Literally programming

In Hans Hagen's article 'Illiterate programming' (elsewhere in this Maps) he concludes by jokingly telling us that future documentation will be written using Keith McKay's style of literate programming.

Let me present to you a different method to achieve similar madness. What you see at the end of this article is the implementation of a text generating program. The implementation is in very traditional literate programming, although the output is a bit special.

Before we get into the program itself though, this article is a convenient way to explain some of the mechanics of using the traditional Knuthian form of literate programming.[1]

Literate programming for dummies

In literate programming, you create a single file (a so called 'WEB file') which contains both the application source code as well as all its documentation. But a WEB file as it stands is neither of these things. To create actual source code, you have to use a converter that 'tangles' the file into something your source compiler understands. And another converter must be used that 'weaves' the file into a format that TEX can interpret to produce typeset documentation along with nicely formatted source code.

The names of these converter programs depend on the target programming language used. There are some variants of the WEB literate programming system. I am using 'CWEB', which targets the C programming language. In this system, the 'weaving' program is called `cweave` and the 'tangling' program is called `ctangle`. In the traditional Knuth WEB system, the target programming language was Pascal and the programs were called `weave` and `tangle`. And there are more variants like NOWEB, which is programming language agnostic. There are some differences between all these literate programming systems, but overall they are all something like 95 to 99% compatible with each other.

The WEB file consists of (often many) small parts that are called 'modules'. Each of these start off with a documentation part. Next there is a part for shorthand definitions, and finally a section containing program code. The latter two parts are optional, so it is possible to write a module that contains nothing except documentation, or only documentation and definitions, or only documentation and program code.

Interspersed throughout the file there are special commands for the two converters. These commands all start with the character `@`.

Here is a short example, from the source of the typeset listing on the following pages:

```
@ First, here is the overview for the C program, and a few trivial helpers

Since the program below will make a lot of use of |rand()|, the |stldlib.h|
header is added immediately, and the randomizer is seeded with the current
time, which needs |time.h|.

@d true 1
@d false 0
```

1. However, I will be using C instead of Pascal as the actual programming language.

```
@c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
...
```

The module starts off with `@ ` . The command is an `@` sign followed by a space. The first few lines that follow are the documentation part. This documentation is essentially plain TEX. The generated documentation file includes a special macro package called `cwebmac.tex`[2] that runs on top of plain TEX. This package is needed to interpret some of the typesetting commands that are generated by the 'weaving' converter. To make documenting program code easier, the WEB system offers the very useful `|...|` verbatim shortcut, but otherwise the input is very similar to the plain TEX input one would write in a regular TEX file.

So `@ ` starts an unmarked module. Another way to start a module is with `@*`. This treats the following input (up to the first full stop) as a section title.

For the documentation part of a small program, that is really all you need to know in terms of special WEB commands. OK, maybe you need `@@` to write a literal at sign but that should be all. There are a few more commands, but I will skip over those as we are working on a '... for dummies' article section here. If you are working on a long and complicated program, you may want to use a few of the more advanced commands that I will not mention here.

All possible commands in CWEB are documented in its manual, see https://tug.ctan.org/info/knuth/cwebman.pdf if you want to know more than what is in this short and basic introduction.

Next up is the definitions part. Definitions start with the command `@d`. Whatever you define in one of the definitions you can use as an identifier in the source code that follows in the rest of the file. In the C targeting version of WEB, this is very much the same idea as `#define` in the regular source code, so you could use that instead. In fact, `ctangle` translates that bit of code into:

```
#define true 1
#define false 0
```

but the traditional `tangle` program (the 'tangling' converter for the Pascal programming language) would diligently change all occurrences of either 'true' or 'false' in the generated source code into either a one or a zero.

Definitions can have arguments as well, for example:

```
@d isempty(a) !strcmp(a, "")
```

like before, `ctangle` converts that into

```
#define isempty(a) !strcmp(a,"")
```

So why bother at all with definitions? The advantage of using `@d` in the definitions part over `#define` in the source part is that all definitions are output at the top of the generated source file. This means you can use the definition anywhere you want in the program code. An important feature, because we will see shortly that the 'tangle' converter can optionally totally reorder your source code.

After the definitions (or right after the documentation part, since the definition part is optional) comes the program code. In the most basic form, the source code part of the module starts with the `@c` command. We saw that in the earlier example. All such parts are output sequentially following all of the definitions from the WEB file. The module from the example above is converted to this:

2. For the programming language Pascal there is a similar but slightly different macro package, `webmac.tex`

```
/*2:*/
#line 23 "sylfac.w"

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

The `/*2:*/` means that this part of the generated source came from the second module in the WEB file, and that started on line 23. There was a module before that, but it contained nothing but documentation, so it was not output by the 'tangling' converter at all. The 'tangling' converter always removes all documentation parts.

Even though this is the actual start of the program code, the line containing `/*2:*/` does not appear until line 27 of the generated source. As mentioned earlier, all the definitions are output before that, at the very top of the generated file.

With the commands above, you could write complete programs. But the most powerful feature of the WEB system is that you can create special program code parts that can be reused elsewhere. The 'tangling' processor will reorganize such parts for you. The example earlier ended with a line containing three dots. The full module in the source code looks like this:

```
@ First, here is the overview for the C program, and a few trivial helpers

Since the program below will make a lot of use of |rand()|, the |stldlib.h|
header is added immediately, and the randomizer is seeded with the current
time, which needs |time.h|.

@d true 1
@d false 0

@c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

@<Extra headers@>
@<Global variables@>
@<Subroutines@>

int main (int argc, char **argv)
{
    srand(time(NULL));
    @<Produce Output@>;
    return 0;
}
```

The objects between the `@<...@>` are reusable code parts. In the rest of the program, there are modules that fill in these reusable parts. For example, `@<Extra headers@>` is defined by a few modules that appear later in the WEB file. Instead of starting with `@c`, source code parts can start with the command sequence `@<...@>=`.

The 'Extra headers' is first added to in this module:

```
@ The |strdup| function is in |string.h|, so let's include that header.

@<Extra headers@>=
#include <string.h>
```

And a few modules later another bit of code is added to it:

```
@ The header |ctype.h| is needed for |toupper()|.

@<Extra headers@>=
```

```
#include <ctype.h>
```

Repeated uses of the same title in this manner all add to the definition. In the generated C source it will come out like this:

```
/*2:*/
#line 23 "sylfac.w"

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/*10:*/
#line 176 "sylfac.w"

#include <string.h>

/*:10*//*18:*/
#line 267 "sylfac.w"

#include <ctype.h>
```

For the curious: `/*:10*/` indicates the end of module 10, and `/*10:*/` was its start. But more important are the `#line` directives, because those will be used by the C compiler when reporting errors. If those where not there, it would report errors based on the line number of the generated file. Thanks to the reordering feature, the line number in the generated C file has very little to do with the line number in the WEB file.

So far, I have spent a lot of attention on the generated C code. But what of the generated TEX code? Here is that same module, converted to TEX by the 'weaving' program:

```
\M{2}First, here is the overview for the C program, and a few trivial helpers

Since the program below will make a lot of use of \PB{\\{rand}(\,)}, the \PB{$%
\\{stldlib}.\|h$} header is added immediately, and the randomizer
is seeded with the current time, which needs \PB{$\\{time}.\|h$}.

\Y\B\4\D$\\{true}$ \5
\T{1}\par
\B\4\D$\\{false}$ \5
\T{0}\par
\Y\B\8\#\&{include} \.{<stdio.h>}\6
\8\#\&{include} \.{<stdlib.h>}\6
\8\#\&{include} \.{<time.h>}\6
\X10:Extra headers\X\6
\X6:Global variables\X\6
\X9:Subroutines\X\7
\1\1\&{int} \\{main}(\&{int} \\{argc}${},\39{}$\&{char} ${}{*}{*}\\{argv})\2%
\2{}$\6
${}\{{}$\1\6
${}\\{srand}(\\{time}(\NULL));{}$\6
\X20:Produce Output\X;\6
\&{return} \T{0};\6
\4${}\}{}$\2\par
\fi
```

Not the easiest thing to read. But it compiles just fine: all these extra one-letter commands are defined in `cwebmac.tex`; you can have fun look up their definitions if reading weirdly compact macro programming is your thing. As you can see, the `cweave` program knows quite a bit about the C programming language. Surprisingly perhaps, much more than the `ctangle` program. Pretty printing is much more challenging for a program that 'just' shuffling the source code blocks around.

That wraps up the quick intro to WEB programming. It should be obvious that neither the generated C code nor the generated TEX code are intended to be read

or altered. The whole idea is to do all your work in the WEB file, focussing on the program itself instead of being distracted by its source layout or documentation formatting. That is for sure the idea. But I must admit that I have always found it very tempting to spend days on end tinkering with the more obscure WEB language commands to make sure the TEX output looks 'exactly right'!

The syllable factory

Before inclusion of the typeset WEB pages of the example program, I should explain a little about it. Hans and Frans convinced me to write an article as a small introduction to literate programming. But to talk about literate programming, you need a literal program. I thought that the traditional 'Hello world' would be too short and boring, so I had to come up with something else.

Keith McKay's article on asemic writing provided the inspiration. What if, instead of asemic *writing*, I tried to do something with asemic *printing*?

The example program is called `sylfac`, short for 'Syllable factory'. It uses a few simple experimentally discovered rules to (slowly) output a never-ending list of 'sentences' using 'words' constructed of `syllables` that are loosely inspired by the Dutch language.

Here is an example of its output:

> Spijtufsaars geosves saasi se pukuimvigbaas schotaut et foo er aasdij oon. Le laan plirs sijt be huituu puk. Kip kokslek raukt zijkt luhot kik. Koe mu dakt tup zoe futike rut schaf wis eithoerstilaks? Saert fe ookbort paas hun spoos choklooch gi muirt. Mu uiskaafstas tuts bo slaukgu bil stoo ies laas. Slatuits ijt speetiep tu jiekt kliet? Ho klaurttiegtifga vasets oetit lat vui kle. Ka tat schui sletofrausme ersru art kikre stief, wu feitoklo aul oer du at eetwa. Da slu ta slon turt schaa, baak boebi spot. Eset diech ti pet kluit scheet. Lau scha ga ust sijks orskit ur ga. Wanfi schie ha fetli fij dokzo vienbisasstert ho sto dasotsran keis. Sa li ol vuil lust! Dootet zaan defuudeek ko kla tich agso uin so. Pie ut be ploo ge kossoch. Bi spi hoo dos dui ka bak. Schakstoojof at jijtaks kuutaustbokek ikt kas ak an su. Ut osde jaaks ho plistis stau lin ki fei vas rui, mi fok lei gehon atkla. Vui fiktkluistkeekluukt sui bustietburtkei teezokschu guu gi ist omkite be? Lijksmuistartguul seers taan too diemsle sas soos spuim och, scha huu.

I hope Keith can appreciate this, even though it does not look anywhere near as pretty as his drawings!

Taco Hoekwater

1. Sylfac – A Syllable Factory.

This is Sylfac, a program that fabricates fake words from fake syllables that are somewhat reminiscent of Dutch.

This program is part of an experiment whose goal it is to create a program that generates prose that is essentially asemic writing, but not random. Ideally, the created output will still bear enough similarities to a natural language to imply what the original inspiration was.

This initial attempt is likely to be a dead end, but it helps set up a baseline.

2. First, here is the overview for the C program, and a few trivial helpers

Since the program below will make a lot of use of *rand*( ), the *stldlib.h* header is added immediately, and the randomizer is seeded with the current time, which needs *time.h*.

```
#define true 1
#define false 0
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
  ⟨ Extra headers 10 ⟩
  ⟨ Global variables 6 ⟩
  ⟨ Subroutines 9 ⟩
  int main(int argc, char **argv)
  {
    srand(time(Λ));
    ⟨ Produce Output 20 ⟩;
    return 0;
  }
```

3. Syllable definitions.
In most languages, a syllable consists of three parts: zero or more leading consonants, a vowel, and zero or more trailing consonants.

This program uses a predefined distribution to decide what exactly counts as one of these three parts. Initially, this distribution was based on the inverse of the letter values in the Dutch version of the board game Scrabble[tm]: the maximum letter value is 10 (for Q), and the minimum value is 1 (for A, E, I, N, and O). The desired frequencies should be inverted: 10 times as many A's as Q's. This was easily done by using the letter values after subtracting them from 10. The resulting list was:

$$A_{10}B_8C_6D_8E_{10}F_7G_8H_7I_{10}J_7K_8L_8M_8N_{10}O_{10}P_8Q_1R_9S_9T_9U_7V_7W_6X_3Y_3Z_7$$

How to encode this inside a C program? It is very easy to create an array of strings where each letter is repeated as often as statistically requested. A simple randomize function can then be used to extract separate letters with a chance of being selected. To satisfy the 'zero' part in the 'zero or more' consonants, add empty strings to the array.

Of course, the vowels need to be separated from the consonants. And it is also best to separate leading and trailing consonant clusters.[1]

4. The distribution used above has used the *letter values* printed on the tiles. But there is also another possibility based on Scrabble[tm]: using the *number* of tiles provided for a specific letter. There are 100 non-blank tiles in Scrabble, and for Dutch they are distributed like this in the current version of Scrabble:

$$A^6B^2C^2D^6E^{18}F^2G^3H^2I^4J^2K^3L^3M^3N^{10}O^6P^2Q^1R^5S^5T^5U^3V^2W^2X^1Y^1Z^2$$

5. Both of those methods have downsides, though. The most obvious one is that (at least in Dutch) vowels can be longer than one letter, and consonants are more often than not clustered together.

Another downside (at least for the current version of this program) is that this definition method makes rather large arrays that are not fun to type in or read. The current program is intended at least as much a demonstration of CWEB as it is a functional program. Having large sections of data detracts from that.

In the end, I have opted for a system that has a 'compressed' list of occurrences, deleted a few special characters like Q and X, included a few multi-letter vowels, and also included a few combinations of consonants that happen a lot.[2]

I constructed the following actual data after browsing a file containing existing Dutch syllables. Because there is no statistical information for the appearance of these syllables in actual words, it is still fairly haphazard. I know for a fact that lots of Dutch words end with the syllable 'en', but of course that syllable was only mentioned in the file once.

Anyway, I find it good enough for now. Example output is something like: `Deel tei scha kij ortkat gaat hoe kim da toksri am.`

6. Vowel definitions.
The list of vowels is very dependent on the inspiration language.
The first define is the length of the array, the second is that maximum length of any of the included strings.

```
#define numvowels 32
#define maxvowellength 2
```

⟨ Global variables 6 ⟩ ≡

```
char *vowels[numvowels] = {"a","a","a","a","a","a","aa","au","e","e","e","e","ee","ei","e",
    "i","i","i","i","ie","ij","o","o","o","o","oe","oo","o","u","u","ui","uu"};
```

See also sections 7, 8, 12, and 14.

This code is used in section 2.

[1] Some experimentation made that clear.
[2] In Dutch, of course.

7. Leading consonant cluster definitions.

```
#define numleading 55
#define maxleadinglength 3
⟨ Global variables 6 ⟩ +≡
  char *leading[numleading] = {"","","","","","","","","","","","","","b","b","b","ch","d",
      "d","d","f","f","g","g","g","h","h","j","k","k","kl","k","l","l","m","n","p","pl",
      "r","r","s","s","s","sl","sp","sch","st","t","t","t","t","v","v","w","z"};
```

8. Trailing consonant cluster definitions.

```
#define numtrailing 49
#define maxtrailinglength 2
⟨ Global variables 6 ⟩ +≡
  char *trailing[numtrailing] = {"","","","","","","","","","","","","","","","","","","","",
      "f","g","ch","k","k","l","l","m","n","n","p","r","s","s","s","ks","s","rs","s","ts",
      "t","t","t","kt","t","t","rt","st"};
```

9. Here is the helper function that creates a syllable.

```
#define isempty(a) ¬strcmp(a,"")
#define maxsyllablelength (maxvowellength + maxleadinglength + maxtrailinglength)
⟨ Subroutines 9 ⟩ ≡
  char *make_syllable( )
  {
    char syllable_buffer[maxsyllablelength + 1];
    int max_len = sizeof syllable_buffer;
    char *first = leading[rand( ) % numleading];
    char *vowel = vowels[rand( ) % numvowels];
    char *second = trailing[rand( ) % numtrailing];
    while (isempty(first) ∧ isempty(second)) {
      first = leading[rand( ) % numleading];
    }
    int j = snprintf(syllable_buffer, maxsyllablelength, "%s%s%s", first, vowel, second);
    return strdup(syllable_buffer);
  }
```

See also section 13.

This code is used in section 2.

10. The *strdup* function is in *string.h*, so let's include that header.

```
⟨ Extra headers 10 ⟩ ≡
#include <string.h>
```

See also sections 18 and 22.

This code is used in section 2.

11. Creating output.

12. The next subroutine generates a complete word. A word can have more than one syllable (up to four) at the moment. To encode the probabilities, a similar approach is used as for the letters.

```
#define syllableranges 26
#define maxsyllables 4
⟨ Global variables 6 ⟩ +≡
  int syllable_count[syllableranges] = {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,2,2,2,2,3,4};
```

13. Syllables are simply concatenated. A possible future extension would do something special if the previous syllable ends with a vowel and the new one starts with a vowel. In Dutch, sometimes an 'trema' is added to the first letter of the second syllable, like for example in *zeeën*.

```
⟨ Subroutines 9 ⟩ +≡
  char *make_word( )
  {
    char word_buffer[maxsyllablelength * maxsyllables + 1] = {0};
    int max_len = sizeof word_buffer;
    int sylcount = syllable_count[rand( ) % syllableranges];
    while (sylcount-- > 0) {
      char *syllable = make_syllable( );
      size_t existing = strlen(word_buffer);
      int j = snprintf(word_buffer + existing, max_len - existing, "%s", syllable);
      free(syllable);
    }
    return strdup(word_buffer);
  }
```

14. In the output, we want to use capitalized sentences that end with punctuation. The next definition should be familiar by now

```
#define allpunctuations 16
⟨ Global variables 6 ⟩ +≡
  char *punct[allpunctuations] = {".",".",".",".",".",".",".",".",".",".",".",".",".",".",",",
```

16. Adding punctuation.

Besides finding and adding a punctuation character, there is some special code to check for comma. Commas are a bit special because after that a subsentence starts. Such post-comma subsentences are often a bit shorter than the part before the comma, so the variable *sentencelength* is updated with a different value. If the punctuation was *not* a comma, the next word needs to be uppercased.

```
#define subsentencelength (rand() % 8) + 1
⟨ Add punctuation and update sentence length 16 ⟩ ≡
  int puncindex = rand() % allpunctuations;
  fputs(punct[puncindex], stdout);
  if (strcmp(punct[puncindex], ",")) {
    newsentence = true;
    sentencelength = normalsentencelength;
  }
  else {
    sentencelength = subsentencelength;
  }
  wordcounter = 0;
  charsonline++;
```

This code is used in section 20.

17. Output a word.

If this is the first word in a sentence, uppercase it.

```
⟨ Output a word 17 ⟩ ≡
  char *word = make_word();
  wordcounter++;
  if (newsentence ≡ true) {
    newsentence = false;
    *word = toupper(*word);
  }
  fputs(word, stdout);
  charsonline += strlen(word);
  free(word);
```

This code is used in section 20.

18. The header *ctype.h* is needed for *toupper*().

```
⟨ Extra headers 10 ⟩ +≡
#include <ctype.h>
```

19.

```
⟨ Add a space 19 ⟩ ≡
  putchar(' ');
  charsonline++;
```

This code is used in section 20.

20. And here it all comes together
Not much left to do here. The loop runs forever, press the interrupt key on the terminal to stop the app.

```
⟨ Produce Output 20 ⟩ ≡
  ⟨ Local variables for output 15 ⟩;
  while (1) {
    ⟨ Output a word 17 ⟩;
    if (wordcounter > sentencelength) {
      ⟨ Add punctuation and update sentence length 16 ⟩;
    }
    ⟨ Add a space 19 ⟩;
    if (charsonline > 100) {
      puts("");
      charsonline = 0;
    }
    ⟨ Pretend to be chatgpt 21 ⟩;
  }
```

This code is used in section 2.

21.

```
⟨ Pretend to be chatgpt 21 ⟩ ≡
  fflush(stdout);
  usleep((rand() % 1000) * 300);
```

This code is used in section 20.

22. Faking AI is hard work

```
⟨ Extra headers 10 ⟩ +≡
#include <unistd.h>
```

23. Index.

allpunctuations: 14, 16.
argc: 2.
argv: 2.
charsonline: 15, 16, 17, 19, 20.
ctype: 18.
existing: 13.
false: 2, 17.
fflush: 21.
first: 9.
fputs: 16, 17.
free: 13, 17.
isempty: 9.
j: 9, 13.
leading: 7, 9.
main: 2.
make_syllable: 9, 13.
make_word: 13, 17.
max_len: 9, 13.
maxleadinglength: 7, 9.
maxsyllablelength: 9, 13.
maxsyllables: 12, 13.
maxtrailinglength: 8, 9.
maxvowellength: 6, 9.
newsentence: 15, 16, 17.
normalsentencelength: 15, 16.
numleading: 7, 9.
numtrailing: 8, 9.
numvowels: 6, 9.
puncindex: 16.
punct: 14, 16.
putchar: 19.
puts: 20.
rand: 2, 9, 13, 15, 16, 21.
second: 9.
sentencelength: 15, 16, 20.
snprintf: 9, 13.
srand: 2.
stdout: 16, 17, 21.
stldlib: 2.
strcmp: 9, 16.
strdup: 9, 10, 13.
string: 10.
strlen: 13, 17.
subsentencelength: 16.
sylcount: 13.
syllable: 13.
syllable_buffer: 9.
syllable_count: 12, 13.
syllableranges: 12, 13.
time: 2.
toupper: 17, 18.
trailing: 8, 9.
true: 2, 15, 16, 17.
usleep: 21.
vowel: 9.
vowels: 6, 9.
word: 17.
word_buffer: 13.
wordcounter: 15, 16, 17, 20.

⟨ Add a space 19 ⟩ Used in section 20.
⟨ Add punctuation and update sentence length 16 ⟩ Used in section 20.
⟨ Extra headers 10, 18, 22 ⟩ Used in section 2.
⟨ Global variables 6, 7, 8, 12, 14 ⟩ Used in section 2.
⟨ Local variables for output 15 ⟩ Used in section 20.
⟨ Output a word 17 ⟩ Used in section 20.
⟨ Pretend to be chatgpt 21 ⟩ Used in section 20.
⟨ Produce Output 20 ⟩ Used in section 2.
⟨ Subroutines 9, 13 ⟩ Used in section 2.

SYLFAC

	Section	Page
Sylfac – A Syllable Factory	1	1
Syllable definitions	3	2
Creating output	11	4
Index	23	7