1. Literate Programming
Literate programming is a programming paradigm introduced by Donald Knuth in which a computer program is given an explanation of its logic in a natural language, such as English, see `https://en.wikipedia.org/wiki/Literate`

Knuth means, that a programmer at any time is able to understand, what you do and therefore it should be able to lef t ideas as comments except you get low. I refere to "Literate Programming", see `knuth_lit.pdf`: "Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do."

Und hier sind meine Gedanken: Und zudem soll zu jdem x-beliebigen Zeitpunkt der Autor, der erste Verantwortliche, mir erklren, was jedes Statement bedeutet oder ein Nacherfolger, der n-Verantwortliche, fr den die Vorgnger-Bedingung gilt.

2. C-Generator
The **ctangle** program converts the **CWEB** source document `hello.w` into the **C** program `hello.c` that may be compiled in the usual way. The output file includes `#line` specifications so that debugging can be done in terms of the **CWEB** source file, see `man ctangle`.

The command line should have two names on it. The first is **ctangle** and the second is taken as the **CWEB** input file `hello.w`.

```
ctangle hello.w
```

3. Compilation
Compiling *main* or a C program is a multi-stage process. At an overview level, the process can be split into four separate stages: preprocessing, compilation, assembly, and linking,
see `https://2150.medium.com/what-gcc-main-c-means-3d5e5e3e4d76`:

I'll walk through each of the four stages of compiling the C program **lit**:

4. Preprocessor
The first stage of compilation is called preprocessing. In this stage, lines starting with a # character are interpreted by the preprocessor as preprocessor commands. These commands form a simple macro language with its own syntax and semantics. This language is used to reduce repetition in source code by providing functionality to inline files, define macros, and to conditionally omit code.

Before interpreting commands, the preprocessor does some initial processing. his includes joining continued lines (lines ending with a and stripping comments.

To print the result of the preprocessing stage, pass the *-E* option to **gcc**:

```
gcc -E hello.c
```

5. Compilation
The second stage the preprocessed code is translated to assembly instructions specific to the target processor architecture. These form an intermediate human readable language.

The existence of this step allows for C code to contain inline assembly instructions and for different assemblers to be used.

Some compilers also supports the use of an integrated assembler, in which the compilation stage generates machine code directly, avoiding the overhead of generating the intermediate assembly instructions and invoking the assembler.

To save the result of the compilation stage, pass the *-S* option to **gcc**:

```
gcc S hello.c
```

6. Assembler
During this stage, an assembler is used to translate the assembly instructions to object code. The output consists of actual instructions to be run by the target processor.

To save the result of the assembly stage, pass the *-c* option to **gcc**:

```
gcc c hello.c
```

Running the above command will create a file named `hello.o`, containing the object code of the program. The contents of this file is in a binary format and an be inspected using hexdump or od by running either one of the following commands:

```
hexdump hello.o od c hello.o
```

7. Linking
The object code generated in the assembly stage is composed of machine instructions that the processor understands but some pieces of the program are out of order or missing. To produce an executable program, the existing pieces save to be rearranged and the missing ones filled in. This process is called linking. The linker will arrange the pieces of object code so that functions in some pieces can successfully call functions in other ones. It will also add pieces containing the instructions for library functions used by the program. In the ase of the **hello** program, the linker will add the object code for the *printf* function.

The result of this stage is the final executable program.

```
gcc o hello hello.c
```

8. User header
The next few sections contain stuff from the file `"lextern.w"` that must be included in both `"lit.w"` and `"log.w"`. It appears in file `"lextern.h"`, which is also included in `"lextern.w"` to propagate possible changes from this `EXTERN` interface consistently.

9. Substitution
Ritchie writes in "The Development of the C Language", see `c_development.html`: "... but the most important was the introduction of the preprocessor ... The preprocessor performs macro substitution, using conventions distinct from the rest of the language. ..."

Here's what Wittgenstein says in the **TLP**, see `tlp.pdf`: "6.24 The method by which mathematics arrives at its equations is the method of substitution. For equations express the substitutability of two expressions, and we proceed from a number of equations to new equations, replacing expressions by others in accordance with the equations."

Stallman defines *header file* in "The C Preprocessor", see `cpp.pdf`: A *header file* is a file containing C declarations and macro definitions (see Chapter 3 [Macros], page 13) to be shared between several source files. You request the use of a header file in your program by including it, with the C preprocessing directive **#include**.

10. System Header
The system header `<stdio.h>` declares three types, several macros, and many functions for performing input and output, see **ISO/IEC 9899:TC3**.

To use the *printf()* function we must include `<stdio.h>`. *printf* is the name of one of the main C output functions, and stands for "print formatted". The *printf* format string is a control parameter used by a class of functions in the input/output libraries of C and many other programming languages. The string is written in a simple template language: characters are usually copied literally into the function's output, but format specifiers, which start with a translate a piece of data (such as a number) to characters, see `https://en.wikipedia.org/wiki/Printf_format_string`.

The output function *printf* translates internal values to characters:

```
int printf(char *format, arg1, arg2, ...);
```

printf converts, formats, and prints its arguments on the format. It returns the number of characters printed, see **KR**.

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

11. External variables
Because external variables are globally accessible, they can be used instead of argument lists to communicate data between functions. Furthermore, because external variables remain in existence permanently, rather than appearing and disappearing as functions are called and exited, they retain their values even ater the functions that set them ve returned, see **KR**.

Declaration of the global variables or function simply declares that the variable or function exists, but the memory is not allocated for them.

argc: copy of *ac* parameter to *main*.
argv: copy of *av* parameter to *main*

```
extern int argc;
extern char **argv;
```

12. Program
`LIT` has a fairly straightforward outline. It operates in three phases: First it inputs the source file and stores cross-reference data, then it inputs the source once again and produces the TEX output file, finally it sorts and outputs the index.

13. Main function

A function definition has this form, see **KR**:Rechercheb Rechercheb

```
return-type function-name(parameter declarations, or void)
{
  declarations
  statements
}
```

The function *main* is called by the operation system. Each call passes two arguments to main, which each time returns an integer.

main() - The function *main* is called by the operation system. Each call passes two arguments to main, which each time returns an integer.

The way for running another program on `Linux` involves first calling *fork()*, which creates a new process as a copy of the first one, and then calling *exec()* to replace this copy (of the shell) with the actual program to run.

Richie and Kerninghan write: "... *main* is a special function. Our program begins executing at the beginning of *main*. This means that every program must have a *main* somewhere and will usually call other functions to help perform its job."

In the `C99` standard is defined: "The function called at program startup is named *main*. ... It shall be defined with a return type of **int** and ... or with two parameters (referred to here *ac* and *av*)."

ac: If the value of *ac* is greater than zero, the array members $av[0]$ through $av[ac-1]$ inclusive shall contain pointers to strings, which are given by the host environment prior to program startup.

av: If the value of *ac* is greater than zero, the string pointed to by $av[0]$ represents the program name. If the value of *ac* is greater than one, the strings pointed to by $av[1]$ through $av[ac-1]$ represent the program parameters.

return ... from the initial call to the *main* function is equivalent to calling the *exit* function with the value returned by the *main* function as its argument; reaching the end of *main* } that terminates the *main* function returns a value compatible with **int.**

1 xxx

x yyy

```
void main(int ac, char **av)
{
  argc = ac;      /* 1 */
  argv = av;      /* 2 */
  printf("Hi␣Herbert␣and␣Renate.\n");
}
```

14. Index.

LIT