APLWC
A Programming Language With Constraints

Peter Mikkelsen
petermikkelsen10@gmail.com

29th December 2025

Contents

Preface v

Changelog vii

I The Language 1

1 Syntax 3

2 Semantics 5

II The Implementation 7

3 Overall structure 9

3.1 Program Structure . . . 9

3.2 Copyright and license notice . . . 11

3.3 Command line options . . . 11

4 Memory management 17

5 The REPL 19

5.1 Read . . . 20

5.2 Parse . . . 21

5.3 Eval . . . 21

5.4 Print . . . 21

6 Index 23

6.1 Chunks . . . 23

6.2 Identifiers . . . 23

7 Full source code listing 25

III Tutorial 29

Preface

First of all, this document is work in progress. Don't expect anything from it.

This book describes the `APLWC` language, which is an experiment to create a programming language that allows the programmer to write declarative programs using constraints, in a consise and simple way. The programming language itself is an extension of APL, but it doesn't claim to be backwards compatible with other APL systems. I have chosen to write this document to finaly force myself to put some real thoughts into what this language and its implementation should look like. Part I describes the language itself, hopefully in enough detail that it serves as a language specification and as reference documentation. Part II describes the implementation of the interpreter that I wrote (there may be other interpreters in the future, who knows), and the description is a literate program using the noweb system. What that means is that every singe line of code is included in this book. Time will tell if doing it this way is a good or bad idea, but it certainly sounds fun, which is all that matters for personal projects like this one.

The name `APLWC` should be quite self-explanatory. I have not yet decided if the name is final, or how it should be stylised. `APLWC`, APLWC, APLwc, or aplwc are all fine, and a mix of those will probably be used throughout the book.

For readers unfamilar with the `APLWC` language, part III contains a tutorial to get you started.

Changelog

Version 0.0.1: Initial release

bla bla bla

Part I

The Language

Chapter 1

Syntax

Chapter 2

Semantics

Part II

The Implementation

Chapter 3

Overall structure

The implementation here is a C program which runs on linux, and possibly other posix systems. Since I am using the noweb literate programming system, the code will be explained in an order that makes sense to me, and that often means referring to code which is not yet written. It is recommended that the reader knows a little bit about literate programming, but it isn't required.

3.1 Program Structure

The entire source code will be extracted into a single C source file, with the following layout

⟨*aplwc.c* 1⟩≡

```
/* NOTE: THIS FILE IS AUTOGENERATED - DO NOT EDIT */
⟨Copyright notice 5⟩

⟨Includes 3⟩

/* Useful macro definitions */
⟨Macros 20⟩

/* Type declarations */
⟨Global type declarations 7⟩

/* Type definitions */
⟨Global type definitions 6⟩

/* Function prototypes */
⟨Function prototypes 21⟩

/* Global variables */
⟨Global variables 12⟩

/* Main entrypoint to the APLWC interpreter */
int
```

```
main(int argc, char *argv[])
{
        /* Parse command line options */
        ⟨Handle command line options 23⟩
        /* REPL */
        ⟨Read-eval-print loop 27⟩
        /* Done */
        ⟨Exit with success 2⟩
}
```

This definition is continued in chunks 14, 18, 26, and 29.
Root chunk (not used in this document).
Defines:
  main, never used.
Uses FILE 16.

Limiting the program to a single C source file has some downsides, but it also has the benefit that identifiers aren't duplicated (at least not top-level ones), and therefore the index in section 6.2 becomes easier to understand. Also, splitting a piece of software up into multiple pieces is in my experience often done to make the code more managable, but here we use different sections of the book to split the code into seperate logical pieces of work, which gives us the same benefit. The biggest issue is the lack of information hiding: nothing really prevents me from calling a utility function related to the symbol table code, from somewhere in the input/output code, even though the I/O code shouldn't care about the internals of the symbol table. Luckily, as the only developer of this piece of software, I hope I can manage not to do things like that.

The example above was the first look at what this book's literal program looks like. Every global definition has a name which is shown in angled brackets, as well as a number. In the PDF version of this book, the numbers are references and can be clicked to take the reader directly to the given definition. Notice how the first definition above has the name, followed by a ≡ symbol. Later we will see cases where a definition is extended, in which case the name will be followed by a + as well as ≡.

In an attempt to keep the code simple, and to make it easier to port to systems which lacks compilers with support for the latest and greatest C standards, the code targets C99.

Before we get started, let's try to define what it means to exit with success.

⟨*Exit with success* 2⟩≡

```
exit(EXIT_SUCCESS);
```

This code is used in chunks 1, 15, and 17.
Uses exit 3 and EXIT_SUCCESS 3.

We must also include the header which defines the `exit` function and the constant `EXIT_SUCCESS` used above.

⟨*Includes* 3⟩≡

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

This definition is continued in chunks 10, 16, 24, and 33.
This code is used in chunk 1.
Defines:
  exit, used in chunks 2 and 4.
  EXIT_FAILURE, used in chunk 4.
  EXIT_SUCCESS, used in chunk 2.

Similarly, we will later need a way to exit with failure

⟨*Exit with failure* 4⟩≡

```
exit(EXIT_FAILURE);
```

This code is used in chunk 23.
Uses `exit` 3 and `EXIT_FAILURE` 3.

3.2 Copyright and license notice

The easiest part to fill in is the copyright notice, so let's do that.

⟨*Copyright notice* 5⟩≡

```
/* Copyright 2025 Peter Mikkelsen
 *
 * This program is free software: you can redistribute it
 * and/or modify it under the terms of the GNU General
 * Public License as published by the Free Software
 * Foundation, either version 3 of the License, or (at
 * your option) any later version.
 *
 * This program is distributed in the hope that it will
 * be useful, but WITHOUT ANY WARRANTY; without even the
 * implied warranty of MERCHANTABILITY or FITNESS FOR A
 * PARTICULAR PURPOSE. See the GNU General Public License
 * for more details.
 *
 * You should have received a copy of the GNU General
 * Public License along with this program. If not, see
 * <https://www.gnu.org/licenses/>.
 */
```

This code is used in chunk 1.

3.3 Command line options

As the development progresses, there will be a need for some command line options to control various aspects of the interpreter. Until then, we will write some code which prints a helpful error message and exits when an unknown option is given. The idea is to store a global table of recognised option names, as well as function pointers to parse each option. The structure to describe a single option can be defined as

⟨*Global type definitions* 6⟩≡

```
struct CommandLineOption
{
        ⟨CommandLineOption members 8⟩
};
```

This code is used in chunk 1.
Defines:
`CommandLineOption`, used in chunks 7, 12, 19, and 23.

And we will give it a typedef name as well.

⟨*Global type declarations* 7⟩≡

```
typedef struct CommandLineOption CommandLineOption;
```

This definition is continued in chunk 32.
This code is used in chunk 1.
Uses `CommandLineOption` 6.

In general, all structs will have a seperate typedef, which comes before the structure definition itself, as that makes it possible for any structure to refer to any of the other structures by name, no matter the order in which they are defined.

The first member of the `CommandLineOption` structure is a character string which contains the command line option name itself, including any leading hyphens.

⟨*CommandLineOption members* 8⟩≡

```
char *name;
```

This definition is continued in chunks 9 and 11.
This code is used in chunk 6.

Then we also need a function pointer to parse the option. It will be a function which takes the command line argument string as its only argument, and returns a boolean indicating success.

⟨*CommandLineOption members* 8⟩+≡

```
bool (*parse)(char *);
```

This code is used in chunk 6.
Uses `bool` 10.

Since we are working with C99, we need to include the header which defines the `bool` type.

⟨*Includes* 3⟩+≡

```
#include <stdbool.h>
```

This code is used in chunk 1.
Defines:
`bool`, used in chunks 9, 15, 17, 21, and 28.
`false`, never used.
`true`, used in chunk 27.

Lastly, the structure also needs a character string with a descriptive message, for reasons that will become clear in a moment.

⟨*CommandLineOption members* 8⟩+≡

```
char *desc;
```

This code is used in chunk 6.

Now we can define a global table of recognised command line options.

⟨*Global variables* 12⟩≡

```
CommandLineOption command_line_options[] = {
        ⟨Command line options 13⟩
};
```

This definition is continued in chunks 22 and 28.
This code is used in chunk 1.
Defines:
`command_line_options`, used in chunks 19 and 25.
Uses `CommandLineOption` 6.

As a first example, let's add support for a `-v` option to print version information, as well as a `-h` option to display a help message. Common to both of these options is that the program should exit with a successful exit status immediately, and not process any other options, or enter the main REPL.

⟨*Command line options* 13⟩≡

```
{
        .name  = "-v",
        .desc  = "Show version information.",
        .parse = parse_option_v,
},
{
        .name  = "-h",
        .desc  = "Show this help message.",
        .parse = parse_option_h,
},
```

This code is used in chunk 12.
Uses `help` 19, `parse_option_h` 17, and `parse_option_v` 15.

We will need a bunch of utility functions to parse the options.

⟨*aplwc.c* 1⟩+≡

```
/* Functions to parse command line options */
⟨Command line option functions 15⟩
```

The version function is simple

⟨*Command line option functions* 15⟩≡

```
bool
parse_option_v(char *option)
{
        printf("APLWC version %s\n", aplwc_version);
        ⟨Exit with success 2⟩
}
```

This definition is continued in chunk 17.
This code is used in chunk 14.
Defines:
  `parse_option_v`, used in chunks 13 and 21.
Uses `aplwc_version` 22, `bool` 10, and `printf` 16.

It uses the `printf` function from `stdio.h`, so lets include that.

⟨*Includes* 3⟩+≡

```
#include <stdio.h>
```

This code is used in chunk 1.
Defines:
  `FILE`, used in chunks 1, 30, and 35.
  `printf`, used in chunks 15, 19, 23, and 35.

The help function is also simple, but it calls a utility function which walks through all the command line options in the `command_line_options` table, to generate a nice usage message.

⟨*Command line option functions* 15⟩+≡

```
bool
parse_option_h(char *option)
{
        help();
        ⟨Exit with success 2⟩
}
```

This code is used in chunk 14.
Defines:
`parse_option_h`, used in chunks 13 and 21.
Uses `bool` 10 and `help` 19.

Let's create a definition for all utility functions, such as `help`.

⟨*aplwc.c* 1⟩+≡

```
/* Utility functions */
⟨Utility functions 19⟩
```

⟨*Utility functions* 19⟩≡

```
void
help(void)
{
        size_t i = 0;
        CommandLineOption *o = command_line_options;

        printf("APLWC version %s\n", aplwc_version);
        printf("Supported command line options:\n");
        for(; i < nelem(command_line_options); i++, o++)
                printf("%s\t%s\n", o->name, o->desc);
}
```

This code is used in chunk 18.
Defines:
`help`, used in chunks 13, 17, 21, and 23.
Uses `aplwc_version` 22, `command_line_options` 12, `CommandLineOption` 6, `nelem` 20, and `printf` 16.

Here, `nelem` is a handy macro which computes the number of elements in an array

⟨*Macros* 20⟩≡

```
#define nelem(x) (sizeof(x)/sizeof((x)[0]))
```

This code is used in chunk 1.
Defines:
`nelem`, used in chunks 19 and 25.

Finally, the functions above must be prototyped as well.

⟨*Function prototypes* 21⟩≡

```
bool parse_option_v(char *);
bool parse_option_h(char *);
void help(void);
```

This definition is continued in chunk 30.
This code is used in chunk 1.
Uses `bool` 10, `help` 19, `parse_option_h` 17, and `parse_option_v` 15.

The functions also referred to a global variable aplwc_version, so let's define that.

⟨*Global variables* 12⟩+≡

```
char *aplwc_version = "0.0.1";
```

This code is used in chunk 1.
Defines:
  aplwc_version, used in chunks 15 and 19.

Now that we have a few options defined, we can write the code in the main function which handles command line options.

⟨*Handle command line options* 23⟩≡

```
for(int i = 1; i < argc; i++){
        CommandLineOption *o = NULL;
        char *arg = argv[i];
        ⟨Lookup command line option arg 25⟩
        if(o && o->parse(arg))
                continue;
        printf("Unrecognised option: %s\n", arg);
        help();
        ⟨Exit with failure 4⟩
}
```

This code is used in chunk 1.
Uses CommandLineOption 6, help 19, and printf 16.

Looking up a single command line option is easy using the strcmp function, which we must first include.

⟨*Includes* 3⟩+≡

```
#include <string.h>
```

This code is used in chunk 1.
Defines:
  strcmp, used in chunk 25.

⟨*Lookup command line option* arg 25⟩≡

```
for(size_t i = 0; i < nelem(command_line_options); i++){
        if(strcmp(arg, command_line_options[i].name)==0){
                o = &command_line_options[i];
                break;
        }
}
```

This code is used in chunk 23.
Uses command_line_options 12, nelem 20, and strcmp 24.

Chapter 4

Memory management

Since we are writing in C, sooner or later we must make a decision about how we manage dynamically allocated memory. One option is to be explicit, and manually call `malloc` and `free`, but that becomes tricky very fast. Also, I have found that it doesn't work well when explaining code in chunks like it is done here, as the `free` is sometimes very far away from the corresponding `malloc`. Therefore, we will implement a form of automatic memory management. Specifically, the type of memory manager we will implement, is a *reference counting* memory manager, where every piece of allocated memory (referred to as a *chunk* from now on) contains a reference count. When the count goes to zero, the memory will automatically be `free`'d, and any chunks that it referred to will have their reference counts decemented by one.

It is well known that reference counting cannot easily deal with memory cycles, since two chunks which refer to each other can keep each other's reference count above zero, even if "the rest of the program" cannot reach any of the two chunks. If we ever get to a situation where cycles are possible, we will have to deal with that, but until then we will just avoid cycles to make our lives easier.

We want our memory manager to be able to handle diffent kinds of chunks - some may be of fixed size, such as specific AST nodes, while others will be of dynamic size, such as UTF-8 strings and `APLWC` arrays. That means, that in addition to a reference count, each chunk should also record its own size somehow. Another problem is how the memory manager should know which othjer chunks a given chunk refers to. Ideally, the memory manager itself should have some generic mechanism for figuring this out, so that we can add support for more and more chunk types as the implementation progresses. If we give each chunk a unique type or tag, we could have a table of *chunk descriptiors*, which contains, among other things, a function pointer that somehow tells the memory manager about any chunks that a given chunk refers to.

Let's begin implementing this.

⟨*aplwc.c* 1⟩+≡

Chapter 5

The REPL

The interpreter is essenstially a big read-eval-print loop which asks the user for a line of code typically (the read phase), tokenises it and evaluates it (the eval phase), prints the result, if any (the print phase), and lastly it loops around to do it all again. Whether the tokenise/parsing of the line falls into the read or the eval phase isn't clear, so I have added an extra phase between the read and eval.

```
⟨Read-eval-print loop 27⟩≡
  running = true;
  while(running){
          /* Read a line of user input */
          ⟨REPL-read 31⟩
          /* Tokenise/parse it */
          ⟨REPL-parse 36⟩
          /* Evaluate it */
          ⟨REPL-eval 37⟩
          /* Print result */
          ⟨REPL-print 38⟩
          /* Cleanup and prepare for next round */
          ⟨REPL-cleanup 34⟩
  }
```

This code is used in chunk 1.
Uses `running` 28 and `true` 10.

the variable `running` is a global variable.

```
⟨Global variables 12⟩+≡
  bool running;
```

This code is used in chunk 1.
Defines:
  `running`, used in chunk 27.
Uses `bool` 10.

Now we can define the individual parts of the REPL separately.

5.1 Read

To read a line from the user, we can read successive characters from standard input, until we hit a newline character. But first, we must make it clear that we expect the input to be UTF-8, and therefore a single C `char` doesn't necessarily contain a complete unicode character. What we will need is a function to read a line of valid UTF-8 from the user. Let's define a block for all our unicode/UTF-8 related functions.

⟨*aplwc.c* 1⟩+≡

```
/* Unicode/UTF-8 related functions */
⟨Unicode functions 35⟩
```

The read part of the REPL is trivial if we imagine that we have a `readline` function which returns UTF-8.

⟨*Function prototypes* 21⟩+≡

```
utf8 readline(FILE *);
```

This code is used in chunk 1.
Uses `FILE` 16 and `utf8` 32.

⟨*REPL-read* 31⟩≡

```
utf8 input_line = readline(stdin);
```

This code is used in chunk 27.
Uses `utf8` 32.

We must also define the type `utf8`, which is a string of unsigned bytes.

⟨*Global type declarations* 7⟩+≡

```
typedef uint8_t *utf8;
```

This code is used in chunk 1.
Defines:
  `utf8`, used in chunks 30, 31, and 35.
Uses `uint8_t` 33.

The `uint8_t` type lives in a header which should also be included.

⟨*Includes* 3⟩+≡

```
#include <stdint.h>
```

This code is used in chunk 1.
Defines:
  `uint8_t`, used in chunk 32.

Let's now define the `readline`. The string is dynamically allocated using `malloc`, and it's result must be `free`'d when we are done with it.

⟨*REPL-cleanup* 34⟩≡

```
free(input_line);
```

This code is used in chunk 27.

⟨*Unicode functions* 35⟩≡

```
utf8
readline(FILE *f)
{
        size_t size = 0;
        size_t len = 0;
        utf8 buf = NULL;

        if(setvbuf(f, NULL, _IOLBF, 0) != 0)
                printf("Couln't setup line buffering\n");

        while(1){
                if(size == len){
                        size += 16;
                        buf = realloc(buf, size);
                        if(buf == NULL)
                                goto end;
                }
                if(fread(buf+len, size, 1, f) == 1)
                        len++;
                else
                        goto end;
        }
        printf("Read %d bytes\n", (int)len);
        buf = realloc(buf, len);
end:
        return buf;
}
```

This code is used in chunk 29.
Uses `FILE` 16, `printf` 16, and `utf8` 32.

5.2 Parse

⟨*REPL-parse* 36⟩≡

This code is used in chunk 27.

5.3 Eval

⟨*REPL-eval* 37⟩≡

This code is used in chunk 27.

5.4 Print

⟨*REPL-print* 38⟩≡

This code is used in chunk 27.

Chapter 6

Index

6.1 Chunks

⟨*aplwc.c* 1⟩ 1, 14, 18, 26, 29
⟨*Command line option functions* 15⟩ 14, 15, 17
⟨*Command line options* 13⟩ 12, 13
⟨*CommandLineOption members* 8⟩ 6, 8, 9, 11
⟨*Copyright notice* 5⟩ 1, 5
⟨*Exit with failure* 4⟩ 4, 23
⟨*Exit with success* 2⟩ 1, 2, 15, 17
⟨*Function prototypes* 21⟩ 1, 21, 30
⟨*Global type declarations* 7⟩ 1, 7, 32
⟨*Global type definitions* 6⟩ 1, 6
⟨*Global variables* 12⟩ 1, 12, 22, 28
⟨*Handle command line options* 23⟩ 1, 23
⟨*Includes* 3⟩ 1, 3, 10, 16, 24, 33
⟨*Lookup command line option* `arg` 25⟩ 23, 25
⟨*Macros* 20⟩ 1, 20
⟨*Read-eval-print loop* 27⟩ 1, 27
⟨*REPL-cleanup* 34⟩ 27, 34
⟨*REPL-eval* 37⟩ 27, 37
⟨*REPL-parse* 36⟩ 27, 36
⟨*REPL-print* 38⟩ 27, 38
⟨*REPL-read* 31⟩ 27, 31
⟨*Unicode functions* 35⟩ 29, 35
⟨*Utility functions* 19⟩ 18, 19

6.2 Identifiers

`aplwc_version`: 15, 19, 22
`bool`: 9, 10, 15, 17, 21, 28
`command_line_options`: 12, 19, 25

CommandLineOption: 6, 7, 12, 19, 23
exit: 2, 3, 4
EXIT_FAILURE: 3, 4
EXIT_SUCCESS: 2, 3
false: 10
FILE: 1, 16, 30, 35
help: 13, 17, 19, 21, 23
main: 1
nelem: 19, 20, 25
parse_option_h: 13, 17, 21
parse_option_v: 13, 15, 21
printf: 15, 16, 19, 23, 35
running: 27, 28
strcmp: 24, 25
true: 10, 27
uint8_t: 32, 33
utf8: 30, 31, 32, 35

Chapter 7

Full source code listing

```
/* NOTE: THIS FILE IS AUTOGENERATED - DO NOT EDIT */
/* Copyright 2025 Peter Mikkelsen
 *
 * This program is free software: you can redistribute it
 * and/or modify it under the terms of the GNU General
 * Public License as published by the Free Software
 * Foundation, either version 3 of the License, or (at
 * your option) any later version.
 *
 * This program is distributed in the hope that it will
 * be useful, but WITHOUT ANY WARRANTY; without even the
 * implied warranty of MERCHANTABILITY or FITNESS FOR A
 * PARTICULAR PURPOSE. See the GNU General Public License
 * for more details.
 *
 * You should have received a copy of the GNU General
 * Public License along with this program. If not, see
 * <https://www.gnu.org/licenses/>.
 */

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

/* Useful macro definitions */
#define nelem(x) (sizeof(x)/sizeof((x)[0]))

/* Type declarations */
typedef struct CommandLineOption CommandLineOption;
typedef uint8_t *utf8;

/* Type definitions */
struct CommandLineOption
{
  char *name;
  bool (*parse)(char *);
  char *desc;
```

```
};

/* Function prototypes */
bool parse_option_v(char *);
bool parse_option_h(char *);
void help(void);
utf8 readline(FILE *);

/* Global variables */
CommandLineOption command_line_options[] = {
  {
   .name = "-v",
   .desc = "Show␣version␣information.",
   .parse = parse_option_v,
   },
  {
   .name = "-h",
   .desc = "Show␣this␣help␣message.",
   .parse = parse_option_h,
   },
};

char *aplwc_version = "0.0.1";
bool running;

/* Main entrypoint to the APLWC interpreter */
int
main(int argc, char *argv[])
{
  /* Parse command line options */
  for(int i = 1; i < argc; i++) {
    CommandLineOption *o = NULL;
    char *arg = argv[i];

    for(size_t i = 0; i < nelem(command_line_options); i++) {
      if(strcmp(arg, command_line_options[i].name) == 0) {
        o = &command_line_options[i];
        break;
      }
    }
    if(o && o->parse(arg))
      continue;
    printf("Unrecognised␣option:␣%s\n", arg);
    help();
    exit(EXIT_FAILURE);
  }
  /* REPL */
  running = true;
  while(running) {
    /* Read a line of user input */
    utf8 input_line = readline(stdin);

    /* Tokenise/parse it */

    /* Evaluate it */

    /* Print result */

    /* Cleanup and prepare for next round */
    free(input_line);
  }
  /* Done */
  exit(EXIT_SUCCESS);
}

/* Functions to parse command line options */
bool
parse_option_v(char *option)
{
  printf("APLWC␣version␣%s\n", aplwc_version);
  exit(EXIT_SUCCESS);
}

bool
parse_option_h(char *option)
{
  help();
  exit(EXIT_SUCCESS);
}

/* Utility functions */
void
help(void)
{
  size_t i = 0;
  CommandLineOption *o = command_line_options;

  printf("APLWC␣version␣%s\n", aplwc_version);
  printf("Supported␣command␣line␣options:\n");
  for(; i < nelem(command_line_options); i++, o++)
    printf("%s\t%s\n", o->name, o->desc);
}

/* Unicode/UTF-8 related functions */
utf8
readline(FILE *f)
{
  size_t size = 0;
  size_t len = 0;
  utf8 buf = NULL;

  if(setvbuf(f, NULL, _IOLBF, 0) != 0)
    printf("Couln't␣setup␣line␣buffering\n");

  while(1) {
    if(size == len) {
      size += 16;
      buf = realloc(buf, size);
      if(buf == NULL)
        goto end;
    }
    if(fread(buf + len, size, 1, f) == 1)
      len++;
    else
      goto end;
  }
  printf("Read␣%d␣bytes\n", (int)len);
  buf = realloc(buf, len);
end:
  return buf;
}
```

Part III

Tutorial

To be written, but here are some random code from wikipedia to test the font support (APL387).

```
∇ AREA←DEGREES SEGMENTAREA RADIUS;FRACTION;CA;SIGN
  FRACTION←DEGREES÷360
  CA←CIRCLEAREA RADIUS
  SIGN←(×DEGREES)≠×RADIUS
  AREA←FRACTION×CA×~SIGN
∇
```