Almog JSON Parser

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 243 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 248 of file Almog_Lexer.h.

Referenced by al_lexer_chop_char().

3.1.2.2 content

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

Definition at line 244 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 245 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 246 of file Almog_Lexer.h.

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

3.1.2.5 line_num

```
size_t Lexer::line_num
```

Definition at line 247 of file Almog_Lexer.h.

Referenced by al_lexer_chop_char().

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

- Almog_Lexer.h

3.2 Literal_Token Struct Reference

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

`#include <Almog_Lexer.h>`

Public Attributes

- enum Token_Kind kind
- const char *const text

3.2.1 Detailed Description

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

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

Note

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

Definition at line 165 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 993 of file Almog_Lexer.h.

3.2.2.2 text

`const char* const Literal_Token::text`

Definition at line 167 of file Almog_Lexer.h.

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

- Almog_Lexer.h

3.3 Location Struct Reference

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

`#include <Almog_Lexer.h>`

Public Attributes

- size_t line_num
- size_t col

3.3.1 Detailed Description

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

`al_lexer_next_token()` stores:

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

Definition at line 178 of file Almog_Lexer.h.

3.3.2 Member Data Documentation

3.3.2.1 col

`size_t Location::col`

Definition at line 180 of file Almog_Lexer.h.

Referenced by ajp_expect_token(), ajp_unknown_key(), and al_token_print().

3.3.2.2 line_num

`size_t Location::line_num`

Definition at line 179 of file Almog_Lexer.h.

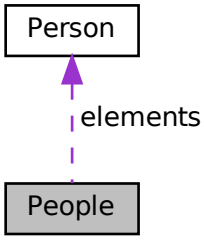
Referenced by ajp_expect_token(), ajp_unknown_key(), and al_token_print().

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

- Almog_Lexer.h

3.4 People Struct Reference

Collaboration diagram for People:



Public Attributes

- size_t length
- size_t capacity
- struct Person * elements

3.4.1 Detailed Description

Definition at line 13 of file temp.c.

3.4.2 Member Data Documentation

3.4.2.1 capacity

```
size_t People::capacity
```

Definition at line 15 of file temp.c.

3.4.2.2 elements

```
struct Person* People::elements
```

Definition at line 16 of file temp.c.

Referenced by free_people(), and main().

3.4.2.3 length

```
size_t People::length
```

Definition at line 14 of file temp.c.

Referenced by free_people(), and main().

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

- temp.c

3.5 Person Struct Reference

Public Attributes

- const char * name
- int age
- const char * location
- int body_count

3.5.1 Detailed Description

Definition at line 6 of file temp.c.

3.5.2 Member Data Documentation

3.5.2.1 age

```
int Person::age
```

Definition at line 8 of file temp.c.

Referenced by parse_person(), and print_person().

3.5.2.2 body_count

```
int Person::body_count
```

Definition at line 10 of file temp.c.

Referenced by parse_person(), and print_person().

3.5.2.3 location

```
const char* Person::location
```

Definition at line 9 of file temp.c.

Referenced by free_person(), parse_person(), and print_person().

3.5.2.4 name

```
const char* Person::name
```

Definition at line 7 of file temp.c.

Referenced by free_person(), parse_person(), and print_person().

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

- temp.c

3.6 String Struct Reference

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

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

Public Attributes

- size_t length
- size_t capacity
- char * elements

3.6.1 Detailed Description

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

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

Definition at line 190 of file Almog_Lexer.h.

3.6.2 Member Data Documentation

3.6.2.1 capacity

```
size_t String::capacity
```

Definition at line 192 of file Almog_Lexer.h.

3.6.2.2 elements

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

Definition at line 193 of file Almog_Lexer.h.

3.6.2.3 length

```
size_t String::length
```

Definition at line 191 of file Almog_Lexer.h.

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

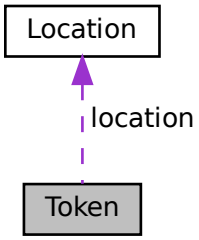
- Almog_Lexer.h

3.7 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

3.7.1 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 202 of file Almog_Lexer.h.

3.7.2 Member Data Documentation

3.7.2.1 kind

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

Definition at line 193 of file Almog_Lexer.h.

Referenced by ajp_array_has_items(), ajp_expect_token(), ajp_object_next_member(), ajp_parse_int(), al_token_kind_name(), and al_token_print().

3.7.2.2 location

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

Definition at line 205 of file Almog_Lexer.h.

Referenced by ajp_expect_token(), ajp_unknown_key(), and al_token_print().

3.7.2.3 text

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

Definition at line 204 of file Almog_Lexer.h.

Referenced by ajp_object_next_member(), ajp_parse_bool(), ajp_parse_int(), ajp_parse_string(), ajp_unknown_key(), and al_token_print().

3.7.2.4 text_len

```
size_t Token::text_len
```

Definition at line 205 of file Almog_Lexer.h.

Referenced by ajp_object_next_member(), ajp_parse_bool(), ajp_parse_string(), ajp_unknown_key(), and al_token_print().

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

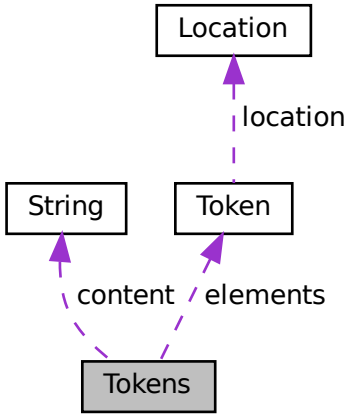
- Almog_Lexer.h

3.8 Tokens Struct Reference

Result of lexing an entire file.

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

Collaboration diagram for Tokens:



Public Attributes

- struct String content
- size_t length
- size_t capacity
- struct Token * elements
- size_t current_token
- const char * current_key
- size_t current_key_len
- char * file_path

3.8.1 Detailed Description

Result of lexing an entire file.

Owns 2 dynamic buffers:

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

Warning

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

Definition at line 221 of file Almog_Lexer.h.

3.8.2 Member Data Documentation

3.8.2.1 capacity

```
size_t Tokens::capacity
```

Definition at line 224 of file Almog_Lexer.h.

3.8.2.2 content

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

Definition at line 205 of file Almog_Lexer.h.

3.8.2.3 current_key

```
const char* Tokens::current_key
```

Definition at line 227 of file Almog_Lexer.h.

Referenced by ajp_object_next_member(), and parse_person().

3.8.2.4 current_key_len

```
size_t Tokens::current_key_len
```

Definition at line 228 of file Almog_Lexer.h.

Referenced by ajp_object_next_member().

3.8.2.5 current_token

```
size_t Tokens::current_token
```

Definition at line 226 of file Almog_Lexer.h.

Referenced by ajp_array_begin(), ajp_array_end(), ajp_array_has_items(), ajp_expect_token(), ajp_get_and_expect_token(), ajp_object_begin(), ajp_object_end(), ajp_object_next_member(), ajp_parse_bool(), ajp_parse_int(), ajp_parse_string(), and ajp_unknown_key().

3.8.2.6 elements

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

Definition at line 225 of file Almog_Lexer.h.

Referenced by ajp_array_has_items(), ajp_expect_token(), ajp_object_next_member(), ajp_parse_bool(), ajp_parse_int(), ajp_parse_string(), and ajp_unknown_key().

3.8.2.7 file_path

```
char* Tokens::file_path
```

Definition at line 229 of file Almog_Lexer.h.

Referenced by ajp_expect_token(), and ajp_unknown_key().

3.8.2.8 length

```
size_t Tokens::length
```

Definition at line 223 of file Almog_Lexer.h.

Referenced by ajp_array_begin(), ajp_array_end(), ajp_array_has_items(), ajp_expect_token(), ajp_get_and_expect_token(), ajp_object_begin(), ajp_object_end(), ajp_object_next_member(), ajp_parse_bool(), ajp_parse_int(), and ajp_parse_string().

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

- Almog_Lexer.h

Chapter 4

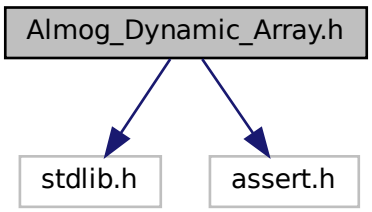
File Documentation

4.1 Almog_Dynamic_Array.h File Reference

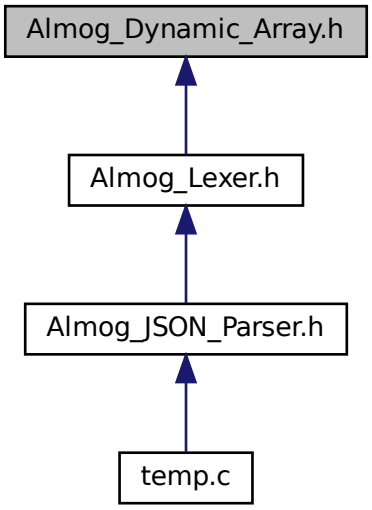
Header-only C macros that implement a simple dynamic array.

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

Include dependency graph for Almog_Dynamic_Array.h:



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



Macros

- #define ADA_INIT_CAPACITY 10

  Default initial capacity used by ada_init_array.
- #define ADA_MALLOC malloc

  Allocation function used by this header (defaults to malloc).
- #define ADA_EXIT exit
- #define ADA_REALLOC realloc

  Reallocation function used by this header (defaults to realloc).
- #define ADA_ASSERT assert

  Assertion macro used by this header (defaults to assert).
- #define ada_init_array(type, header)

  Initialize an array header and allocate its initial storage.
- #define ada_resize(type, header, new_capacity)

  Resize the underlying storage to hold new_capacity elements.
- #define ada_append(type, header, value)

  Append a value to the end of the array, growing if necessary.
- #define ada_insert(type, header, value, index)

  Insert value at position index, preserving order (O(n)).
- #define ada_insert_unordered(type, header, value, index)

  Insert value at index without preserving order (O(1) amortized).
- #define ada_remove(type, header, index)

  Remove element at index, preserving order (O(n)).
- #define ada_remove_unordered(type, header, index)

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

4.1.1 Detailed Description

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

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

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

How to use: 1) Define a header struct with length/capacity/elements fields. 2) Initialize it with ada_init_array(T, header). 3) Modify it with ada_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_appand

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

Value:

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

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

Parameters

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 176 of file Almog_Dynamic_Array.h.

4.1.2.2 ADA_ASSERT

```
#define ADA_ASSERT assert
```

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

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

Definition at line 103 of file Almog_Dynamic_Array.h.

4.1.2.3 ADA_EXIT

```
#define ADA_EXIT exit
```

Definition at line 79 of file Almog_Dynamic_Array.h.

4.1.2.4 ada_init_array

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

Value:

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

Initialize an array header and allocate its initial storage.

Parameters

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 127 of file Almog_Dynamic_Array.h.

4.1.2.5 ADA_INIT_CAPACITY

```
#define ADA_INIT_CAPACITY 10
```

Default initial capacity used by ada_init_array.

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

Definition at line 62 of file Almog_Dynamic_Array.h.

4.1.2.6 ada_insert

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

Value:

```
    do {                                                                \
    ADA_ASSERT((int)(index) >= 0);
          \
    ADA_ASSERT((float)(index) - (int)(index) == 0);
          \
    ada_append(type, (header), (header).elements[(header).length-1]);
                \
    for (int ada_for_loop_index = (header).length-2; ada_for_loop_index > (int)(index);
       ada_for_loop_index--) {   \
         (header).elements[ada_for_loop_index] = (header).elements [ada_for_loop_index-1];
               \
    }
          \
    (header).elements[(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 203 of file Almog_Dynamic_Array.h.

4.1.2.7 ada_insert_unordered

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

Value:

```
    do {   \
    ADA_ASSERT((int)(index) >= 0);                                  \
    ADA_ASSERT((float)(index) - (int)(index) == 0);                 \
    if ((size_t)(index) == (header).length) {                         \
        ada_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 229 of file Almog_Dynamic_Array.h.

4.1.2.8 ADA_MALLOC

```
#define ADA_MALLOC malloc
```

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

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

Definition at line 74 of file Almog_Dynamic_Array.h.

4.1.2.9 ADA_REALLOC

```
#define ADA_REALLOC realloc
```

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

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

Definition at line 91 of file Almog_Dynamic_Array.h.

4.1.2.10 ada_remove

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

Value:

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

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

Parameters

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 253 of file Almog_Dynamic_Array.h.

4.1.2.11 ada_remove_unordered

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

Value:

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

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

Parameters

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 274 of file Almog_Dynamic_Array.h.

4.1.2.12 ada_resize

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

Value:

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

Resize the underlying storage to hold new_capacity elements.

Parameters

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 ADA_EXIT(1).

Note

Reallocation uses ADA_REALLOC and is also checked via ADA_ASSERT.

Definition at line 150 of file Almog_Dynamic_Array.h.

4.2 Almog_Dynamic_Array.h

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

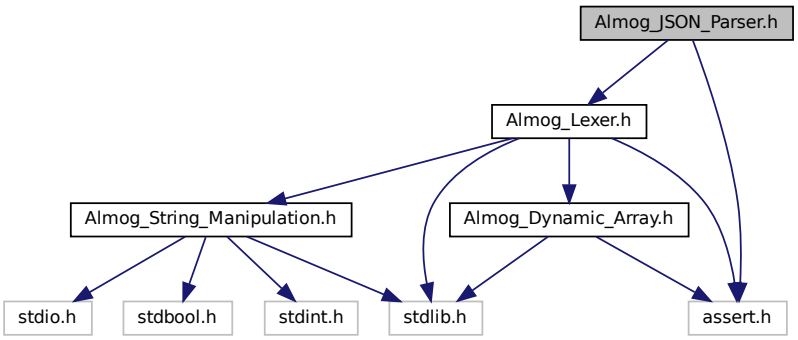
```
00128         (header).capacity = ADA_INIT_CAPACITY;                                          \
00129         (header).length = 0;                                                          \
00130         (header).elements = (type *)ADA_MALLOC(sizeof(type) * (header).capacity);   \
00131         ADA_ASSERT((header).elements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

4.3 Almog_JSON_Parser.h File Reference

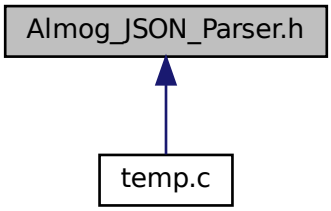
Minimal, token-walking JSON parsing helpers (single-header library).

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

Include dependency graph for Almog_JSON_Parser.h:



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



Macros

- #define AJP_ASSERT assert

  Assertion macro used by the JSON parser (defaults to `assert()`).
- #define ajp_dprintERROR(fmt, ...)

  Print a formatted parser error to stderr with file/line/function info.

Functions

- bool ajp_array_begin (struct Tokens *tokens)

  Consume a JSON array opening bracket "[".
- bool ajp_array_end (struct Tokens *tokens)

  Consume a JSON array closing bracket "]".
- bool ajp_array_has_items (struct Tokens *tokens)

  Decide whether an array has more items to parse, handling commas.
- bool ajp_expect_token (struct Tokens *tokens, enum Token_Kind token_kind)

  Check whether the current token has the expected kind (no consume).
- bool ajp_get_and_expect_token (struct Tokens *tokens, enum Token_Kind token_kind)

  Expect a token kind and advance by one token (consume).
- bool ajp_object_begin (struct Tokens *tokens)

  Consume a JSON object opening brace "{".
- bool ajp_object_end (struct Tokens *tokens)

  Consume a JSON object closing brace "}".
- bool ajp_object_next_member (struct Tokens *tokens)

  Advance to the next object member and expose its key in `tokens`.
- bool ajp_parse_bool (struct Tokens *tokens, bool *boolean)

  Parse a boolean value into `boolean`.
- bool ajp_parse_int (struct Tokens *tokens, int *number)

  Parse an integer value into `number`.
- bool ajp_parse_string (struct Tokens *tokens, const char **string)

  Parse a JSON string into a newly allocated C string.
- void ajp_unknown_key (struct Tokens *tokens)

  Report an "unknown key" error for the current object member.

4.3.1 Detailed Description

Minimal, token-walking JSON parsing helpers (single-header library).

This module implements a small "parser" layer on top of the token stream produced by al_lex_entire_file (see "↩ Almog_Lexer.h").

The API is intentionally low-level: it does not build an AST. Instead, it provides small utilities that:

- check/consume structural tokens (`{`, `}`, `[`, `]`, `,`)
- step through object members
- parse primitive values (string, int, bool)

Usage pattern (typical):

1. Lex a JSON file into `struct Tokens`.
2. Call `ajp_array_begin` / `ajp_object_begin`.
3. Iterate with `ajp_array_has_items` / `ajp_object_next_member`.
4. For each key/value, call a parse function (string/int/bool) and fill your own structs.

Single-header usage:

- In exactly one translation unit, define `ALMOG_JSON_PARSER_IMPLEMENTATION` before including this header to compile the implementation.
- In other translation units include the header normally for declarations.

Important notes / limitations (implementation-defined behavior):

- The parser operates on `struct Tokens` from the lexer and mutates `tokens->current_token` as it consumes tokens.
- Some functions rely on AJP_ASSERT for bounds checks. If assertions are disabled, out-of-bounds reads become possible on malformed input.
- This parser assumes a certain tokenization of JSON by the lexer. (E.g., keys are expected as `TOKEN_STRING_LIT` tokens.)
- Error reporting uses ajp_dprintERROR (prints to stderr).

Note

Inspired by Tsoding's JSON parser approach: https://youtu.be/FBpgdSjJ6nQ

Definition in file Almog_JSON_Parser.h.

4.3.2 Macro Definition Documentation

4.3.2.1 AJP_ASSERT

```
#define AJP_ASSERT assert
```

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

Override by defining `AJP_ASSERT(expr)` before including this header.

Note

This library uses assertions mainly for token-stream bounds checks during parsing helpers. If you compile with `NDEBUG`, these checks may compile out depending on your assert implementation.

Definition at line 57 of file Almog_JSON_Parser.h.

4.3.2.2 ajp_dprintERROR

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

Value:

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

Print a formatted parser error to stderr with file/line/function info.

The printed message format is:

- `__FILE__`, `__LINE__`, `__func__` of the call site,
- then the user-provided formatted message.

Example:

```
ajp_dprintERROR("%s:%zu:%zu: expected %s", path, line, col, what);
```

Warning

This macro always forwards `__VA_ARGS__`. Calling it without any variadic arguments (i.e., only a format string) will not compile.

Definition at line 76 of file Almog_JSON_Parser.h.

4.3.3 Function Documentation

4.3.3.1 ajp_array_begin()

```
bool ajp_array_begin (
              struct Tokens * tokens )
```

Consume a JSON array opening bracket "[".

Equivalent to expecting the current token to be TOKEN_LBRACKET and advancing `tokens->current_token` by 1 on success.

Parameters

tokens	Token stream (mutated).

Returns

true on success, false on mismatch (and prints an error).

Precondition

`tokens` is non-NULL.

`tokens->current_token` points to the next token to be consumed.

Definition at line 112 of file Almog_JSON_Parser.h.

References AJP_ASSERT, ajp_get_and_expect_token(), Tokens::current_token, Tokens::length, and TOKEN_LBRACKET.

Referenced by parse_people().

4.3.3.2 ajp_array_end()

```
bool ajp_array_end (
            struct Tokens * tokens )
```

Consume a JSON array closing bracket ‘]‘.

Equivalent to expecting the current token to be TOKEN_RBRACKET and advancing `tokens->current_token` by 1 on success.

Parameters

tokens	Token stream (mutated).

Returns

true on success, false on mismatch (and prints an error).

Definition at line 128 of file Almog_JSON_Parser.h.

References AJP_ASSERT, ajp_get_and_expect_token(), Tokens::current_token, Tokens::length, and TOKEN_RBRACKET.

Referenced by parse_people().

4.3.3.3 ajp_array_has_items()

```
bool ajp_array_has_items (
            struct Tokens * tokens )
```

Decide whether an array has more items to parse, handling commas.

This helper inspects the current token (without consuming an item value):

- If the token is a comma (`TOKEN_COMMA`), it consumes the comma and returns true (meaning “there is another item”).
- If the token is `TOKEN_RBRACKET`, returns false (meaning “end of array”).
- Otherwise returns true (meaning “an item should follow now”).

Intended loop:

```
ajp_array_begin(tokens);
while (ajp_array_has_items(tokens)) {
    ... parse one value ...
}
ajp_array_end(tokens);
```

Parameters

tokens	Token stream (mutated when consuming commas).

Returns

false when the next token is `]`, true otherwise.

Warning

This function treats "anything other than `]`" as "has items". That means malformed arrays can still return true and later fail in the value parser.

Definition at line 160 of file Almog_JSON_Parser.h.

References AJP_ASSERT, Tokens::current_token, Tokens::elements, Token::kind, Tokens::length, TOKEN_COMMA, and TOKEN_RBRACKET.

Referenced by parse_people().

4.3.3.4 ajp_expect_token()

```
bool ajp_expect_token (
            struct Tokens * tokens,
            enum Token_Kind token_kind )
```

Check whether the current token has the expected kind (no consume).

If the current token kind differs, an error is printed via ajp_dprintERROR and false is returned.

Parameters

tokens	Token stream (not advanced).
token_kind	Expected token kind.

Returns

true if the current token kind matches, false otherwise.

Precondition

`tokens->current_token < tokens->length`.

Definition at line 187 of file Almog_JSON_Parser.h.

References AJP_ASSERT, ajp_dprintERROR, al_token_kind_name(), Location::col, Tokens::current_token, Tokens::elements, Tokens::file_path, Token::kind, Tokens::length, Location::line_num, and Token::location.

Referenced by ajp_get_and_expect_token().

4.3.3.5 ajp_get_and_expect_token()

```
bool ajp_get_and_expect_token (
            struct Tokens * tokens,
            enum Token_Kind token_kind )
```

Expect a token kind and advance by one token (consume).

Semantics:

1. Checks current token kind equals `token_kind` (prints error if not).
2. Advances `tokens->current_token++` unconditionally after the check.

Parameters

tokens	Token stream (mutated).
token_kind	Expected token kind.

Returns

true if the token matched, false otherwise.

Warning

Even on mismatch, this function still advances the token cursor by one. This makes error recovery "skip one token" style.

Definition at line 213 of file Almog_JSON_Parser.h.

References AJP_ASSERT, ajp_expect_token(), Tokens::current_token, and Tokens::length.

Referenced by ajp_array_begin(), ajp_array_end(), ajp_object_begin(), ajp_object_end(), ajp_parse_bool(), and ajp_parse_string().

4.3.3.6 ajp_object_begin()

```
bool ajp_object_begin (
            struct Tokens * tokens )
```

Consume a JSON object opening brace ''{'`.

Equivalent to expecting the current token to be TOKEN_LBRACE and advancing `tokens->current_token` by 1 on success.

Parameters

tokens	Token stream (mutated).

Returns

true on success, false on mismatch (and prints an error).

Definition at line 234 of file Almog_JSON_Parser.h.

References AJP_ASSERT, ajp_get_and_expect_token(), Tokens::current_token, Tokens::length, and TOKEN_LBRACE.

Referenced by parse_person().

4.3.3.7 ajp_object_end()

```
bool ajp_object_end (
            struct Tokens * tokens )
```

Consume a JSON object closing brace ‘}’`.

Equivalent to expecting the current token to be TOKEN_RBRACE and advancing `tokens->current_token` by 1 on success.

Parameters

tokens	Token stream (mutated).

Returns

true on success, false on mismatch (and prints an error).

Definition at line 250 of file Almog_JSON_Parser.h.

References AJP_ASSERT, ajp_get_and_expect_token(), Tokens::current_token, Tokens::length, and TOKEN_RBRACE.

Referenced by parse_person().

4.3.3.8 ajp_object_next_member()

```
bool ajp_object_next_member (
            struct Tokens * tokens )
```

Advance to the next object member and expose its key in `tokens`.

Intended JSON object shape:
`{ "key": value, "key2": value2 }`

Behavior (as implemented):

- If the current token is `TOKEN_COMMA`, it consumes it.
- If the current token is `TOKEN_RBRACE`, returns false (no more members).

- Otherwise it assumes the current token is the key token and:
  - sets `tokens->current_key` / `tokens->current_key_len` from that token
  - advances `tokens->current_token += 2`

The `+= 2` is meant to skip:

- the key token, and
- the colon token

so that after this function returns true, `tokens->current_token` should point at the value token for the member.

Usage:

```
ajp_object_begin(tokens);
while (ajp_object_next_member(tokens)) {
    if (asm_strncmp(tokens->current_key, "name", 4)) {
        ajp_parse_string(tokens, &out->name);
    } else {
        ajp_unknown_key(tokens);
        return false;
    }
}
ajp_object_end(tokens);
```

Parameters

tokens	Token stream (mutated).

Returns

true if a next member is available and the cursor was advanced to its value, false if the next token is `}`.

Warning

This function does not validate that the key is a string literal or that the following token is a colon. If the token stream is not in the expected shape, `current_key` may be nonsensical and the cursor may skip incorrectly.

Definition at line 302 of file Almog_JSON_Parser.h.

References AJP_ASSERT, Tokens::current_key, Tokens::current_key_len, Tokens::current_token, Tokens::elements, Token::kind, Tokens::length, Token::text, Token::text_len, TOKEN_COMMA, and TOKEN_RBRACE.

Referenced by parse_person().

4.3.3.9 ajp_parse_bool()

```
bool ajp_parse_bool (
            struct Tokens * tokens,
            bool * boolean )
```

Parse a boolean value into `boolean`.

Expected token kind (implementation detail): `TOKEN_STRING_LIT`. The token text is compared against `"true"` and `"false"` (case-sensitive).

Parameters

tokens	Token stream (mutated).
boolean	Output pointer; set to true/false on success.

Returns

true on success, false on mismatch/unrecognized spelling.

Warning

As implemented, this function expects the boolean to appear as a string-literal token, not as an identifier token. Ensure your lexer produces `TOKEN_STRING_LIT` for your boolean representation, or adjust the implementation.

Definition at line 335 of file Almog_JSON_Parser.h.

References AJP_ASSERT, ajp_get_and_expect_token(), asm_strncmp(), Tokens::current_token, Tokens::elements, Tokens::length, Token::text, Token::text_len, and TOKEN_STRING_LIT.

4.3.3.10 ajp_parse_int()

```
bool ajp_parse_int (
            struct Tokens * tokens,
            int * number )
```

Parse an integer value into `number`.

Expected shapes (implementation detail):

- `TOKEN_INT_LIT_DEC`
- or a leading sign token (`TOKEN_PLUS` or `TOKEN_MINUS`) in front of an integer literal

The implementation consumes tokens and converts text using `asm_str2int(..., 10)`.

Parameters

tokens	Token stream (mutated).
number	Output pointer; written on success.

Returns

true on success, false on mismatch.

Warning

The current implementation advances the token cursor in a way that can skip extra tokens around signs. Documented here for correctness, but consider revisiting the implementation if you observe misparses.

Definition at line 371 of file Almog_JSON_Parser.h.

References AJP_ASSERT, asm_str2int(), Tokens::current_token, Tokens::elements, Token::kind, Tokens::length, Token::text, TOKEN_INT_LIT_DEC, TOKEN_MINUS, and TOKEN_PLUS.

Referenced by parse_person().

4.3.3.11 ajp_parse_string()

```
bool ajp_parse_string (
            struct Tokens * tokens,
            const char ** string )
```

Parse a JSON string into a newly allocated C string.

Expects a token of kind `TOKEN_STRING_LIT` and duplicates its bytes using `asm_strdup(token.text, token.text_len)`.

Parameters

tokens	Token stream (mutated).
string	Output pointer; receives a heap-allocated, null-terminated string on success.

Returns

true on success, false otherwise.

Postcondition

On success, the caller owns `*string` and must free it using the allocator compatible with `ASM_MALLOC` (used by `asm_strdup`).

Definition at line 407 of file Almog_JSON_Parser.h.

References AJP_ASSERT, ajp_get_and_expect_token(), asm_strdup(), Tokens::current_token, Tokens::elements, Tokens::length, Token::text, Token::text_len, and TOKEN_STRING_LIT.

Referenced by parse_person().

4.3.3.12 ajp_unknown_key()

```
void ajp_unknown_key (
            struct Tokens * tokens )
```

Report an “unknown key” error for the current object member.

This helper prints an error using the token at `tokens->current_token - 2`, which is expected (by this parser’s object iteration scheme) to be the key token of the current member.

Parameters

tokens	Token stream (not advanced).

Note

This function is typically called from user code when the key does not match any expected field.

Definition at line 430 of file Almog_JSON_Parser.h.

References ajp_dprintERROR, Location::col, Tokens::current_token, Tokens::elements, Tokens::file_path, Location::line_num, Token::location, Token::text, and Token::text_len.

Referenced by parse_person().

4.4 Almog_JSON_Parser.h

```
00001
00040 #ifndef ALMOG_JSON_PARSER_H_
00041 #define ALMOG_JSON_PARSER_H_
00042
00043 #include "Almog_Lexer.h"
00044
00055 #ifndef AJP_ASSERT
00056 #include <assert.h>
00057 #define AJP_ASSERT assert
00058 #endif /* AJP_ASSERT */
00059
00076 #define ajp_dprintERROR(fmt, ...) \
00077     fprintf(stderr, "\n%s:%d:\n[Error] in function '%s':\n        " \
00078     fmt "\n\n", __FILE__, __LINE__, __func__, __VA_ARGS__)
00079
00080
00081 bool ajp_array_begin(struct Tokens *tokens);
00082 bool ajp_array_end(struct Tokens *tokens);
00083 bool ajp_array_has_items(struct Tokens *tokens);
00084 bool ajp_expect_token(struct Tokens *tokens, enum Token_Kind token_kind);
00085 bool ajp_get_and_expect_token(struct Tokens *tokens, enum Token_Kind token_kind);
00086 bool ajp_object_begin(struct Tokens *tokens);
00087 bool ajp_object_end(struct Tokens *tokens);
00088 bool ajp_object_next_member(struct Tokens *tokens);
00089 bool ajp_parse_bool(struct Tokens *tokens, bool *boolean);
00090 bool ajp_parse_int(struct Tokens *tokens, int *number);
00091 bool ajp_parse_string(struct Tokens *tokens, const char **string);
00092 void ajp_unknown_key(struct Tokens *tokens);
00093
00094
00095 #endif /*ALMOG_JSON_PARSER_H_*/
00096
00097 #ifdef ALMOG_JSON_PARSER_IMPLEMENTATION
00098 #undef ALMOG_JSON_PARSER_IMPLEMENTATION
00099
00112 bool ajp_array_begin(struct Tokens *tokens)
00113 {
00114     AJP_ASSERT(tokens->current_token < tokens->length - 1);
00115
00116     return ajp_get_and_expect_token(tokens, TOKEN_LBRACKET);
00117 }
00118
00128 bool ajp_array_end(struct Tokens *tokens)
00129 {
00130     AJP_ASSERT(tokens->current_token < tokens->length - 1);
00131
00132     return ajp_get_and_expect_token(tokens, TOKEN_RBRACKET);
00133 }
00134
00160 bool ajp_array_has_items(struct Tokens *tokens)
00161 {
00162     AJP_ASSERT(tokens->current_token < tokens->length - 1);
00163
00164     struct Token current = tokens->elements[tokens->current_token];
00165     if (current.kind == TOKEN_COMMA) {
00166         tokens->current_token++;
00167         return true;
```

```
00168     } else if (current.kind == TOKEN_RBRACKET) {
00169         return false;
00170     }
00171
00172     return true;
00173 }
00174
00187 bool ajp_expect_token(struct Tokens *tokens, enum Token_Kind token_kind)
00188 {
00189     AJP_ASSERT(tokens->current_token < tokens->length);
00190
00191     struct Token current = tokens->elements[tokens->current_token];
00192     if (current.kind != token_kind) {
00193         ajp_dprintERROR("%s:%zu:%zu: expected %s, but got %s.", tokens->file_path,
       current.location.line_num, current.location.col, al_token_kind_name(token_kind),
       al_token_kind_name(current.kind));
00194         return false;
00195     }
00196     return true;
00197 }
00198
00213 bool ajp_get_and_expect_token(struct Tokens *tokens, enum Token_Kind token_kind)
00214 {
00215     AJP_ASSERT(tokens->current_token < tokens->length - 1);
00216
00217     if (tokens->current_token >= tokens->length) {
00218         return false;
00219     }
00220     bool res = ajp_expect_token(tokens, token_kind);
00221     tokens->current_token++;
00222     return res;
00223 }
00224
00234 bool ajp_object_begin(struct Tokens *tokens)
00235 {
00236     AJP_ASSERT(tokens->current_token < tokens->length - 1);
00237
00238     return ajp_get_and_expect_token(tokens, TOKEN_LBRACE);
00239 }
00240
00250 bool ajp_object_end(struct Tokens *tokens)
00251 {
00252     AJP_ASSERT(tokens->current_token < tokens->length - 1);
00253
00254     return ajp_get_and_expect_token(tokens, TOKEN_RBRACE);
00255 }
00256
00302 bool ajp_object_next_member(struct Tokens *tokens)
00303 {
00304     AJP_ASSERT(tokens->current_token < tokens->length - 2);
00305
00306     struct Token current = tokens->elements[tokens->current_token];
00307     if (current.kind == TOKEN_COMMA) {
00308         tokens->current_token += 1;
00309         current = tokens->elements[tokens->current_token];
00310     } else if (current.kind == TOKEN_RBRACE) {
00311         return false;
00312     }
00313
00314     tokens->current_key = current.text;
00315     tokens->current_key_len = current.text_len;
00316     tokens->current_token += 2;
00317     return true;
00318 }
00319
00335 bool ajp_parse_bool(struct Tokens *tokens, bool *boolean)
00336 {
00337     AJP_ASSERT(tokens->current_token < tokens->length - 1);
00338
00339     if (!ajp_get_and_expect_token(tokens, TOKEN_STRING_LIT)) return false;
00340     struct Token current = tokens->elements[tokens->current_token-1];
00341     if (asm_strncmp(current.text, "true", current.text_len)) {
00342         *boolean = true;
00343     } else if (asm_strncmp(current.text, "false", current.text_len)) {
00344         *boolean = false;
00345     } else {
00346         return false;
00347     }
00348
00349     return true;
00350 }
00351
00371 bool ajp_parse_int(struct Tokens *tokens, int *number)
00372 {
00373     AJP_ASSERT(tokens->current_token < tokens->length - 1);
00374
00375     struct Token current = tokens->elements[tokens->current_token];
```

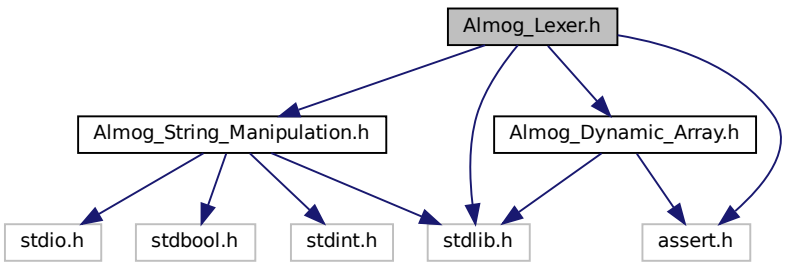
```
00376     tokens->current_token++;
00377     switch (current.kind) {
00378         case TOKEN_PLUS:
00379             tokens->current_token++;
00380             /* fall through */
00381         case TOKEN_MINUS:
00382             tokens->current_token++;
00383             /* fall through */
00384         case TOKEN_INT_LIT_DEC:
00385             *number = asm_str2int(current.text, NULL, 10);
00386             break;
00387         default:
00388             return false;
00389     }
00390     return true;
00391 }
00392
00407 bool ajp_parse_string(struct Tokens *tokens, const char **string)
00408 {
00409     AJP_ASSERT(tokens->current_token < tokens->length - 1);
00410
00411     if (!ajp_get_and_expect_token(tokens, TOKEN_STRING_LIT)) return false;
00412     struct Token current = tokens->elements[tokens->current_token-1];
00413     *string = asm_strdup(current.text, current.text_len);
00414
00415     return true;
00416 }
00417
00430 void ajp_unknown_key(struct Tokens *tokens)
00431 {
00432     struct Token current = tokens->elements[tokens->current_token-2];
00433     ajp_dprintERROR("%s:%zu:%zu: Unexpected filed '%.*s' ", tokens->file_path,
      current.location.line_num, current.location.col, (int)current.text_len, current.text);
00434 }
00435
00436
00437
00438
00439 #endif /*ALMOG_JSON_PARSER_IMPLEMENTATION*/
00440
```

4.5 Almog_Lexer.h File Reference

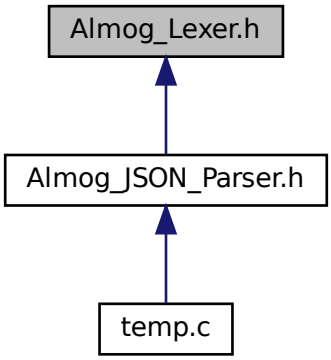
A small single-header lexer for C/C++-like source text.

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

Include dependency graph for Almog_Lexer.h:



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



Classes

- struct Literal_Token

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

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

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

  A token produced by the lexer.
- struct Tokens

  Result of lexing an entire file.
- struct Lexer

  Lexer state over a caller-provided input buffer.

Macros

- #define AL_ASSERT assert

  Assertion macro used by the lexer (defaults to `assert()`).
- #define AL_MALLOC malloc

  Deallocation macro used by al_lexer_alloc() (defaults to `malloc()`).
- #define AL_FREE free

  Deallocation macro used by al_tokens_free() (defaults to `free()`).
- #define literal_tokens_count (sizeof(literal_tokens) / sizeof(literal_tokens[0]))
- #define keywords_count (sizeof(keywords) / sizeof(keywords[0]))
- #define AL_UNUSED(x) (void)x

  Mark a variable as intentionally unused.

Enumerations

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

  Token categories produced by the lexer.

Functions

- bool al_is_identifier (char c)

  Returns whether `c` can appear in an identifier after the first character.
- bool al_is_identifier_start (char c)

  Returns whether `c` can start an identifier.
- struct Tokens al_lex_entire_file (char *file_path)
- struct Lexer al_lexer_alloc (const char *content, size_t len)

  Create a lexer over an input buffer.
- char al_lexer_chop_char (struct Lexer *l)

  Consume and return the next character from the input.
- void al_lexer_chop_while (struct Lexer *l, bool(*pred)(char))

  Consume characters while `pred` returns true for the next character.
- struct Token al_lexer_next_token (struct Lexer *l)

  Return the next token from the input and advance the lexer.
- bool al_lexer_start_with (struct Lexer *l, const char *prefix)

  Check whether the remaining input at the current cursor starts with `prefix`.
- void al_lexer_trim_left (struct Lexer *l)

  Consume leading whitespace characters.
- char al_lexer_peek (const struct Lexer *l, size_t off)

  Peek at a character in the input without advancing the lexer.
- void al_token_print (struct Token tok)

  Print a human-readable representation of `tok` to stdout.
- const char * al_token_kind_name (enum Token_Kind kind)

  Convert a token kind enum to a stable string name.
- struct Tokens al_tokens_alloc (void)
- void al_tokens_free (struct Tokens tokens)

Variables

- static struct Literal_Token literal_tokens [ ]

  Operator/punctuation token table.
- static const char *const keywords [ ]

  List of keywords recognized by the lexer.

4.5.1 Detailed Description

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

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

Note

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

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

Definition in file Almog_Lexer.h.

4.5.2 Macro Definition Documentation

4.5.2.1 AL_ASSERT

```
#define AL_ASSERT assert
```

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

Define `AL_ASSERT` before including this header to override.

Definition at line 30 of file Almog_Lexer.h.

4.5.2.2 AL_FREE

```
#define AL_FREE free
```

Deallocation macro used by al_tokens_free() (defaults to `free()`).

Define `AL_FREE` before including this header to override.

Definition at line 52 of file Almog_Lexer.h.

4.5.2.3 AL_MALLOC

```
#define AL_MALLOC malloc
```

Deallocation macro used by al_lexer_alloc() (defaults to `malloc()`).

Define `AL_MALLOC` before including this header to override.

Definition at line 41 of file Almog_Lexer.h.

4.5.2.4 AL_UNUSED

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

Mark a variable as intentionally unused.

Parameters

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

Definition at line 352 of file Almog_Lexer.h.

4.5.2.5 keywords_count

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

Definition at line 343 of file Almog_Lexer.h.

4.5.2.6 literal_tokens_count

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

Definition at line 311 of file Almog_Lexer.h.

4.5.3 Enumeration Type Documentation

4.5.3.1 Token_Kind

```
enum Token_Kind
```

Token categories produced by the lexer.

The lexer attempts to classify source text into:

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

Enumerator

TOKEN_EOF	
TOKEN_INVALID	
TOKEN_PP_DIRECTIVE	
TOKEN_COMMENT	
TOKEN_STRING_LIT	

Enumerator

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

Enumerator

TOKEN_PERCENTEQ	
TOKEN_ANDEQ	
TOKEN_OREQ	
TOKEN_XOREQ	
TOKEN_LSHIFTEQ	
TOKEN_RSHIFTEQ	
TOKEN_ARROW	
TOKEN_ELLIPSIS	

Definition at line 65 of file Almog_Lexer.h.

4.5.4 Function Documentation

4.5.4.1 al_is_identifier()

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

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

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

Parameters

c	Character to test.

Returns

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

Definition at line 383 of file Almog_Lexer.h.

References asm_isalnum().

4.5.4.2 al_is_identifier_start()

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

Returns whether `c` can start an identifier.

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

Parameters

c	Character to test.

Returns

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

Definition at line 396 of file Almog_Lexer.h.

References asm_isalpha().

4.5.4.3 al_lex_entire_file()

```
struct Tokens al_lex_entire_file (
            char * file_path )
```

Definition at line 396 of file Almog_Lexer.h.

Referenced by main().

4.5.4.4 al_lexer_alloc()

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

Create a lexer over an input buffer.

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

Parameters

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 396 of file Almog_Lexer.h.

4.5.4.5 al_lexer_chop_char()

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

Consume and return the next character from the input.

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

Parameters

l	Lexer to advance.

Returns

The consumed character.

Precondition

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

Definition at line 464 of file Almog_Lexer.h.

References AL_ASSERT, Lexer::begining_of_line, Lexer::content, Lexer::content_len, Lexer::cursor, and Lexer::line_num.

Referenced by al_lexer_chop_while(), and al_lexer_trim_left().

4.5.4.6 al_lexer_chop_while()

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

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

Uses al_lexer_chop_char internally, so newline bookkeeping is applied.

Parameters

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

Definition at line 484 of file Almog_Lexer.h.

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

4.5.4.7 al_lexer_next_token()

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

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

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

The returned token:

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

Tokenization behavior matches the implementation:

- End of input => `TOKEN_EOF`
- Preprocessor directive: a `#` at column 1 (after trimming) consumes until newline (and includes the newline if present) => `TOKEN_PP_DIRECTIVE`
- Identifiers: `[A-Za-z_][A-Za-z0-9_]*` => `TOKEN_IDENTIFIER`, upgraded to `TOKEN_KEYWORD` if it matches an entry in `keywords[]`
- String literal: starts with `"</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 484 of file Almog_Lexer.h.

4.5.4.8 al_lexer_peek()

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

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

Parameters

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 817 of file Almog_Lexer.h.

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

4.5.4.9 al_lexer_start_with()

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

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

Parameters

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 775 of file Almog_Lexer.h.

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

4.5.4.10 al_lexer_trim_left()

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

Consume leading whitespace characters.

Whitespace is defined by `asm_isspace` from "Almog_String_Manipulation.h". Uses al_lexer_chop_char, so newlines update line/column bookkeeping.

Parameters

l	Lexer to advance.

Definition at line 800 of file Almog_Lexer.h.

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

4.5.4.11 al_token_kind_name()

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

Convert a token kind enum to a stable string name.

The returned pointer refers to a string literal.

Parameters

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 849 of file Almog_Lexer.h.

References AL_ASSERT, Token::kind, TOKEN_ANDAND, TOKEN_ANDEQ, TOKEN_ARROW, TOKEN_BANG, TOKEN_BITAND, TOKEN_BITOR, TOKEN_BSLASH, TOKEN_CARET, TOKEN_CHAR_LIT, TOKEN_COLON,

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

Referenced by ajp_expect_token(), and al_token_print().

4.5.4.12 al_token_print()

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

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

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

Note

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

Parameters

tok	Token to print.

Definition at line 835 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.

4.5.4.13 al_tokens_alloc()

```
struct Tokens al_tokens_alloc (
            void  )
```

Definition at line 849 of file Almog_Lexer.h.

4.5.4.14 al_tokens_free()

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

Definition at line 993 of file Almog_Lexer.h.

Referenced by main().

4.5.5 Variable Documentation

4.5.5.1 keywords

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

Initial value:

```
= {
    "auto", "break", "case", "char", "const", "continue", "default", "do", "double",
    "else", "enum", "extern", "float", "for", "goto", "if", "int", "long", "register",
    "return", "short", "signed", "sizeof", "static", "struct", "switch", "typedef",
    "union", "unsigned", "void", "volatile", "while",
}
```

List of keywords recognized by the lexer.

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

Definition at line 320 of file Almog_Lexer.h.

4.5.5.2 literal_tokens

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

Operator/punctuation token table.

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

Note

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

Definition at line 1 of file Almog_Lexer.h.

4.6 Almog_Lexer.h

```
00001
00016 #ifndef ALMOG_LEXER_H_
00017 #define ALMOG_LEXER_H_
00018
00019 #include "Almog_String_Manipulation.h"
00020 #include "Almog_Dynamic_Array.h"
00021
00028 #ifndef AL_ASSERT
00029 #include <assert.h>
00030 #define AL_ASSERT assert
00031 #endif /* AL_ASSERT */
00032
00039 #ifndef AL_MALLOC
00040 #include <stdlib.h>
00041 #define AL_MALLOC malloc
00042 #endif /* AL_MALLOC */
00043
00050 #ifndef AL_FREE
00051 #include <stdlib.h>
00052 #define AL_FREE free
00053 #endif /* AL_FREE */
00054
00065 enum Token_Kind {
00066     /* Sentinel / unknown */
00067     TOKEN_EOF,
00068     TOKEN_INVALID,
00069
00070     /* High-level / multi-char / "word-like" */
00071     TOKEN_PP_DIRECTIVE,
00072     TOKEN_COMMENT,
00073     TOKEN_STRING_LIT,
00074     TOKEN_CHAR_LIT,
00075     TOKEN_INT_LIT_BIN,
00076     TOKEN_INT_LIT_OCT,
00077     TOKEN_INT_LIT_DEC,
00078     TOKEN_INT_LIT_HEX,
00079     TOKEN_FLOAT_LIT_DEC,
00080     TOKEN_FLOAT_LIT_HEX,
00081     TOKEN_KEYWORD,
00082     TOKEN_IDENTIFIER,
00083
00084
00085     /* Grouping / separators */
00086     TOKEN_LPAREN,
00087     TOKEN_RPAREN,
00088     TOKEN_LBRACKET,
00089     TOKEN_RBRACKET,
00090     TOKEN_LBRACE,
00091     TOKEN_RBRACE,
00092
00093     /* Punctuation */
00094     TOKEN_DOT,
00095     TOKEN_COMMA,
00096     TOKEN_SEMICOLON,
00097     TOKEN_BSLASH,
00098     TOKEN_HASH,
00099
00100     /* Ternary */
00101     TOKEN_QUESTION,
00102     TOKEN_COLON,
00103
00104     /* Assignment / equality */
00105     TOKEN_EQ,
00106     TOKEN_EQEQ,
00107     TOKEN_NE,
00108     TOKEN_BANG,
00109
00110     /* Relational */
00111     TOKEN_LT,
00112     TOKEN_GT,
00113     TOKEN_LE,
00114     TOKEN_GE,
00115
00116     /* Bitwise / boolean */
00117     TOKEN_BITAND,
00118     TOKEN_ANDAND,
00119     TOKEN_BITOR,
00120     TOKEN_OROR,
00121     /* Bitwise unary */
00122     TOKEN_CARET,
00123     TOKEN_TILDE,
00124
00125     /* Shifts */
00126     TOKEN_LSHIFT,
00127     TOKEN_RSHIFT,
```

```
00128
00129     /* Inc / dec */
00130     TOKEN_PLUSPLUS,
00131     TOKEN_MINUSMINUS,
00132
00133     /* Arithmetic */
00134     TOKEN_PLUS,
00135     TOKEN_MINUS,
00136     TOKEN_STAR,
00137     TOKEN_SLASH,
00138     TOKEN_PERCENT,
00139
00140     /* Compound assignment */
00141     TOKEN_PLUSEQ,
00142     TOKEN_MINUSEQ,
00143     TOKEN_STAREQ,
00144     TOKEN_SLASHEQ,
00145     TOKEN_PERCENTEQ,
00146     TOKEN_ANDEQ,
00147     TOKEN_OREQ,
00148     TOKEN_XOREQ,
00149     TOKEN_LSHIFTEQ,
00150     TOKEN_RSHIFTEQ,
00151
00152     /* Member access / varargs */
00153     TOKEN_ARROW,
00154     TOKEN_ELLIPSIS,
00155 };
00156
00165 struct Literal_Token {
00166     enum Token_Kind kind;
00167     const char * const text;
00168 };
00169
00178 struct Location {
00179     size_t line_num;
00180     size_t col;
00181 };
00182
00190 struct String {
00191     size_t length;
00192     size_t capacity;
00193     char* elements;
00194 };
00195
00202 struct Token {
00203     enum Token_Kind kind;
00204     const char *text;
00205     size_t text_len;
00206     struct Location location;
00207 };
00208
00221 struct Tokens {
00222     struct String content;
00223     size_t length;
00224     size_t capacity;
00225     struct Token* elements;
00226     size_t current_token;
00227     const char *current_key;
00228     size_t current_key_len;
00229     char *file_path;
00230 };
00231
00243 struct Lexer {
00244     const char * content;
00245     size_t content_len;
00246     size_t cursor;
00247     size_t line_num;
00248     size_t begining_of_line;
00249 };
00250
00260 static struct Literal_Token literal_tokens[] = {
00261     {.text = "("  , .kind = TOKEN_LPAREN},
00262     {.text = ")"  , .kind = TOKEN_RPAREN},
00263     {.text = "["  , .kind = TOKEN_LBRACKET},
00264     {.text = "]"  , .kind = TOKEN_RBRACKET},
00265     {.text = "{"  , .kind = TOKEN_LBRACE},
00266     {.text = "}"  , .kind = TOKEN_RBRACE},
00267     {.text = "#"  , .kind = TOKEN_HASH},
00268     {.text = "...", .kind = TOKEN_ELLIPSIS},
00269     {.text = "."  , .kind = TOKEN_DOT},
00270     {.text = ","  , .kind = TOKEN_COMMA},
00271     {.text = "?"  , .kind = TOKEN_QUESTION},
00272     {.text = ":"  , .kind = TOKEN_COLON},
00273     {.text = "==" , .kind = TOKEN_EQEQ},
00274     {.text = "!=" , .kind = TOKEN_NE},
00275     {.text = "="  , .kind = TOKEN_EQ},
```

```
00276     {.text = "!"  , .kind = TOKEN_BANG},
00277     {.text = ";"  , .kind = TOKEN_SEMICOLON},
00278     {.text = "\\" , .kind = TOKEN_BSLASH},
00279     {.text = "->" , .kind = TOKEN_ARROW},
00280     {.text = ">"  , .kind = TOKEN_GT},
00281     {.text = ">=" , .kind = TOKEN_GE},
00282     {.text = "<"  , .kind = TOKEN_LT},
00283     {.text = "<=" , .kind = TOKEN_LE},
00284     {.text = "«=", .kind = TOKEN_LSHIFTEQ},
00285     {.text = "»=", .kind = TOKEN_RSHIFTEQ},
00286     {.text = "++" , .kind = TOKEN_PLUSPLUS},
00287     {.text = "--" , .kind = TOKEN_MINUSMINUS},
00288     {.text = "«" , .kind = TOKEN_LSHIFT},
00289     {.text = "»" , .kind = TOKEN_RSHIFT},
00290     {.text = "+=", .kind = TOKEN_PLUSEQ},
00291     {.text = "-=", .kind = TOKEN_MINUSEQ},
00292     {.text = "*=", .kind = TOKEN_STAREQ},
00293     {.text = "/=", .kind = TOKEN_SLASHEQ},
00294     {.text = "%=", .kind = TOKEN_PERCENTEQ},
00295     {.text = "&=", .kind = TOKEN_ANDEQ},
00296     {.text = "|=", .kind = TOKEN_OREQ},
00297     {.text = "^=", .kind = TOKEN_XOREQ},
00298     {.text = "||", .kind = TOKEN_OROR},
00299     {.text = "&&", .kind = TOKEN_ANDAND},
00300     {.text = "|" , .kind = TOKEN_BITOR},
00301     {.text = "&" , .kind = TOKEN_BITAND},
00302     {.text = "^" , .kind = TOKEN_CARET},
00303     {.text = "~" , .kind = TOKEN_TILDE},
00304     {.text = "+" , .kind = TOKEN_PLUS},
00305     {.text = "-" , .kind = TOKEN_MINUS},
00306     {.text = "*" , .kind = TOKEN_STAR},
00307     {.text = "/" , .kind = TOKEN_SLASH},
00308     {.text = "%" , .kind = TOKEN_PERCENT},
00309 };
00310
00311 #define literal_tokens_count (sizeof(literal_tokens) / sizeof(literal_tokens[0]))
00312
00319 #ifndef AL_CPP_KEYWORDS
00320 static const char * const keywords[] = {
00321     "auto", "break", "case", "char", "const", "continue", "default", "do", "double",
00322     "else", "enum", "extern", "float", "for", "goto", "if", "int", "long", "register",
00323     "return", "short", "signed", "sizeof", "static", "struct", "switch", "typedef",
00324     "union", "unsigned", "void", "volatile", "while",
00325 };
00326 #else
00327 static const char * const keywords[] = {
00328     "auto", "break", "case", "char", "const", "continue", "default", "do", "double",
00329     "else", "enum", "extern", "float", "for", "goto", "if", "int", "long", "register",
00330     "return", "short", "signed", "sizeof", "static", "struct", "switch", "typedef",
00331     "union", "unsigned", "void", "volatile", "while", "alignas", "alignof", "and",
00332     "and_eq", "asm", "atomic_cancel", "atomic_commit", "atomic_noexcept", "bitand",
00333     "bitor", "bool", "catch", "char16_t", "char32_t", "char8_t", "class", "co_await",
00334     "co_return", "co_yield", "compl", "concept", "const_cast", "consteval", "constexpr",
00335     "constinit", "decltype", "delete", "dynamic_cast", "explicit", "export", "false",
00336     "friend", "inline", "mutable", "namespace", "new", "noexcept", "not", "not_eq",
00337     "nullptr", "operator", "or", "or_eq", "private", "protected", "public", "reflexpr",
00338     "reinterpret_cast", "requires", "static_assert", "static_cast", "synchronized",
00339     "template", "this", "thread_local", "throw", "true", "try", "typeid", "typename",
00340     "using", "virtual", "wchar_t", "xor", "xor_eq",
00341 };
00342 #endif /*AL_CPP_KEYWORDS*/
00343 #define keywords_count (sizeof(keywords) / sizeof(keywords[0]))
00344
00352 #define AL_UNUSED(x) (void)x
00353
00354 bool            al_is_identifier(char c);
00355 bool            al_is_identifier_start(char c);
00356 struct Tokens   al_lex_entire_file(char *file_path);
00357 struct Lexer    al_lexer_alloc(const char *content, size_t len);
00358 char            al_lexer_chop_char(struct Lexer *l);
00359 void            al_lexer_chop_while(struct Lexer *l, bool (*pred)(char));
00360 struct Token    al_lexer_next_token(struct Lexer *l);
00361 bool            al_lexer_start_with(struct Lexer *l, const char *prefix);
00362 void            al_lexer_trim_left(struct Lexer *l);
00363 char            al_lexer_peek(const struct Lexer *l, size_t off);
00364 void            al_token_print(struct Token tok);
00365 const char *    al_token_kind_name(enum Token_Kind kind);
00366 struct Tokens   al_tokens_alloc(void);
00367 void            al_tokens_free(struct Tokens tokens);
00368
00369 #endif /*ALMOG_LEXER_H_*/
00370
00371 #ifdef ALMOG_LEXER_IMPLEMENTATION
00372 #undef ALMOG_LEXER_IMPLEMENTATION
00373
00383 bool al_is_identifier(char c)
00384 {
```

```
00385     return asm_isalnum(c) || c == '_';
00386 }
00387
00396 bool al_is_identifier_start(char c)
00397 {
00398     return asm_isalpha(c) || c == '_';
00399 }
00400
00401 struct Tokens al_lex_entire_file(char *file_path)
00402 {
00403     FILE *fp = fopen(file_path, "r");
00404     if (!fp) {
00405         exit(1);
00406     }
00407
00408     struct Tokens tokens = al_tokens_alloc();
00409     asm_strncpy(tokens.file_path, file_path, ASM_MAX_LEN);
00410
00411     char temp_str[ASM_MAX_LEN];
00412     int len = 0;
00413     while ((len = asm_get_line(fp, temp_str)) != EOF) {
00414         for (int i = 0; i < len; i++) {
00415             ada_append(char, tokens.content, temp_str[i]);
00416         }
00417         ada_append(char, tokens.content, '\n');
00418     }
00419
00420     struct Lexer l = al_lexer_alloc(tokens.content.elements, tokens.content.length);
00421
00422     struct Token t = al_lexer_next_token(&l);
00423     while (t.kind != TOKEN_EOF) {
00424         ada_append(struct Token, tokens, t);
00425         t = al_lexer_next_token(&l);
00426     }
00427     ada_append(struct Token, tokens, t);
00428
00429     return tokens;
00430 }
00431
00442 struct Lexer al_lexer_alloc(const char *content, size_t len)
00443 {
00444     struct Lexer l = {0};
00445     l.content = content;
00446     l.content_len = len;
00447     return l;
00448 }
00449
00464 char al_lexer_chop_char(struct Lexer *l)
00465 {
00466     AL_ASSERT(l->cursor < l->content_len);
00467     char c = l->content[l->cursor++];
00468     if (c == '\n') {
00469         l->line_num++;
00470         l->begining_of_line = l->cursor;
00471     }
00472     return c;
00473 }
00474
00484 void al_lexer_chop_while(struct Lexer *l, bool (*pred)(char))
00485 {
00486     while (l->cursor < l->content_len && pred(l->content[l->cursor])) {
00487         al_lexer_chop_char(l);
00488     }
00489 }
00490
00534 struct Token al_lexer_next_token(struct Lexer *l)
00535 {
00536     al_lexer_trim_left(l);
00537
00538     struct Token token = {
00539         .kind = TOKEN_INVALID,
00540         .text = &(l->content[l->cursor]),
00541         .text_len = 0,
00542         .location.line_num = l->line_num+1,
00543         .location.col = l->cursor - l->begining_of_line+1,
00544     };
00545     size_t start = l->cursor;
00546
00547     if (l->cursor >= l->content_len) {
00548         token.kind = TOKEN_EOF;
00549     } else if (l->content[l->cursor] == '#' && token.location.col == 1) {
00550         token.kind = TOKEN_PP_DIRECTIVE;
00551         for (;l->cursor < l->content_len && l->content[l->cursor] != '\n';) {
00552             al_lexer_chop_char(l);
00553         }
00554         if (l->cursor < l->content_len) {
00555             al_lexer_chop_char(l);
```

```
00556         }
00557     } else if (al_is_identifier_start(l->content[l->cursor])) {
00558         token.kind = TOKEN_IDENTIFIER;
00559         for ( ; l->cursor < l->content_len && al_is_identifier(l->content[l->cursor]); ) {
00560             al_lexer_chop_char(l);
00561         }
00562         {
00563             size_t ident_len = l->cursor - start;
00564             for (size_t i = 0; i < keywords_count; i++) {
00565                 size_t kw_len = asm_length(keywords[i]);
00566                 if (ident_len == kw_len && asm_strncmp(token.text, keywords[i], kw_len)) {
00567                     token.kind = TOKEN_KEYWORD;
00568                     break;
00569                 }
00570             }
00571         }
00572     } else if (l->content[l->cursor] == '"') {
00573         token.kind = TOKEN_STRING_LIT;
00574         al_lexer_chop_char(l);
00575         token.text++;
00576         start = l->cursor+1;
00577 
00578         for ( ; (l->cursor < l->content_len) && (l->content[l->cursor] != '"') &&
    (l->content[l->cursor] != '\n'); ) {
00579             al_lexer_chop_char(l);
00580         }
00581         if ((l->cursor < l->content_len) && (l->content[l->cursor] == '"')) {
00582             al_lexer_chop_char(l);
00583         }
00584     } else if (l->content[l->cursor] == '\'') {
00585         token.kind = TOKEN_CHAR_LIT;
00586         al_lexer_chop_char(l);
00587         token.text++;
00588         start = l->cursor+1;
00589 
00590         for ( ; (l->cursor < l->content_len) && (l->content[l->cursor] != '\'') &&
    (l->content[l->cursor] != '\n'); ) {
00591             al_lexer_chop_char(l);
00592         }
00593         if ((l->cursor < l->content_len) && (l->content[l->cursor] == '\'')) {
00594             al_lexer_chop_char(l);
00595         }
00596     } else if (al_lexer_start_with(l, "//")) {
00597         token.kind = TOKEN_COMMENT;
00598         for (;l->cursor < l->content_len && l->content[l->cursor] != '\n';) {
00599             al_lexer_chop_char(l);
00600         }
00601         if (l->cursor < l->content_len) {
00602             al_lexer_chop_char(l);
00603         }
00604     } else if (al_lexer_start_with(l, "/*")) {
00605         token.kind = TOKEN_COMMENT;
00606         al_lexer_chop_char(l);
00607         al_lexer_chop_char(l);
00608         for ( ; l->cursor < l->content_len; ) {
00609             if ((l->content[l->cursor-1] == '*') && (l->content[l->cursor] == '/')) {
00610                 al_lexer_chop_char(l);
00611                 break;
00612             }
00613             al_lexer_chop_char(l);
00614         }
00615     } else if (asm_isdigit(l->content[l->cursor]) || (l->content[l->cursor] == '.' &&
    asm_isdigit(al_lexer_peek(l, 1)))) {
00616         token.kind = TOKEN_INT_LIT_DEC;
00617         bool is_float = false;
00618         bool invalid = false;
00619 
00620         if (l->content[l->cursor] == '.') {
00621             token.kind = TOKEN_FLOAT_LIT_DEC;
00622             is_float = true;
00623             al_lexer_chop_char(l);
00624             al_lexer_chop_while(l, asm_isdigit);
00625 
00626             /* optional exponent */
00627             if (al_lexer_peek(l, 0) == 'e' || al_lexer_peek(l, 0) == 'E') {
00628                 is_float = true;
00629                 al_lexer_chop_char(l);
00630                 if (al_lexer_peek(l, 0) == '+' || al_lexer_peek(l, 0) == '-') {
00631                     al_lexer_chop_char(l);
00632                 }
00633                 if (!asm_isdigit(al_lexer_peek(l, 0))) {
00634                     invalid = true; /* ".5e" or ".5e+" */
00635                 }
00636                 al_lexer_chop_while(l, asm_isdigit);
00637             }
00638         } else {
00639             /* starts with digit */
```

```
00640             if (al_lexer_peek(l, 0) == '0' && (al_lexer_peek(l, 1) == 'x' || al_lexer_peek(l, 1) ==
      'X')) {
00641                 token.kind = TOKEN_INT_LIT_HEX;
00642                 al_lexer_chop_char(l);
00643                 al_lexer_chop_char(l);
00644
00645                 size_t mantissa_digits = 0;
00646                 while (asm_isXdigit(al_lexer_peek(l, 0)) || asm_isxdigit(al_lexer_peek(l, 0))) {
00647                     mantissa_digits++;
00648                     al_lexer_chop_char(l);
00649                 }
00650                 if (al_lexer_peek(l, 0) == '.') {
00651                     token.kind = TOKEN_FLOAT_LIT_HEX;
00652                     is_float = true;
00653                     al_lexer_chop_char(l);
00654                     while (asm_isXdigit(al_lexer_peek(l, 0)) || asm_isxdigit(al_lexer_peek(l, 0))) {
00655                         mantissa_digits++;
00656                         al_lexer_chop_char(l);
00657                     }
00658                 }
00659                 if (mantissa_digits == 0) {
00660                     invalid = true; /* "0x" or "0x." */
00661                 }
00662
00663                 /* Hex float requires p/P exponent if it's a float form. */
00664                 if (al_lexer_peek(l, 0) == 'p' || al_lexer_peek(l, 0) == 'P') {
00665                     is_float = true;
00666                     al_lexer_chop_char(l);
00667                     if (al_lexer_peek(l, 0) == '+' || al_lexer_peek(l, 0) == '-') {
00668                         al_lexer_chop_char(l);
00669                     }
00670                     if (!asm_isdigit(al_lexer_peek(l, 0))) {
00671                         invalid = true; /* "0x1.fp" / "0x1p+" */
00672                     }
00673                     al_lexer_chop_while(l, asm_isdigit);
00674                 } else if (is_float) {
00675                     /* Had a '.' in hex mantissa but no p-exponent => invalid hex float */
00676                     invalid = true;
00677                 }
00678             } else if (al_lexer_peek(l, 0) == '0' && (al_lexer_peek(l, 1) == 'b' || al_lexer_peek(l,
      1) == 'B')) {
00679                 token.kind = TOKEN_INT_LIT_BIN;
00680                 al_lexer_chop_char(l);
00681                 al_lexer_chop_char(l);
00682                 if (!asm_isbdigit(al_lexer_peek(l, 0))) {
00683                     invalid = true; /* "0b" */
00684                 }
00685                 al_lexer_chop_while(l, asm_isbdigit);
00686             } else if (al_lexer_peek(l, 0) == '0' && (al_lexer_peek(l, 1) == 'o' || al_lexer_peek(l,
      1) == 'O')) {
00687                 token.kind = TOKEN_INT_LIT_OCT;
00688                 al_lexer_chop_char(l);
00689                 al_lexer_chop_char(l);
00690                 if (!asm_isodigit(al_lexer_peek(l, 0))) {
00691                     invalid = true; /* "0o" */
00692                 }
00693                 while (asm_isodigit(al_lexer_peek(l, 0))) {
00694                     al_lexer_chop_char(l);
00695                 }
00696             } else {
00697                 token.kind = TOKEN_INT_LIT_DEC;
00698                 al_lexer_chop_while(l, asm_isdigit);
00699
00700                 if (al_lexer_peek(l, 0) == '.') {
00701                     token.kind = TOKEN_FLOAT_LIT_DEC;
00702                     is_float = true;
00703                     al_lexer_chop_char(l);
00704                     al_lexer_chop_while(l, asm_isdigit);
00705                 }
00706
00707                 if (al_lexer_peek(l, 0) == 'e' || al_lexer_peek(l, 0) == 'E') {
00708                     is_float = true;
00709                     al_lexer_chop_char(l);
00710                     if (al_lexer_peek(l, 0) == '+' || al_lexer_peek(l, 0) == '-') {
00711                         al_lexer_chop_char(l);
00712                     }
00713                     if (!asm_isdigit(al_lexer_peek(l, 0))) {
00714                         invalid = true; /* "1e" / "1e+" */
00715                     }
00716                     al_lexer_chop_while(l, asm_isdigit);
00717                 }
00718             }
00719         }
00720
00721         /* Suffix handling */
00722         if (is_float) {
00723             /* float suffixes: f/F/l/L (accept at most one, but we'll be permissive) */
```

```
00724             while (al_lexer_peek(l, 0) == 'f' || al_lexer_peek(l, 0) == 'F' ||
00725                    al_lexer_peek(l, 0) == 'l' || al_lexer_peek(l, 0) == 'L') {
00726                 al_lexer_chop_char(l);
00727             }
00728         } else {
00729             /* integer suffixes: u/U/l/L/z/Z (permissive) */
00730             while (al_lexer_peek(l, 0) == 'u' || al_lexer_peek(l, 0) == 'U' ||
00731                    al_lexer_peek(l, 0) == 'l' || al_lexer_peek(l, 0) == 'L' ||
00732                    al_lexer_peek(l, 0) == 'z' || al_lexer_peek(l, 0) == 'Z') {
00733                 al_lexer_chop_char(l);
00734             }
00735         }
00736 
00737         if (invalid) token.kind = TOKEN_INVALID;
00738     } else {
00739         size_t longest_matching_token = 0;
00740         enum Token_Kind best_kind = TOKEN_INVALID;
00741         for (size_t i = 0; i < literal_tokens_count; i++) {
00742             if (al_lexer_start_with(l, literal_tokens[i].text)) {
00743                 /* NOTE: assumes that literal_tokens[i].text does not have any '\n' */
00744                 size_t text_len = asm_length(literal_tokens[i].text);
00745                 if (text_len > longest_matching_token) {
00746                     longest_matching_token = text_len;
00747                     best_kind = literal_tokens[i].kind;                }
00748             }
00749         }
00750         if (longest_matching_token > 0) {
00751             token.kind = best_kind;
00752             for (size_t i = 0; i < longest_matching_token; i++) {
00753                 al_lexer_chop_char(l);
00754             }
00755         } else {
00756             token.kind = TOKEN_INVALID;
00757             al_lexer_chop_char(l);
00758         }
00759     }
00760 
00761     token.text_len = l->cursor - start;
00762 
00763     return token;
00764 }
00765 
00775 bool al_lexer_start_with(struct Lexer *l, const char *prefix)
00776 {
00777     size_t prefix_len = asm_length(prefix);
00778     if (prefix_len == 0) {
00779         return true;
00780     }
00781     if (l->cursor + prefix_len > l->content_len) {
00782         return false;
00783     }
00784     for (size_t i = 0; i < prefix_len; i++) {
00785         if (prefix[i] != l->content[l->cursor + i]) {
00786             return false;
00787         }
00788     }
00789     return true;
00790 }
00791 
00800 void al_lexer_trim_left(struct Lexer *l)
00801 {
00802     for (;l->cursor < l->content_len;) {
00803         if (!asm_isspace(l->content[l->cursor])) {
00804             break;
00805         }
00806         al_lexer_chop_char(l);
00807     }
00808 }
00809 
00817 char al_lexer_peek(const struct Lexer *l, size_t off)
00818 {
00819     size_t i = l->cursor + off;
00820     if (i >= l->content_len) return '\0';
00821     return l->content[i];
00822 }
00823 
00835 void al_token_print(struct Token tok)
00836 {
00837     printf("%4zu:%-3zu:(%-19s) -> \"%.*s\"\n", tok.location.line_num, tok.location.col,
      al_token_kind_name(tok.kind), (int)tok.text_len, tok.text);
00838 }
00839 
00849 const char *al_token_kind_name(enum Token_Kind kind)
00850 {
00851     switch (kind) {
00852         case TOKEN_EOF:
00853             return ("TOKEN_EOF");
```

```
00854         case TOKEN_INVALID:
00855             return ("TOKEN_INVALID");
00856         case TOKEN_PP_DIRECTIVE:
00857             return ("TOKEN_PP_DIRECTIVE");
00858         case TOKEN_IDENTIFIER:
00859             return ("TOKEN_IDENTIFIER");
00860         case TOKEN_LPAREN:
00861             return ("TOKEN_LPAREN");
00862         case TOKEN_RPAREN:
00863             return ("TOKEN_RPAREN");
00864         case TOKEN_LBRACKET:
00865             return ("TOKEN_LBRACKET");
00866         case TOKEN_RBRACKET:
00867             return ("TOKEN_RBRACKET");
00868         case TOKEN_LBRACE:
00869             return ("TOKEN_LBRACE");
00870         case TOKEN_RBRACE:
00871             return ("TOKEN_RBRACE");
00872         case TOKEN_DOT:
00873             return ("TOKEN_DOT");
00874         case TOKEN_COMMA:
00875             return ("TOKEN_COMMA");
00876         case TOKEN_SEMICOLON:
00877             return ("TOKEN_SEMICOLON");
00878         case TOKEN_BSLASH:
00879             return ("TOKEN_BSLASH");
00880         case TOKEN_QUESTION:
00881             return ("TOKEN_QUESTION");
00882         case TOKEN_COLON:
00883             return ("TOKEN_COLON");
00884         case TOKEN_LT:
00885             return ("TOKEN_LT");
00886         case TOKEN_GT:
00887             return ("TOKEN_GT");
00888         case TOKEN_GE:
00889             return ("TOKEN_GE");
00890         case TOKEN_LE:
00891             return ("TOKEN_LE");
00892         case TOKEN_KEYWORD:
00893             return ("TOKEN_KEYWORD");
00894         case TOKEN_INT_LIT_BIN:
00895             return ("TOKEN_INT_LIT_BIN");
00896         case TOKEN_INT_LIT_OCT:
00897             return ("TOKEN_INT_LIT_OCT");
00898         case TOKEN_INT_LIT_DEC:
00899             return ("TOKEN_INT_LIT_DEC");
00900         case TOKEN_INT_LIT_HEX:
00901             return ("TOKEN_INT_LIT_HEX");
00902         case TOKEN_FLOAT_LIT_DEC:
00903             return ("TOKEN_FLOAT_LIT_DEC");
00904         case TOKEN_FLOAT_LIT_HEX:
00905             return ("TOKEN_FLOAT_LIT_HEX");
00906         case TOKEN_COMMENT:
00907             return ("TOKEN_COMMENT");
00908         case TOKEN_STRING_LIT:
00909             return ("TOKEN_STRING_LIT");
00910         case TOKEN_CHAR_LIT:
00911             return ("TOKEN_CHAR_LIT");
00912         case TOKEN_EQ:
00913             return ("TOKEN_EQ");
00914         case TOKEN_EQEQ:
00915             return ("TOKEN_EQEQ");
00916         case TOKEN_NE:
00917             return ("TOKEN_NE");
00918         case TOKEN_BANG:
00919             return ("TOKEN_BANG");
00920         case TOKEN_BITAND:
00921             return ("TOKEN_BITAND");
00922         case TOKEN_ANDAND:
00923             return ("TOKEN_ANDAND");
00924         case TOKEN_BITOR:
00925             return ("TOKEN_BITOR");
00926         case TOKEN_OROR:
00927             return ("TOKEN_OROR");
00928         case TOKEN_CARET:
00929             return ("TOKEN_CARET");
00930         case TOKEN_TILDE:
00931             return ("TOKEN_TILDE");
00932         case TOKEN_PLUSPLUS:
00933             return ("TOKEN_PLUSPLUS");
00934         case TOKEN_MINUSMINUS:
00935             return ("TOKEN_MINUSMINUS");
00936         case TOKEN_LSHIFT:
00937             return ("TOKEN_LSHIFT");
00938         case TOKEN_RSHIFT:
00939             return ("TOKEN_RSHIFT");
00940         case TOKEN_PLUS:
```

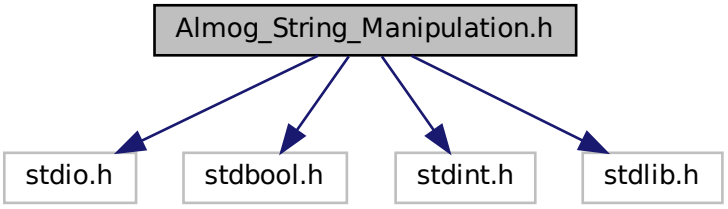
```
00941             return ("TOKEN_PLUS");
00942         case TOKEN_MINUS:
00943             return ("TOKEN_MINUS");
00944         case TOKEN_STAR:
00945             return ("TOKEN_STAR");
00946         case TOKEN_SLASH:
00947             return ("TOKEN_SLASH");
00948         case TOKEN_HASH:
00949             return ("TOKEN_HASH");
00950         case TOKEN_PERCENT:
00951             return ("TOKEN_PERCENT");
00952         case TOKEN_PLUSEQ:
00953             return ("TOKEN_PLUSEQ");
00954         case TOKEN_MINUSEQ:
00955             return ("TOKEN_MINUSEQ");
00956         case TOKEN_STAREQ:
00957             return ("TOKEN_STAREQ");
00958         case TOKEN_SLASHEQ:
00959             return ("TOKEN_SLASHEQ");
00960         case TOKEN_PERCENTEQ:
00961             return ("TOKEN_PERCENTEQ");
00962         case TOKEN_ANDEQ:
00963             return ("TOKEN_ANDEQ");
00964         case TOKEN_OREQ:
00965             return ("TOKEN_OREQ");
00966         case TOKEN_XOREQ:
00967             return ("TOKEN_XOREQ");
00968         case TOKEN_LSHIFTEQ:
00969             return ("TOKEN_LSHIFTEQ");
00970         case TOKEN_RSHIFTEQ:
00971             return ("TOKEN_RSHIFTEQ");
00972         case TOKEN_ARROW:
00973             return ("TOKEN_ARROW");
00974         case TOKEN_ELLIPSIS:
00975             return ("TOKEN_ELLIPSIS");
00976         default:
00977             AL_ASSERT(0 && "Unknown kind");
00978     }
00979     return NULL;
00980 }
00981
00982 struct Tokens al_tokens_alloc(void)
00983 {
00984     struct Tokens tokens = {0};
00985     ada_init_array(struct Token, tokens);
00986     ada_init_array(char, tokens.content);
00987     tokens.current_token = 0;
00988     tokens.file_path = (char *)malloc(sizeof(char) * ASM_MAX_LEN);
00989
00990     return tokens;
00991 }
00992
00993 void al_tokens_free(struct Tokens tokens)
00994 {
00995     AL_FREE(tokens.content.elements);
00996     AL_FREE(tokens.elements);
00997     AL_FREE(tokens.file_path);
00998 }
00999
01000 #endif /*ALMOG_LEXER_IMPLEMENTATION*/
01001
```

4.7 Almog_String_Manipulation.h File Reference

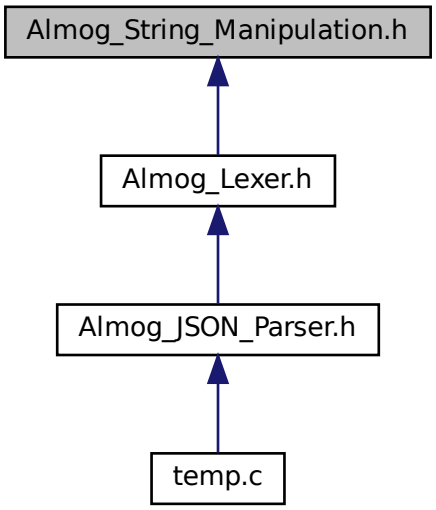
Lightweight string and line manipulation helpers.

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

Include dependency graph for Almog_String_Manipulation.h:



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



Macros

- #define ASM_MALLOC malloc
- #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.
- char * asm_strdup (const char *const s, size_t length)

  Allocate and copy up to `length` characters from `s`.
- int asm_strncat (char *const s1, const char *const s2, const size_t N)

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

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

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

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

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

  Remove leading ASCII whitespace from a string in-place.

4.7.1 Detailed Description

Lightweight string and line manipulation helpers.

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

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

Usage

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

Notes and limitations

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

Definition in file Almog_String_Manipulation.h.

4.7.2 Macro Definition Documentation

4.7.2.1 asm_dprintCHAR

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

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

Parameters

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

Definition at line 88 of file Almog_String_Manipulation.h.

4.7.2.2 asm_dprintDOUBLE

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

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

Parameters

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

Definition at line 115 of file Almog_String_Manipulation.h.

4.7.2.3 asm_dprintERROR

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

Value:

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

Definition at line 126 of file Almog_String_Manipulation.h.

4.7.2.4 asm_dprintFLOAT

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

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

Parameters

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

Definition at line 106 of file Almog_String_Manipulation.h.

4.7.2.5 asm_dprintINT

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

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

Parameters

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

Definition at line 97 of file Almog_String_Manipulation.h.

4.7.2.6 asm_dprintSIZE_T

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

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

Parameters

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

Definition at line 124 of file Almog_String_Manipulation.h.

4.7.2.7 asm_dprintSTRING

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

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

Parameters

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

Definition at line 79 of file Almog_String_Manipulation.h.

4.7.2.8 ASM_MALLOC

```
#define ASM_MALLOC malloc
```

Definition at line 50 of file Almog_String_Manipulation.h.

4.7.2.9 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 154 of file Almog_String_Manipulation.h.

4.7.2.10 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 69 of file Almog_String_Manipulation.h.

4.7.2.11 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 141 of file Almog_String_Manipulation.h.

4.7.3 Function Documentation

4.7.3.1 asm_check_char_belong_to_base()

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

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

Parameters

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 212 of file Almog_String_Manipulation.h.

References asm_dprintERROR, and asm_isdigit().

Referenced by asm_get_char_value_in_base(), asm_str2double(), asm_str2float(), asm_str2int(), and asm_str2size_t().

4.7.3.2 asm_copy_array_by_indexes()

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

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

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

Parameters

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 247 of file Almog_String_Manipulation.h.

4.7.3.3 asm_get_char_value_in_base()

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

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

Parameters

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

Returns -1 if `c` is not valid for `base`.

Definition at line 269 of file Almog_String_Manipulation.h.

References asm_check_char_belong_to_base(), asm_isdigit(), and asm_isupper().

Referenced by asm_str2double(), asm_str2float(), asm_str2int(), and asm_str2size_t().

4.7.3.4 asm_get_line()

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

Read a single line from a stream into a buffer.

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

Parameters

fp	Input stream (must be non-NULL).
dst	Destination buffer. Must have capacity of at least ASM_MAX_LEN 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 301 of file Almog_String_Manipulation.h.

References asm_dprintERROR, and ASM_MAX_LEN.

4.7.3.5 asm_get_next_token_from_str()

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

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

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

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

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

Parameters

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 348 of file Almog_String_Manipulation.h.

Referenced by asm_get_token_and_cut().

4.7.3.6 asm_get_token_and_cut()

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

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

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

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

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

Parameters

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 391 of file Almog_String_Manipulation.h.

References asm_get_next_token_from_str(), and asm_shift_left().

4.7.3.7 asm_isalnum()

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

Test for an alphanumeric character (ASCII).

Parameters

c	Character to test.

Returns

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

Definition at line 412 of file Almog_String_Manipulation.h.

References asm_isalpha(), and asm_isdigit().

Referenced by al_is_identifier().

4.7.3.8 asm_isalpha()

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

Test for an alphabetic character (ASCII).

Parameters

c	Character to test.

Returns

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

Definition at line 423 of file Almog_String_Manipulation.h.

References asm_islower(), and asm_isupper().

Referenced by al_is_identifier_start(), and asm_isalnum().

4.7.3.9 asm_isbdigit()

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

Test for a binary digit (ASCII).

Parameters

c	Character to test.

Returns

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

Definition at line 434 of file Almog_String_Manipulation.h.

4.7.3.10 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 449 of file Almog_String_Manipulation.h.

4.7.3.11 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 464 of file Almog_String_Manipulation.h.

Referenced by asm_check_char_belong_to_base(), asm_get_char_value_in_base(), asm_isalnum(), asm_isxdigit(), and asm_isXdigit().

4.7.3.12 asm_isgraph()

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

Test for any printable character except space (ASCII).

Parameters

c	Character to test.

Returns

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

Definition at line 479 of file Almog_String_Manipulation.h.

Referenced by asm_isprint().

4.7.3.13 asm_islower()

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

Test for a lowercase letter (ASCII).

Parameters

c	Character to test.

Returns

true if c is 'a'–'z'; false otherwise.

Definition at line 494 of file Almog_String_Manipulation.h.

Referenced by asm_isalpha(), and asm_toupper().

4.7.3.14 asm_isodigit()

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

Test for an octal digit (ASCII).

Parameters

c	Character to test.

Returns

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

Definition at line 509 of file Almog_String_Manipulation.h.

4.7.3.15 asm_isprint()

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

Test for any printable character including space (ASCII).

Parameters

c	Character to test.

Returns

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

Definition at line 525 of file Almog_String_Manipulation.h.

References asm_isgraph().

4.7.3.16 asm_ispunct()

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

Test for a punctuation character (ASCII).

Parameters

c	Character to test.

Returns

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

Definition at line 537 of file Almog_String_Manipulation.h.

4.7.3.17 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 553 of file Almog_String_Manipulation.h.

Referenced by al_lexer_trim_left(), asm_str2double(), asm_str2float(), asm_str2int(), asm_str2size_t(), asm_str_is_whitespace(), asm_strip_whitespace(), and asm_trim_left_whitespace().

4.7.3.18 asm_isupper()

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

Test for an uppercase letter (ASCII).

Parameters

c	Character to test.

Returns

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

Definition at line 569 of file Almog_String_Manipulation.h.

Referenced by asm_get_char_value_in_base(), asm_isalpha(), and asm_tolower().

4.7.3.19 asm_isxdigit()

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

Test for a hexadecimal digit (lowercase or decimal).

Parameters

c	Character to test.

Returns

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

Definition at line 584 of file Almog_String_Manipulation.h.

References asm_isdigit().

4.7.3.20 asm_isXdigit()

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

Test for a hexadecimal digit (uppercase or decimal).

Parameters

c	Character to test.

Returns

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

Definition at line 599 of file Almog_String_Manipulation.h.

References asm_isdigit().

4.7.3.21 asm_length()

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

Compute the length of a null-terminated C string.

Parameters

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 618 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_tolower(), asm_toupper(), and asm_trim_left_whitespace().

4.7.3.22 asm_memset()

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

Set a block of memory to a repeated byte value.

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

Parameters

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 653 of file Almog_String_Manipulation.h.

4.7.3.23 asm_pad_left()

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

Left-pad a string in-place.

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

Parameters

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 676 of file Almog_String_Manipulation.h.

References asm_length().

4.7.3.24 asm_print_many_times()

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

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

Parameters

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

Definition at line 693 of file Almog_String_Manipulation.h.

Referenced by print_person().

4.7.3.25 asm_remove_char_from_string()

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

Remove a single character from a string by index.

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

Parameters

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 713 of file Almog_String_Manipulation.h.

References asm_dprintERROR, and asm_length().

Referenced by asm_strip_whitespace().

4.7.3.26 asm_shift_left()

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

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

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

Parameters

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 742 of file Almog_String_Manipulation.h.

References asm_length().

Referenced by asm_get_token_and_cut(), and asm_trim_left_whitespace().

4.7.3.27 asm_str2double()

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

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

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

Parameters

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 816 of file Almog_String_Manipulation.h.

References asm_check_char_belong_to_base(), asm_dprintERROR, asm_get_char_value_in_base(), asm_isspace(), and asm_str2int().

4.7.3.28 asm_str2float()

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

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

Identical to asm_str2double semantically, but returns a float and uses float arithmetic for the fractional part.

Parameters

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 903 of file Almog_String_Manipulation.h.

References asm_check_char_belong_to_base(), asm_dprintERROR, asm_get_char_value_in_base(), asm_isspace(), and asm_str2int().

4.7.3.29 asm_str2int()

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

Convert a string to int in the given base.

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

Parameters

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 977 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 ajp_parse_int(), asm_str2double(), and asm_str2float().

4.7.3.30 asm_str2size_t()

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

Convert a string to size_t in the given base.

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

Parameters

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 1022 of file Almog_String_Manipulation.h.

References asm_check_char_belong_to_base(), asm_dprintERROR, asm_get_char_value_in_base(), and asm_isspace().

4.7.3.31 asm_str_in_str()

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

Count occurrences of a substring within a string.

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

Parameters

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 773 of file Almog_String_Manipulation.h.

References asm_length(), and asm_strncmp().

4.7.3.32 asm_str_is_whitespace()

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

Check whether a string contains only ASCII whitespace characters.

Parameters

s	Null-terminated string to test.

Returns

true if every character in `s` satisfies asm_isspace(), or if `s` is the empty string; false otherwise.

Definition at line 1090 of file Almog_String_Manipulation.h.

References asm_isspace(), and asm_length().

4.7.3.33 asm_strdup()

```
char * asm_strdup (
            const char *const s,
            size_t length )
```

Allocate and copy up to `length` characters from `s`.

Allocates a new buffer of size (length + 1) bytes using ASM_MALLOC, copies up to `length` characters from `s`, and always null-terminates the result.

Parameters

s	Source string (must be null-terminated).
length	Maximum number of characters to copy (excluding '\0').

Returns

Newly allocated string, or NULL if allocation fails.

Note

This is not the same as POSIX strdup(): it does not compute length by itself and may intentionally truncate.

Definition at line 1115 of file Almog_String_Manipulation.h.

References ASM_MALLOC, and asm_strncpy().

Referenced by ajp_parse_string().

4.7.3.34 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 1069 of file Almog_String_Manipulation.h.

References asm_isspace(), asm_length(), and asm_remove_char_from_string().

4.7.3.35 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 1143 of file Almog_String_Manipulation.h.

References asm_dprintERROR, asm_length(), and ASM_MAX_LEN.

4.7.3.36 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 1185 of file Almog_String_Manipulation.h.

References ASM_MAX_LEN.

Referenced by ajp_parse_bool(), asm_str_in_str(), and parse_person().

4.7.3.37 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 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)).

Definition at line 1213 of file Almog_String_Manipulation.h.

Referenced by asm_strdup().

4.7.3.38 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 1231 of file Almog_String_Manipulation.h.

References asm_isupper(), and asm_length().

4.7.3.39 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 1246 of file Almog_String_Manipulation.h.

References asm_islower(), and asm_length().

4.7.3.40 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 1264 of file Almog_String_Manipulation.h.

References asm_isspace(), asm_length(), and asm_shift_left().

4.8 Almog_String_Manipulation.h

```
00001
00041 #ifndef ALMOG_STRING_MANIPULATION_H_
00042 #define ALMOG_STRING_MANIPULATION_H_
00043
00044 #include <stdio.h>
00045 #include <stdbool.h>
00046 #include <stdint.h>
00047
00048 #ifndef ASM_MALLOC
00049 #include <stdlib.h>
00050 #define ASM_MALLOC malloc
00051 #endif
00052
00068 #ifndef ASM_MAX_LEN
00069 #define ASM_MAX_LEN (int)1e3
00070 #endif
00071
00079 #define asm_dprintSTRING(expr) printf(#expr " = %s\n", expr)
00080
00088 #define asm_dprintCHAR(expr) printf(#expr " = %c\n", expr)
00089
00097 #define asm_dprintINT(expr) printf(#expr " = %d\n", expr)
00098
00106 #define asm_dprintFLOAT(expr) printf(#expr " = %#g\n", expr)
00107
00115 #define asm_dprintDOUBLE(expr) printf(#expr " = %#g\n", expr)
00116
00124 #define asm_dprintSIZE_T(expr) printf(#expr " = %zu\n", expr)
00125
00126 #define asm_dprintERROR(fmt, ...) \
00127     fprintf(stderr, "\n%s:%d:\n[Error] in function '%s':\n        " \
00128     fmt "\n\n", __FILE__, __LINE__, __func__, __VA_ARGS__)
00129
00141 #define asm_min(a, b) ((a) < (b) ? (a) : (b))
00142
00154 #define asm_max(a, b) ((a) > (b) ? (a) : (b))
00155
00156 bool    asm_check_char_belong_to_base(const char c, const size_t base);
00157 void    asm_copy_array_by_indexes(char * const target, const int start, const int end, const char *
       const src);
00158 int     asm_get_char_value_in_base(const char c, const size_t base);
00159 int     asm_get_line(FILE *fp, char * const dst);
00160 int     asm_get_next_token_from_str(char * const dst, const char * const src, const char delimiter);
00161 int     asm_get_token_and_cut(char * const dst, char *src, const char delimiter, const bool
       leave_delimiter);
00162 bool    asm_isalnum(const char c);
00163 bool    asm_isalpha(const char c);
00164 bool    asm_isbdigit(const char c);
00165 bool    asm_iscntrl(const char c);
00166 bool    asm_isdigit(const char c);
00167 bool    asm_isgraph(const char c);
00168 bool    asm_islower(const char c);
00169 bool    asm_isodigit(const char c);
00170 bool    asm_isprint(const char c);
00171 bool    asm_ispunct(const char c);
00172 bool    asm_isspace(const char c);
00173 bool    asm_isupper(const char c);
00174 bool    asm_isxdigit(const char c);
00175 bool    asm_isXdigit(const char c);
00176 size_t  asm_length(const char * const str);
00177 void *  asm_memset(void * const des, const unsigned char value, const size_t n);
00178 void    asm_pad_left(char * const s, const size_t padding, const char pad);
00179 void    asm_print_many_times(const char * const str, const size_t n);
00180 void    asm_remove_char_from_string(char * const s, const size_t index);
00181 void    asm_shift_left(char * const s, const size_t shift);
00182 int     asm_str_in_str(const char * const src, const char * const word_to_search);
00183 double  asm_str2double(const char * const s, const char ** const end, const size_t base);
00184 float   asm_str2float(const char * const s, const char ** const end, const size_t base);
00185 int     asm_str2int(const char * const s, const char ** const end, const size_t base);
00186 size_t  asm_str2size_t(const char * const s, const char ** const end, const size_t base);
00187 void    asm_strip_whitespace(char * const s);
00188 bool    asm_str_is_whitespace(const char * const s);
00189 char *  asm_strdup(const char * const s, size_t length);
00190 int     asm_strncat(char * const s1, const char * const s2, const size_t N);
00191 int     asm_strncmp(const char * const s1, const char * const s2, const size_t N);
00192 int     asm_strncpy(char * const s1, const char * const s2, const size_t N);
00193 void    asm_tolower(char * const s);
00194 void    asm_toupper(char * const s);
00195 void    asm_trim_left_whitespace(char *s);
```

```
00196
00197 #endif /*ALMOG_STRING_MANIPULATION_H_*/
00198
00199 #ifdef ALMOG_STRING_MANIPULATION_IMPLEMENTATION
00200 #undef ALMOG_STRING_MANIPULATION_IMPLEMENTATION
00201
00212 bool asm_check_char_belong_to_base(const char c, const size_t base)
00213 {
00214     if (base > 36 || base < 2) {
00215         #ifndef ASM_NO_ERRORS
00216         asm_dprintERROR("Supported bases are [2...36]. Inputted: %zu", base);
00217         #endif
00218         return false;
00219     }
00220     if (base <= 10) {
00221         return c >= '0' && c <= '9'+(char)base-10;
00222     }
00223     if (base > 10) {
00224         return asm_isdigit(c) || (c >= 'A' && c <= ('A'+(char)base-11)) || (c >= 'a' && c <=
      ('a'+(char)base-11));
00225     }
00226
00227     return false;
00228 }
00229
00247 void asm_copy_array_by_indexes(char * const target, const int start, const int end, const char * const
      src)
00248 {
00249     if (start > end) return;
00250     int j = 0;
00251     for (int i = start; i <= end; i++) {
00252         target[j] = src[i];
00253         j++;
00254     }
00255     if (target[j-1] != '\0') {
00256         target[j] = '\0';
00257     }
00258 }
00259
00269 int asm_get_char_value_in_base(const char c, const size_t base)
00270 {
00271     if (!asm_check_char_belong_to_base(c, base)) return -1;
00272     if (asm_isdigit(c)) {
00273         return c - '0';
00274     } else if (asm_isupper(c)) {
00275         return c - 'A' + 10;
00276     } else {
00277         return c - 'a' + 10;
00278     }
00279 }
00280
00301 int asm_get_line(FILE *fp, char * const dst)
00302 {
00303     int i = 0;
00304     int c;
00305     while ((c = fgetc(fp)) != '\n' && c != EOF) {
00306         dst[i++] = c;
00307         if (i >= ASM_MAX_LEN) {
00308             #ifndef ASM_NO_ERRORS
00309             asm_dprintERROR("%s", "index exceeds ASM_MAX_LEN. Line in file is too long.");
00310             #endif
00311             dst[i-1] = '\0';
00312             return -1;
00313         }
00314     }
00315     dst[i] = '\0';
00316     if (c == EOF && i == 0) {
00317         return -1;
00318     }
00319     return i;
00320 }
00321
00348 int asm_get_next_token_from_str(char * const dst, const char * const src, const char delimiter)
00349 {
00350     int i = 0, j = 0;
00351     char c;
00352     while ((c = src[i]) != delimiter && c != '\0') {
00353         dst[j++] = src[i++];
00354     }
00355
00356     dst[j] = '\0';
00357
00358     return j;
00359 }
00360
00391 int asm_get_token_and_cut(char * const dst, char *src, const char delimiter, const bool
      leave_delimiter)
```

```
00392 {
00393     int new_src_start_index = asm_get_next_token_from_str(dst, src, delimiter);
00394     bool delimiter_at_start = src[new_src_start_index] == delimiter;
00395 
00396     if (leave_delimiter) {
00397         asm_shift_left(src, new_src_start_index);
00398     } else if (delimiter_at_start) {
00399         asm_shift_left(src, new_src_start_index + 1);
00400     } else {
00401         src[0] = '\0';
00402     }
00403     return new_src_start_index ? 1 : 0;
00404 }
00405 
00412 bool asm_isalnum(char c)
00413 {
00414     return asm_isalpha(c) || asm_isdigit(c);
00415 }
00416 
00423 bool asm_isalpha(char c)
00424 {
00425     return asm_isupper(c) || asm_islower(c);
00426 }
00427 
00434 bool asm_isbdigit(const char c)
00435 {
00436     if (c == '0' || c == '1') {
00437         return true;
00438     } else {
00439         return false;
00440     }
00441 }
00442 
00449 bool asm_iscntrl(char c)
00450 {
00451     if ((c >= 0 && c <= 31) || c == 127) {
00452         return true;
00453     } else {
00454         return false;
00455     }
00456 }
00457 
00464 bool asm_isdigit(char c)
00465 {
00466     if (c >= '0' && c <= '9') {
00467         return true;
00468     } else {
00469         return false;
00470     }
00471 }
00472 
00479 bool asm_isgraph(char c)
00480 {
00481     if (c >= 33 && c <= 126) {
00482         return true;
00483     } else {
00484         return false;
00485     }
00486 }
00487 
00494 bool asm_islower(char c)
00495 {
00496     if (c >= 'a' && c <= 'z') {
00497         return true;
00498     } else {
00499         return false;
00500     }
00501 }
00502 
00509 bool asm_isodigit(const char c)
00510 {
00511     if ((c >= '0' && c <= '7')) {
00512         return true;
00513     } else {
00514         return false;
00515     }
00516 }
00517 
00525 bool asm_isprint(char c)
00526 {
00527     return asm_isgraph(c) || c == ' ';
00528 }
00529 
00537 bool asm_ispunct(char c)
00538 {
00539     if ((c >= 33 && c <= 47) || (c >= 58 && c <= 64) || (c >= 91 && c <= 96) || (c >= 123 && c <=
       126)) {
```

```
00540         return true;
00541     } else {
00542         return false;
00543     }
00544 }
00545
00553 bool asm_isspace(char c)
00554 {
00555     if (c == ' ' || c == '\n' || c == '\t' ||
00556         c == '\v'|| c == '\f' || c == '\r') {
00557         return true;
00558     } else {
00559         return false;
00560     }
00561 }
00562
00569 bool asm_isupper(char c)
00570 {
00571     if (c >= 'A' && c <= 'Z') {
00572         return true;
00573     } else {
00574         return false;
00575     }
00576 }
00577
00584 bool asm_isxdigit(char c)
00585 {
00586     if ((c >= 'a' && c <= 'f') || asm_isdigit(c)) {
00587         return true;
00588     } else {
00589         return false;
00590     }
00591 }
00592
00599 bool asm_isXdigit(char c)
00600 {
00601     if ((c >= 'A' && c <= 'F') || asm_isdigit(c)) {
00602         return true;
00603     } else {
00604         return false;
00605     }
00606 }
00607
00618 size_t asm_length(const char * const str)
00619 {
00620     char c;
00621     size_t i = 0;
00622
00623     while ((c = str[i++]) != '\0') {
00624         if (i > ASM_MAX_LEN) {
00625             #ifndef ASM_NO_ERRORS
00626             asm_dprintERROR("%s", "index exceeds ASM_MAX_LEN. Probably no NULL termination.");
00627             #endif
00628             return SIZE_MAX;
00629         }
00630     }
00631     return --i;
00632 }
00633
00653 void * asm_memset(void * const des, const unsigned char value, const size_t n)
00654 {
00655     unsigned char *ptr = (unsigned char *)des;
00656     for (size_t i = n; i-- > 0;) {
00657         *ptr++ = value;
00658     }
00659     return des;
00660 }
00661
00676 void asm_pad_left(char * const s, const size_t padding, const char pad)
00677 {
00678     int len = (int)asm_length(s);
00679     for (int i = len; i >= 0; i--) {
00680         s[i+(int)padding] = s[i];
00681     }
00682     for (int i = 0; i < (int)padding; i++) {
00683         s[i] = pad;
00684     }
00685 }
00686
00693 void asm_print_many_times(const char * const str, const size_t n)
00694 {
00695     for (size_t i = 0; i < n; i++) {
00696         printf("%s", str);
00697     }
00698     printf("\n");
00699 }
00700
```

```
00713 void asm_remove_char_from_string(char * const s, const size_t index)
00714 {
00715     size_t len = asm_length(s);
00716     if (len == 0) return;
00717     if (index >= len) {
00718         #ifndef ASM_NO_ERRORS
00719         asm_dprintERROR("%s", "index exceeds array length.");
00720         #endif
00721         return;
00722     }
00723
00724     for (size_t i = index; i < len; i++) {
00725         s[i] = s[i+1];
00726     }
00727 }
00728
00742 void asm_shift_left(char * const s, const size_t shift)
00743 {
00744     size_t len = asm_length(s);
00745
00746     if (shift == 0) return;
00747     if (len <= shift) {
00748         s[0] = '\0';
00749         return;
00750     }
00751
00752     size_t i;
00753     for (i = shift; i < len; i++) {
00754         s[i-shift] = s[i];
00755     }
00756     s[i-shift] = '\0';
00757 }
00758
00773 int asm_str_in_str(const char * const src, const char * const word_to_search)
00774 {
00775     int i = 0, num_of_accur = 0;
00776     while (src[i] != '\0') {
00777         if (asm_strncmp(src+i, word_to_search, asm_length(word_to_search))) {
00778             num_of_accur++;
00779         }
00780         i++;
00781     }
00782     return num_of_accur;
00783 }
00784
00816 double asm_str2double(const char * const s, const char ** const end, const size_t base)
00817 {
00818     if (base < 2 || base > 36) {
00819         #ifndef ASM_NO_ERRORS
00820         asm_dprintERROR("Supported bases are [2...36]. Input: %zu", base);
00821         #endif
00822         if (end) *end = s;
00823         return 0.0;
00824     }
00825     int num_of_whitespace = 0;
00826     while (asm_isspace(s[num_of_whitespace])) {
00827         num_of_whitespace++;
00828     }
00829
00830     int i = 0;
00831     if (s[0+num_of_whitespace] == '-' || s[0+num_of_whitespace] == '+') {
00832         i++;
00833     }
00834     int sign = s[0+num_of_whitespace] == '-' ? -1 : 1;
00835
00836     size_t left = 0;
00837     double right = 0.0;
00838     int expo = 0;
00839     for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
00840         left = base * left + asm_get_char_value_in_base(s[i+num_of_whitespace], base);
00841     }
00842
00843     if (s[i+num_of_whitespace] == '.') {
00844         i++; /* skip the point */
00845
00846         size_t divider = base;
00847         for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
00848             right = right + asm_get_char_value_in_base(s[i+num_of_whitespace], base) /
      (double)divider;
00849             divider *= base;
00850         }
00851     }
00852
00853     if ((s[i+num_of_whitespace] == 'e') || (s[i+num_of_whitespace] == 'E')) {
00854         expo = asm_str2int(&(s[i+num_of_whitespace+1]), end, 10);
00855     } else {
00856         if (end) *end = s + i + num_of_whitespace;
```

```
00857     }
00858
00859     double res = sign * (left + right);
00860
00861     if (expo > 0) {
00862         for (int index = 0; index < expo; index++) {
00863             res *= (double)base;
00864         }
00865     } else {
00866         for (int index = 0; index > expo; index--) {
00867             res /= (double)base;
00868         }
00869     }
00870
00871     return res;
00872 }
00873
00903 float asm_str2float(const char * const s, const char ** const end, const size_t base)
00904 {
00905     if (base < 2 || base > 36) {
00906         #ifndef ASM_NO_ERRORS
00907         asm_dprintERROR("Supported bases are [2...36]. Input: %zu", base);
00908         #endif
00909         if (end) *end = s;
00910         return 0.0f;
00911     }
00912     int num_of_whitespace = 0;
00913     while (asm_isspace(s[num_of_whitespace])) {
00914         num_of_whitespace++;
00915     }
00916
00917     int i = 0;
00918     if (s[0+num_of_whitespace] == '-' || s[0+num_of_whitespace] == '+') {
00919         i++;
00920     }
00921     int sign = s[0+num_of_whitespace] == '-' ? -1 : 1;
00922
00923     int left = 0;
00924     float right = 0.0f;
00925     int expo = 0;
00926     for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
00927         left = base * left + asm_get_char_value_in_base(s[i+num_of_whitespace], base);
00928     }
00929
00930     if (s[i+num_of_whitespace] == '.') {
00931         i++; /* skip the point */
00932
00933         size_t divider = base;
00934         for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
00935             right = right + asm_get_char_value_in_base(s[i+num_of_whitespace], base) / (float)divider;
00936             divider *= base;
00937         }
00938     }
00939
00940     if ((s[i+num_of_whitespace] == 'e') || (s[i+num_of_whitespace] == 'E')) {
00941         expo = asm_str2int(&(s[i+num_of_whitespace+1]), end, 10);
00942     } else {
00943         if (end) *end = s + i + num_of_whitespace;
00944     }
00945
00946     float res = sign * (left + right);
00947
00948     if (expo > 0) {
00949         for (int index = 0; index < expo; index++) {
00950             res *= (float)base;
00951         }
00952     } else {
00953         for (int index = 0; index > expo; index--) {
00954             res /= (float)base;
00955         }
00956     }
00957
00958     return res;
00959 }
00960
00977 int asm_str2int(const char * const s, const char ** const end, const size_t base)
00978 {
00979     if (base < 2 || base > 36) {
00980         #ifndef ASM_NO_ERRORS
00981         asm_dprintERROR("Supported bases are [2...36]. Input: %zu", base);
00982         #endif
00983         if (end) *end = s;
00984         return 0;
00985     }
00986     int num_of_whitespace = 0;
00987     while (asm_isspace(s[num_of_whitespace])) {
00988         num_of_whitespace++;
```

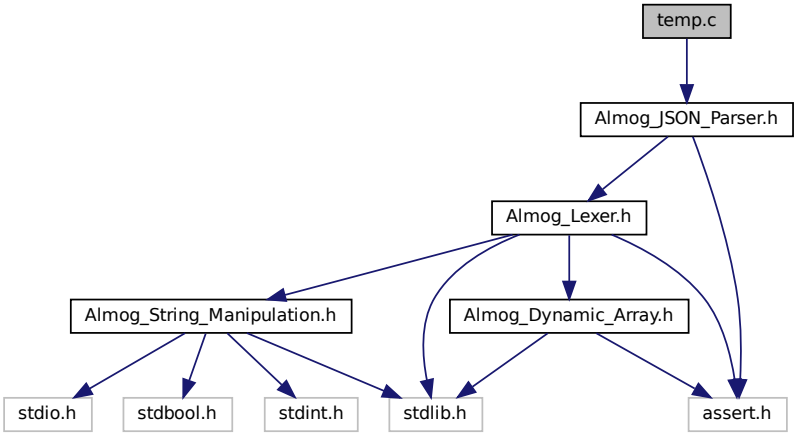
```
00989     }
00990
00991     int n = 0, i = 0;
00992     if (s[0+num_of_whitespace] == '-' || s[0+num_of_whitespace] == '+') {
00993         i++;
00994     }
00995     int sign = s[0+num_of_whitespace] == '-' ? -1 : 1;
00996
00997     for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
00998         n = base * n + asm_get_char_value_in_base(s[i+num_of_whitespace], base);
00999     }
01000
01001     if (end) *end = s + i+num_of_whitespace;
01002
01003     return n * sign;
01004 }
01005
01022 size_t asm_str2size_t(const char * const s, const char ** const end, const size_t base)
01023 {
01024     if (end) *end = s;
01025
01026     int num_of_whitespace = 0;
01027     while (asm_isspace(s[num_of_whitespace])) {
01028         num_of_whitespace++;
01029     }
01030
01031     if (s[0+num_of_whitespace] == '-') {
01032         #ifndef ASM_NO_ERRORS
01033         asm_dprintERROR("%s", "Unable to convert a negative number to size_t.");
01034         #endif
01035         return 0;
01036     }
01037
01038     if (base < 2 || base > 36) {
01039         #ifndef ASM_NO_ERRORS
01040         asm_dprintERROR("Supported bases are [2...36]. Input: %zu", base);
01041         #endif
01042         if (end) *end = s+num_of_whitespace;
01043         return 0;
01044     }
01045
01046     size_t n = 0, i = 0;
01047     if (s[0+num_of_whitespace] == '+') {
01048         i++;
01049     }
01050
01051     for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
01052         n = base * n + asm_get_char_value_in_base(s[i+num_of_whitespace], base);
01053     }
01054
01055     if (end) *end = s + i+num_of_whitespace;
01056
01057     return n;
01058 }
01059
01069 void asm_strip_whitespace(char * const s)
01070 {
01071     size_t len = asm_length(s);
01072     size_t i;
01073     for (i = 0; i < len; i++) {
01074         if (asm_isspace(s[i])) {
01075             asm_remove_char_from_string(s, i);
01076             len--;
01077             i--;
01078         }
01079     }
01080     s[i] = '\0';
01081 }
01082
01090 bool asm_str_is_whitespace(const char * const s)
01091 {
01092     size_t len = asm_length(s);
01093     for (size_t i = 0; i < len; i++) {
01094         if (!asm_isspace(s[i])) {
01095             return false;
01096         }
01097     }
01098
01099     return true;
01100 }
01101
01115 char * asm_strdup(const char * const s, size_t length)
01116 {
01117     char * res = (char *)ASM_MALLOC(sizeof(char) * length+1);
01118     asm_strncpy((char * const)res, s, length);
01119
01120     return res;
```

```
01121 }
01122
01143 int asm_strncat(char * const s1, const char * const s2, const size_t N)
01144 {
01145     size_t len_s1 = asm_length(s1);
01146
01147     int limit = N;
01148     if (limit == 0) {
01149         limit = ASM_MAX_LEN;
01150     }
01151
01152     int i = 0;
01153     while (i < limit && s2[i] != '\0') {
01154         if (len_s1 + (size_t)i >= ASM_MAX_LEN-1) {
01155             #ifndef ASM_NO_ERRORS
01156             asm_dprintERROR("s2 or the first N=%zu digit of s2 does not fit into s1.", N);
01157             #endif
01158             return i;
01159         }
01160
01161         s1[len_s1+(size_t)i] = s2[i];
01162         i++;
01163     }
01164     s1[len_s1+(size_t)i] = '\0';
01165
01166     return i;
01167 }
01168
01185 int asm_strncmp(const char *s1, const char *s2, const size_t N)
01186 {
01187     size_t n = N == 0 ? ASM_MAX_LEN : N;
01188     size_t i = 0;
01189     while (i < n) {
01190         if (s1[i] == '\0' && s2[i] == '\0') {
01191             break;
01192         }
01193         if (s1[i] != s2[i] || (s1[i] == '\0') || (s2[i] == '\0')) {
01194             return 0;
01195         }
01196         i++;
01197     }
01198     return 1;
01199 }
01200
01213 int asm_strncpy(char * const s1, const char * const s2, const size_t N)
01214 {
01215     size_t n = N;
01216
01217     size_t i;
01218     for (i = 0; i < n && s2[i] != '\0'; i++) {
01219         s1[i] = s2[i];
01220     }
01221     s1[i] = '\0';
01222
01223     return i;
01224 }
01225
01231 void asm_tolower(char * const s)
01232 {
01233     size_t len = asm_length(s);
01234     for (size_t i = 0; i < len; i++) {
01235         if (asm_isupper(s[i])) {
01236             s[i] += 'a' - 'A';
01237         }
01238     }
01239 }
01240
01246 void asm_toupper(char * const s)
01247 {
01248     size_t len = asm_length(s);
01249     for (size_t i = 0; i < len; i++) {
01250         if (asm_islower(s[i])) {
01251             s[i] += 'A' - 'a';
01252         }
01253     }
01254 }
01255
01264 void asm_trim_left_whitespace(char * const s)
01265 {
01266     size_t len = asm_length(s);
01267
01268     if (len == 0) return;
01269     size_t i;
01270     for (i = 0; i < len; i++) {
01271         if (!asm_isspace(s[i])) {
01272             break;
01273         }
```

```
01274     }
01275     asm_shift_left(s, i);
01276 }
01277
01278 #ifdef ASM_NO_ERRORS
01279 #undef ASM_NO_ERRORS
01280 #endif
01281
01282 #endif /*ALMOG_STRING_MANIPULATION_IMPLEMENTATION*/
01283
```

4.9 temp.c File Reference

`#include "Almog_JSON_Parser.h"`
Include dependency graph for temp.c:



Classes

- struct Person
- struct People

Macros

- #define ALMOG_STRING_MANIPULATION_IMPLEMENTATION
- #define ALMOG_LEXER_IMPLEMENTATION
- #define ALMOG_JSON_PARSER_IMPLEMENTATION

Functions

- void print_person (struct Person p)
- void free_person (struct Person p)
- void free_people (struct People people)
- bool parse_person (struct Tokens *tokens, struct Person *p)
- bool parse_people (struct Tokens *tokens, struct People *people)
- int main (void)

4.9.1 Macro Definition Documentation

4.9.1.1 ALMOG_JSON_PARSER_IMPLEMENTATION

```
#define ALMOG_JSON_PARSER_IMPLEMENTATION
```

Definition at line 3 of file temp.c.

4.9.1.2 ALMOG_LEXER_IMPLEMENTATION

```
#define ALMOG_LEXER_IMPLEMENTATION
```

Definition at line 2 of file temp.c.

4.9.1.3 ALMOG_STRING_MANIPULATION_IMPLEMENTATION

```
#define ALMOG_STRING_MANIPULATION_IMPLEMENTATION
```

Definition at line 1 of file temp.c.

4.9.2 Function Documentation

4.9.2.1 free_people()

```
void free_people (
            struct People people )
```

Definition at line 34 of file temp.c.

References People::elements, free_person(), and People::length.

Referenced by main().

4.9.2.2 free_person()

```
void free_person (
            struct Person p )
```

Definition at line 28 of file temp.c.

References Person::location, and Person::name.

Referenced by free_people(), and parse_people().

4.9.2.3 main()

```
int main (
            void  )
```

Definition at line 84 of file temp.c.

References ada_init_array, al_lex_entire_file(), al_tokens_free(), People::elements, free_people(), People::length, parse_people(), and print_person().

4.9.2.4 parse_people()

```
bool parse_people (
            struct Tokens * tokens,
            struct People * people )
```

Definition at line 66 of file temp.c.

References ada_append, ajp_array_begin(), ajp_array_end(), ajp_array_has_items(), free_person(), and parse_person().

Referenced by main().

4.9.2.5 parse_person()

```
bool parse_person (
            struct Tokens * tokens,
            struct Person * p )
```

Definition at line 42 of file temp.c.

References Person::age, ajp_object_begin(), ajp_object_end(), ajp_object_next_member(), ajp_parse_int(), ajp_parse_string(), ajp_unknown_key(), asm_strncmp(), Person::body_count, Tokens::current_key, Person::location, and Person::name.

Referenced by parse_people().

4.9.2.6 print_person()

```
void print_person (
            struct Person p )
```

Definition at line 19 of file temp.c.

References Person::age, asm_dprintINT, asm_dprintSTRING, asm_print_many_times(), Person::body_count, Person::location, and Person::name.

Referenced by main().

4.10 temp.c

```
00001 #define ALMOG_STRING_MANIPULATION_IMPLEMENTATION
00002 #define ALMOG_LEXER_IMPLEMENTATION
00003 #define ALMOG_JSON_PARSER_IMPLEMENTATION
00004 #include "Almog_JSON_Parser.h"
00005
00006 struct Person {
00007     const char *name;
00008     int age;
00009     const char *location;
00010     int body_count;
00011 };
00012
00013 struct People {
00014     size_t length;
00015     size_t capacity;
00016     struct Person* elements;
00017 };
00018
00019 void print_person(struct Person p)
00020 {
00021     asm_print_many_times("-", 20);
00022     asm_dprintSTRING(p.name);
00023     asm_dprintINT(p.age);
00024     asm_dprintSTRING(p.location);
00025     asm_dprintINT(p.body_count);
00026 }
00027
00028 void free_person(struct Person p)
00029 {
00030     free((char *)p.name);
00031     free((char *)p.location);
00032 }
00033
00034 void free_people(struct People people)
00035 {
00036     for (size_t i = 0; i < people.length; i++) {
00037         free_person(people.elements[i]);
00038     }
00039     free(people.elements);
00040 }
00041
00042 bool parse_person(struct Tokens *tokens, struct Person *p)
00043 {
00044     if (!ajp_object_begin(tokens)) return false;
00045
00046     for (;ajp_object_next_member(tokens);) {
00047         if (asm_strncmp(tokens->current_key, "name", 4) == true) {
00048             if (!ajp_parse_string(tokens, &(p->name))) return false;
00049         } else if (asm_strncmp(tokens->current_key, "age", 3) == true) {
00050             if (!ajp_parse_int(tokens, &(p->age))) return false;
00051         } else if (asm_strncmp(tokens->current_key, "location", 8) == true) {
00052             if (!ajp_parse_string(tokens, &(p->location))) return false;
00053         } else if (asm_strncmp(tokens->current_key, "body_count", 10) == true) {
00054             if (!ajp_parse_int(tokens, &(p->body_count))) return false;
00055         } else {
00056             ajp_unknown_key(tokens);
00057             return false;
00058         }
00059     }
00060
00061     if (!ajp_object_end(tokens)) return false;
00062
00063     return true;
```

```
00064 }
00065
00066 bool parse_people(struct Tokens *tokens, struct People *people)
00067 {
00068     if (!ajp_array_begin(tokens)) return false;
00069
00070     for (;ajp_array_has_items(tokens);) {
00071         struct Person p = {0};
00072         if (!parse_person(tokens, &p)) {
00073             free_person(p);
00074             return false;
00075         }
00076         ada_append(struct Person, *people, p);
00077     }
00078
00079     if (!ajp_array_end(tokens)) return false;
00080
00081     return true;
00082 }
00083
00084 int main(void)
00085 {
00086     struct Tokens tokens = al_lex_entire_file("profile.json");
00087
00088     struct People people = {0};
00089     ada_init_array(struct Person, people);
00090
00091     parse_people(&tokens, &people);
00092
00093     for (size_t i = 0; i < people.length; i++) {
00094         print_person(people.elements[i]);
00095     }
00096
00097
00098     al_tokens_free(tokens);
00099     free_people(people);
00100
00101     return 0;
00102 }
```

Index