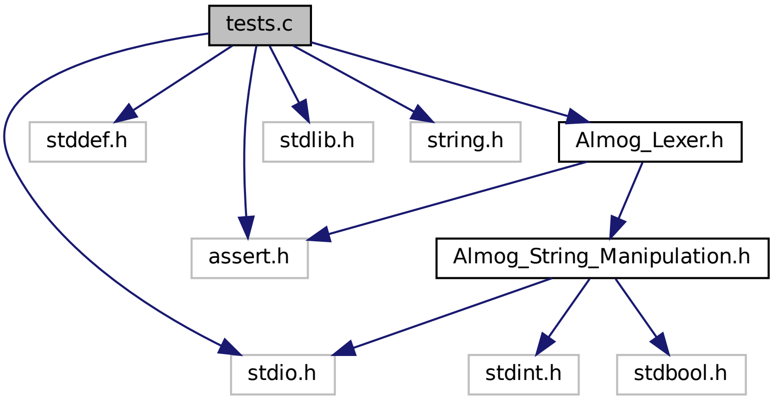
4.8 temp.c

```
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

```

#include <assert.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Compile implementations in THIS translation unit. */
#define ALMOG_STRING_MANIPULATION_IMPLEMENTATION
#define ALMOG_LEXER_IMPLEMENTATION
#include "Almog_Lexer.h"

static const char *kind_name(enum Token_Kind k)
{
    switch (k) {
        case TOKEN_EOF: return "TOKEN_EOF";
        case TOKEN_INVALID: return "TOKEN_INVALID";
        case TOKEN_PP_DIRECTIVE: return "TOKEN_PP_DIRECTIVE";
        case TOKEN_COMMENT: return "TOKEN_COMMENT";
        case TOKEN_STRING_LIT: return "TOKEN_STRING_LIT";
        case TOKEN_CHAR_LIT: return "TOKEN_CHAR_LIT";
        case TOKEN_NUMBER: return "TOKEN_NUMBER";
        case TOKEN_KEYWORD: return "TOKEN_KEYWORD";
        case TOKEN_IDENTIFIER: return "TOKEN_IDENTIFIER";

        case TOKEN_LPAREN: return "TOKEN_LPAREN";
        case TOKEN_RPAREN: return "TOKEN_RPAREN";
        case TOKEN_LBRACKET: return "TOKEN_LBRACKET";
        case TOKEN_RBRACKET: return "TOKEN_RBRACKET";
        case TOKEN_LBRACE: return "TOKEN_LBRACE";
        case TOKEN_RBRACE: return "TOKEN_RBRACE";

        case TOKEN_DOT: return "TOKEN_DOT";
        case TOKEN_COMMA: return "TOKEN_COMMA";
        case TOKEN_SEMICOLON: return "TOKEN_SEMICOLON";
        case TOKEN_BSLASH: return "TOKEN_BSLASH";
        case TOKEN_HASH: return "TOKEN_HASH";

        case TOKEN_QUESTION: return "TOKEN_QUESTION";
        case TOKEN_COLON: return "TOKEN_COLON";

        case TOKEN_EQ: return "TOKEN_EQ";
        case TOKEN_EQEQ: return "TOKEN_EQEQ";
        case TOKEN_NE: return "TOKEN_NE";
        case TOKEN_BANG: return "TOKEN_BANG";

        case TOKEN_LT: return "TOKEN_LT";
        case TOKEN_GT: return "TOKEN_GT";
        case TOKEN_LE: return "TOKEN_LE";
        case TOKEN_GE: return "TOKEN_GE";

        case TOKEN_BITAND: return "TOKEN_BITAND";
        case TOKEN_ANDAND: return "TOKEN_ANDAND";
        case TOKEN_BITOR: return "TOKEN_BITOR";
        case TOKEN_OROR: return "TOKEN_OROR";
        case TOKEN_CARET: return "TOKEN_CARET";
        case TOKEN_TILDE: return "TOKEN_TILDE";

        case TOKEN_LSHIFT: return "TOKEN_LSHIFT";
        case TOKEN_RSHIFT: return "TOKEN_RSHIFT";

        case TOKEN_PLUSPLUS: return "TOKEN_PLUSPLUS";
        case TOKEN_MINUSMINUS: return "TOKEN_MINUSMINUS";

        case TOKEN_PLUS: return "TOKEN_PLUS";
        case TOKEN_MINUS: return "TOKEN_MINUS";
        case TOKEN_STAR: return "TOKEN_STAR";
        case TOKEN_SLASH: return "TOKEN_SLASH";
        case TOKEN_PERCENT: return "TOKEN_PERCENT";

        case TOKEN_PLUSEQ: return "TOKEN_PLUSEQ";
        case TOKEN_MINUSEQ: return "TOKEN_MINUSEQ";
        case TOKEN_STAREQ: return "TOKEN_STAREQ";
        case TOKEN_SLASHEQ: return "TOKEN_SLASHEQ";
        case TOKEN_PERCENTEQ: return "TOKEN_PERCENTEQ";
        case TOKEN_ANDEQ: return "TOKEN_ANDEQ";
        case TOKEN_OREQ: return "TOKEN_OREQ";
        case TOKEN_XOREQ: return "TOKEN_XOREQ";
        case TOKEN_LSHIFTEQ: return "TOKEN_LSHIFTEQ";
        case TOKEN_RSHIFTEQ: return "TOKEN_RSHIFTEQ";

        case TOKEN_ARROW: return "TOKEN_ARROW";
        case TOKEN_ELLIPSIS: return "TOKEN_ELLIPSIS";
    }
    return "TOKEN_<unknown>";
}

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
)
{
    fprintf(stderr, "\n[FAIL] %s\n", test_name);
    fprintf(stderr, "  expected: kind=%s", kind_name(exp_kind));
    if (exp_text) {
        fprintf(stderr, ", text=\"%s\" (len=%zu)", exp_text, strlen(exp_text));
    }
    if (exp_line) fprintf(stderr, ", line=%zu", exp_line);
    if (exp_col) fprintf(stderr, ", col=%zu", exp_col);
    fprintf(stderr, "\n");

    fprintf(stderr, "  got:      kind=%s, text_len=%zu, line=%zu, col=%zu, text=\"%.*s\"\n",
            kind_name(got.kind),
            got.text_len,
            got.location.line_num,
            got.location.col,
            (int)got.text_len,
            got.text ? got.text : "");
    exit(1);
}

/* If exp_text == NULL => don’t check text. If exp_line/col == 0 => don’t check. */
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
)
{
    struct Token t = al_lexer_next_token(l);

    if (t.kind != exp_kind) {
        fail_token(test_name, t, exp_kind, exp_text, exp_line, exp_col);
    }
    if (exp_text) {
        size_t n = strlen(exp_text);
        if (t.text_len != n || memcmp(t.text, exp_text, n) != 0) {
            fail_token(test_name, t, exp_kind, exp_text, exp_line, exp_col);
        }
    }

    if (exp_line && t.location.line_num != exp_line) {
        fail_token(test_name, t, exp_kind, exp_text, exp_line, exp_col);
    }

    if (exp_col && t.location.col != exp_col) {
        fail_token(test_name, t, exp_kind, exp_text, exp_line, exp_col);
    }
}

static void test_basic_program(void)
{
    const char *name = "basic_program";
    const char *src = "int main() { return 0; }";
    struct Lexer l = al_lexer_alloc(src, strlen(src));

    expect_tok(name, &l, TOKEN_KEYWORD, "int", 0, 0);
    expect_tok(name, &l, TOKEN_IDENTIFIER, "main", 0, 0);
    expect_tok(name, &l, TOKEN_LPAREN, "(", 0, 0);
    expect_tok(name, &l, TOKEN_RPAREN, ")", 0, 0);
    expect_tok(name, &l, TOKEN_LBRACE, "{", 0, 0);
    expect_tok(name, &l, TOKEN_KEYWORD, "return", 0, 0);
    expect_tok(name, &l, TOKEN_NUMBER, "0", 0, 0);
    expect_tok(name, &l, TOKEN_SEMICOLON, ";", 0, 0);
    expect_tok(name, &l, TOKEN_RBRACE, "}", 0, 0);
    expect_tok(name, &l, TOKEN_EOF, NULL, 0, 0);
}

static void test_pp_directive_and_locations(void)
{
    const char *name = "pp_directive_and_locations";
    const char *src = "#include <stdio.h>\nint x;\n";
    struct Lexer l = al_lexer_alloc(src, strlen(src));

    /* PP directive is only recognized at col==1 and includes the newline. */
    expect_tok(name, &l, TOKEN_PP_DIRECTIVE, "#include <stdio.h>\n", 1, 1);

    expect_tok(name, &l, TOKEN_KEYWORD, "int", 2, 1);
    expect_tok(name, &l, TOKEN_IDENTIFIER, "x", 2, 5);
    expect_tok(name, &l, TOKEN_SEMICOLON, ";", 2, 6);

    expect_tok(name, &l, TOKEN_EOF, NULL, 0, 0);
}

static void test_whitespace_location_math(void)
{
    const char *name = "whitespace_location";
    const char *src = "a\n  b";
    struct Lexer l = al_lexer_alloc(src, strlen(src));

    expect_tok(name, &l, TOKEN_IDENTIFIER, "a", 1, 1);
    expect_tok(name, &l, TOKEN_IDENTIFIER, "b", 2, 3);
    expect_tok(name, &l, TOKEN_EOF, NULL, 0, 0);
}

static void test_comments(void)
{
    {
        const char *name = "line_comment_includes_newline";
        const char *src = "// hello\nx";
        struct Lexer l = al_lexer_alloc(src, strlen(src));

        expect_tok(name, &l, TOKEN_COMMENT, "// hello\n", 1, 1);
        expect_tok(name, &l, TOKEN_IDENTIFIER, "x", 2, 1);
        expect_tok(name, &l, TOKEN_EOF, NULL, 0, 0);
    }

    {
        const char *name = "block_comment_updates_line_col";
        const char *src = "/*x\ny*/z";
        struct Lexer l = al_lexer_alloc(src, strlen(src));

        expect_tok(name, &l, TOKEN_COMMENT, "/*x\ny*/", 1, 1);
        /* After the newline inside the comment, ’z’ should be on line 2. */
        expect_tok(name, &l, TOKEN_IDENTIFIER, "z", 2, 4);
        expect_tok(name, &l, TOKEN_EOF, NULL, 0, 0);
    }
}

static void test_string_and_char_literals(void)
```

```
{
    const char *name = "string_and_char_literals";
    const char *src = "\"abc\" ’x’ \"unterminated\n";
    struct Lexer l = al_lexer_alloc(src, strlen(src));

    expect_tok(name, &l, TOKEN_STRING_LIT, "\"abc\"", 0, 0);
    expect_tok(name, &l, TOKEN_CHAR_LIT, "’x’", 0, 0);

    /* Lexer stops string literal on ’\n’ if not closed. Still TOKEN_STRING_LIT. */
    expect_tok(name, &l, TOKEN_STRING_LIT, "\"unterminated", 0, 0);
    expect_tok(name, &l, TOKEN_EOF, NULL, 0, 0);
}

static void test_literal_operators_longest_match(void)
{
    const char *name = "literal_operators_longest_match";
    const char *src =
        "( ) [ ] { } ... . , ? : == != = ! ; \\ -> "
        "> >= < <= «= »= ++ -- « » "
        "+= -= *= /= %= &= |= ^= || && | & ^ ~ "
        "+ - * / %";
    struct Lexer l = al_lexer_alloc(src, strlen(src));

    expect_tok(name, &l, TOKEN_LPAREN, "(", 0, 0);
    expect_tok(name, &l, TOKEN_RPAREN, ")", 0, 0);
    expect_tok(name, &l, TOKEN_LBRACKET, "[", 0, 0);
    expect_tok(name, &l, TOKEN_RBRACKET, "]", 0, 0);
    expect_tok(name, &l, TOKEN_LBRACE, "{", 0, 0);
    expect_tok(name, &l, TOKEN_RBRACE, "}", 0, 0);

    expect_tok(name, &l, TOKEN_ELLIPSIS, "...", 0, 0);
    expect_tok(name, &l, TOKEN_DOT, ".", 0, 0);
    expect_tok(name, &l, TOKEN_COMMA, ",", 0, 0);
    expect_tok(name, &l, TOKEN_QUESTION, "?", 0, 0);
    expect_tok(name, &l, TOKEN_COLON, ":", 0, 0);

    expect_tok(name, &l, TOKEN_EQEQ, "==", 0, 0);
    expect_tok(name, &l, TOKEN_NE, "!=", 0, 0);
    expect_tok(name, &l, TOKEN_EQ, "=", 0, 0);
    expect_tok(name, &l, TOKEN_BANG, "!", 0, 0);
    expect_tok(name, &l, TOKEN_SEMICOLON, ";", 0, 0);
    expect_tok(name, &l, TOKEN_BSLASH, "\\", 0, 0);
    expect_tok(name, &l, TOKEN_ARROW, "->", 0, 0);

    expect_tok(name, &l, TOKEN_GT, ">", 0, 0);
    expect_tok(name, &l, TOKEN_GE, ">=", 0, 0);
    expect_tok(name, &l, TOKEN_LT, "<", 0, 0);
    expect_tok(name, &l, TOKEN_LE, "<=", 0, 0);

    expect_tok(name, &l, TOKEN_LSHIFTEQ, "«=", 0, 0);
    expect_tok(name, &l, TOKEN_RSHIFTEQ, "»=", 0, 0);

    expect_tok(name, &l, TOKEN_PLUSPLUS, "++", 0, 0);
    expect_tok(name, &l, TOKEN_MINUSMINUS, "--", 0, 0);
    expect_tok(name, &l, TOKEN_LSHIFT, "«", 0, 0);
    expect_tok(name, &l, TOKEN_RSHIFT, "»", 0, 0);

    expect_tok(name, &l, TOKEN_PLUSEQ, "+=", 0, 0);
    expect_tok(name, &l, TOKEN_MINUSEQ, "-=", 0, 0);
    expect_tok(name, &l, TOKEN_STAREQ, "*=", 0, 0);
    expect_tok(name, &l, TOKEN_SLASHEQ, "/=", 0, 0);
    expect_tok(name, &l, TOKEN_PERCENTEQ, "%=", 0, 0);
    expect_tok(name, &l, TOKEN_ANDEQ, "&=", 0, 0);
    expect_tok(name, &l, TOKEN_OREQ, "|=", 0, 0);
    expect_tok(name, &l, TOKEN_XOREQ, "^=", 0, 0);

    expect_tok(name, &l, TOKEN_OROR, "||", 0, 0);
    expect_tok(name, &l, TOKEN_ANDAND, "&&", 0, 0);
    expect_tok(name, &l, TOKEN_BITOR, "|", 0, 0);
    expect_tok(name, &l, TOKEN_BITAND, "&", 0, 0);
    expect_tok(name, &l, TOKEN_CARET, "^", 0, 0);
    expect_tok(name, &l, TOKEN_TILDE, "~", 0, 0);

    expect_tok(name, &l, TOKEN_PLUS, "+", 0, 0);
    expect_tok(name, &l, TOKEN_MINUS, "-", 0, 0);
    expect_tok(name, &l, TOKEN_STAR, "*", 0, 0);
    expect_tok(name, &l, TOKEN_SLASH, "/", 0, 0);
    expect_tok(name, &l, TOKEN_PERCENT, "%", 0, 0);

    expect_tok(name, &l, TOKEN_EOF, NULL, 0, 0);
}

static void test_numbers_valid_and_invalid(void)
{
    const char *name = "numbers_valid_and_invalid";
    const char *src =
        "0 123 1. .5 1.5 "
        "1e3 1e+3 1e-3 1e 1e+ "
        "0xFF 0x1.fp3 0x1.fp 0x "
        "0b1011 0b "
        "0o77 0o "
        "42u 42ULL "
        "3.14f 2.0L "
        ". .0";
    struct Lexer l = al_lexer_alloc(src, strlen(src));

    expect_tok(name, &l, TOKEN_NUMBER, "0", 0, 0);
    expect_tok(name, &l, TOKEN_NUMBER, "123", 0, 0);
    expect_tok(name, &l, TOKEN_NUMBER, "1.", 0, 0);
    expect_tok(name, &l, TOKEN_NUMBER, ".5", 0, 0);
    expect_tok(name, &l, TOKEN_NUMBER, "1.5", 0, 0);

    expect_tok(name, &l, TOKEN_NUMBER, "1e3", 0, 0);
    expect_tok(name, &l, TOKEN_NUMBER, "1e+3", 0, 0);
    expect_tok(name, &l, TOKEN_NUMBER, "1e-3", 0, 0);
    expect_tok(name, &l, TOKEN_INVALID, "1e", 0, 0);
    expect_tok(name, &l, TOKEN_INVALID, "1e+", 0, 0);

    expect_tok(name, &l, TOKEN_NUMBER, "0xFF", 0, 0);
    expect_tok(name, &l, TOKEN_NUMBER, "0x1.fp3", 0, 0);
    expect_tok(name, &l, TOKEN_INVALID, "0x1.fp", 0, 0);
    expect_tok(name, &l, TOKEN_INVALID, "0x", 0, 0);

    expect_tok(name, &l, TOKEN_NUMBER, "0b1011", 0, 0);
    expect_tok(name, &l, TOKEN_INVALID, "0b", 0, 0);

    expect_tok(name, &l, TOKEN_NUMBER, "0o77", 0, 0);
    expect_tok(name, &l, TOKEN_INVALID, "0o", 0, 0);

    expect_tok(name, &l, TOKEN_NUMBER, "42u", 0, 0);
    expect_tok(name, &l, TOKEN_NUMBER, "42ULL", 0, 0);

    expect_tok(name, &l, TOKEN_NUMBER, "3.14f", 0, 0);
    expect_tok(name, &l, TOKEN_NUMBER, "2.0L", 0, 0);

    /* '.' alone should be DOT, but '.0' should be NUMBER. */
    expect_tok(name, &l, TOKEN_DOT, ".", 0, 0);
    expect_tok(name, &l, TOKEN_NUMBER, ".0", 0, 0);

    expect_tok(name, &l, TOKEN_EOF, NULL, 0, 0);
}

static void test_invalid_single_char(void)
{
    const char *name = "invalid_single_char";
    const char *src = "@";
    struct Lexer l = al_lexer_alloc(src, strlen(src));

    expect_tok(name, &l, TOKEN_INVALID, "@", 1, 1);
    expect_tok(name, &l, TOKEN_EOF, NULL, 0, 0);
}

static void test_keyword_vs_identifier_prefix(void)
{
    const char *name = "keyword_vs_identifier_prefix";
    const char *src =
        "int intensity return return_ goto goto1 _x x_1 __ __9 a9 _9";
    struct Lexer l = al_lexer_alloc(src, strlen(src));

    expect_tok(name, &l, TOKEN_KEYWORD, "int", 0, 0);
    expect_tok(name, &l, TOKEN_IDENTIFIER, "intensity", 0, 0);

    expect_tok(name, &l, TOKEN_KEYWORD, "return", 0, 0);
    expect_tok(name, &l, TOKEN_IDENTIFIER, "return_", 0, 0);

    expect_tok(name, &l, TOKEN_KEYWORD, "goto", 0, 0);
    expect_tok(name, &l, TOKEN_IDENTIFIER, "goto1", 0, 0);

    expect_tok(name, &l, TOKEN_IDENTIFIER, "_x", 0, 0);
    expect_tok(name, &l, TOKEN_IDENTIFIER, "x_1", 0, 0);
    expect_tok(name, &l, TOKEN_IDENTIFIER, "__", 0, 0);
    expect_tok(name, &l, TOKEN_IDENTIFIER, "__9", 0, 0);
    expect_tok(name, &l, TOKEN_IDENTIFIER, "a9", 0, 0);
    expect_tok(name, &l, TOKEN_IDENTIFIER, "_9", 0, 0);

    expect_tok(name, &l, TOKEN_EOF, NULL, 0, 0);
}

static void test_hash_not_pp_directive_when_not_column1(void)
{
    const char *name = "hash_not_pp_directive_when_not_column1";
    const char *src = "  #define X 1\n#define Y 2\n";
    struct Lexer l = al_lexer_alloc(src, strlen(src));

```

```c
    /* Because of leading spaces, ’#’ is not at col 1 => NOT a PP directive. */
    expect_tok(name, &l, TOKEN_HASH, "#", 1, 3);
    expect_tok(name, &l, TOKEN_IDENTIFIER, "define", 1, 4);
    expect_tok(name, &l, TOKEN_IDENTIFIER, "X", 1, 11);
    expect_tok(name, &l, TOKEN_NUMBER, "1", 1, 13);

    /* This one is at col 1 and should be treated as a directive (includes ’\n’). */
    expect_tok(name, &l, TOKEN_PP_DIRECTIVE, "#define Y 2\n", 2, 1);
    expect_tok(name, &l, TOKEN_EOF, NULL, 0, 0);
}

static void test_unterminated_block_comment(void)
{
    const char *name = "unterminated_block_comment";
    const char *src = "/* unterminated";
    struct Lexer l = al_lexer_alloc(src, strlen(src));

    /* Lexer consumes to EOF and still returns TOKEN_COMMENT. */
    expect_tok(name, &l, TOKEN_COMMENT, "/* unterminated", 1, 1);
    expect_tok(name, &l, TOKEN_EOF, NULL, 0, 0);
}

static void test_hex_float_variants(void)
{
    const char *name = "hex_float_variants";
    const char *src =
        "0x1p2 0x1p+2 0x1p-2 0x.1p1 0x.p1 0xp1 0x1.0p0 0x1.0 0x1";
    struct Lexer l = al_lexer_alloc(src, strlen(src));

    expect_tok(name, &l, TOKEN_NUMBER, "0x1p2", 0, 0);
    expect_tok(name, &l, TOKEN_NUMBER, "0x1p+2", 0, 0);
    expect_tok(name, &l, TOKEN_NUMBER, "0x1p-2", 0, 0);
    expect_tok(name, &l, TOKEN_NUMBER, "0x.1p1", 0, 0);

    /* Invalid: dot in hex mantissa but no digits before/after the dot */
    expect_tok(name, &l, TOKEN_INVALID, "0x.p1", 0, 0);
    /* Invalid: no mantissa digits (even though exponent is present) */
    expect_tok(name, &l, TOKEN_INVALID, "0xp1", 0, 0);

    expect_tok(name, &l, TOKEN_NUMBER, "0x1.0p0", 0, 0);
    /* Invalid: ’.’ in hex mantissa requires p/P exponent in this lexer */
    expect_tok(name, &l, TOKEN_INVALID, "0x1.0", 0, 0);
    /* Plain hex integer is valid */
    expect_tok(name, &l, TOKEN_NUMBER, "0x1", 0, 0);

    expect_tok(name, &l, TOKEN_EOF, NULL, 0, 0);
}

static void test_number_stops_on_invalid_digit_in_base(void)
{
    const char *name = "number_stops_on_invalid_digit_in_base";
    const char *src = "0b102 0o78";
    struct Lexer l = al_lexer_alloc(src, strlen(src));

    /*
        Current behavior: it tokenizes the longest valid prefix for the base.
        This test documents that behavior (rather than forcing it to be invalid).
    */
    expect_tok(name, &l, TOKEN_NUMBER, "0b10", 0, 0);
    expect_tok(name, &l, TOKEN_NUMBER, "2", 0, 0);

    expect_tok(name, &l, TOKEN_NUMBER, "0o7", 0, 0);
    expect_tok(name, &l, TOKEN_NUMBER, "8", 0, 0);

    expect_tok(name, &l, TOKEN_EOF, NULL, 0, 0);
}

static void test_helpers_direct(void)
{
    const char *name = "helpers_direct";
    AL_UNUSED(name);

    /* al_is_identifier / al_is_identifier_start */
    assert(al_is_identifier_start(’_’));
    assert(al_is_identifier(’_’));
    assert(al_is_identifier(’a’));
    assert(al_is_identifier(’Z’));
    assert(al_is_identifier(’9’));
    assert(!al_is_identifier_start(’9’));

    /* al_lexer_start_with: empty prefix path */
    {
        struct Lexer l = al_lexer_alloc("abc123", 6);
        assert(al_lexer_start_with(&l, ""));
        assert(al_lexer_start_with(&l, "ab"));
        assert(!al_lexer_start_with(&l, "abcd"));
    }

    /* al_lexer_chop_while + al_lexer_peek */
    {
        struct Lexer l = al_lexer_alloc("abc123", 6);
        al_lexer_chop_while(&l, asm_isalpha);
        assert(l.cursor == 3);
        assert(al_lexer_peek(&l, 0) == ’1’);
        assert(al_lexer_peek(&l, 100) == ’\0’);
    }

    /* al_lexer_chop_char newline bookkeeping */
    {
        struct Lexer l = al_lexer_alloc("x\ny", 3);
        assert(l.line_num == 0);
        assert(l.begining_of_line == 0);
        (void)al_lexer_chop_char(&l); /* ’x’ */
        (void)al_lexer_chop_char(&l); /* ’\n’ */
        assert(l.line_num == 1);
        assert(l.begining_of_line == 2);
    }
}

int main(void)
{
    test_basic_program();
    test_pp_directive_and_locations();
    test_whitespace_location_math();
    test_comments();
    test_string_and_char_literals();
    test_literal_operators_longest_match();
    test_numbers_valid_and_invalid();
    test_keyword_vs_identifier_prefix();
    test_hash_not_pp_directive_when_not_column1();
    test_unterminated_block_comment();
    test_hex_float_variants();
    test_number_stops_on_invalid_digit_in_base();
    test_helpers_direct();
    test_invalid_single_char();

    printf("All lexer tests passed.\n");
    return 0;
}
```

Index