WEB to cweb

WEB to cweb

Converting TEX from WEB to cweb

Für meinen Vater

MARTIN RUCKERT *Munich University of Applied Sciences*

Date: 2018-04-03 15:43:51 +0200 (Tue, 03 Apr 2018)

Revision: 1192

Ruckert, Martin.
WEB to cweb
Includes index.
ISBN 1-548-58234-4

Internet page https://w3-o.cs.hm.edu/~ruckert/web2w/ may contain current information about this book, downloadable software, and news.

ruckert@cs.hm.edu

ISBN-10: 1-548-58234-4
ISBN-13: 987-1548582340

First printing, August 2017

Preface

This book describes a project to convert the TEX source code[5] written by Donald E. Knuth as a "WEB"[6] into a "cweb"[2].

- On December 9, 2016, I started to implement `web2w` as a compiler for `WEB` files which is described below. The compiler, as compilers usually do, reads an input file and continues to produce a parse tree. The resulting parse tree has two structures: a linear structure representing the linear order of the input file and a tree structure representing the embedded Pascal program. Then the embedded Pascal program needs to be translated into an equivalent C program. And finally, the linear structure of the parse tree will be used to output a `cweb` file. Small corrections on the resulting `cweb` file are implemented by a patch file.

  The overall goal is the generation of a `ctex.w` file that is as close as possible to the `tex.web` input file, and can be used to produce `ctex.tex` and `ctex.c` using the standard tools `ctangle` and `cweave`.

  The TEX program can then be compiled from `ctex.c` and the TEX documentation can be generated from `ctex.tex` by TEX itself.

  This will simplify the tool chain necessary to generate TEX from its "sources".
- On April 20, 2017, I was able to create the first "hello world" `dvi` file with my newly generated TEX program and with that, I had reached version 0.1 of `web2w`.
- On April 26, 2017, I succeeded for the first time to generate a program that would pass the trip test and therefore can be called TEX. This was then version 0.2 of `web2w`.

  While the program at this point was a "correct implementation of TEX", its form still needed further improvement. For example, the sizes of arrays were computed and occurred in the source as literal numbers. It would be appropriate for source code that instead the expression defining the array size were used to specify the array size. The use of **return** statements and the elimination of unused *end* labels also asked for improvement.
- On May 11, 2017, I completed version 0.3 of `web2w`. Numerous improvements were added by then: some concerning the presentation of `web2w` itself, others with the goal of generating better `cweb` code for TEX. I decided then to freeze the improvement of the code for a while and prepare this document for publication as a book.
- On July 27, 2017, I completed version 0.4 of `web2w`, the first version that will be published as a book. More improvements (and more versions) are still to come.

Of course, changes in the code part of TEX will necessarily require changes in the documentation part. These can, however, not result from an automatic compilation. So the plan is to develop patch files that generate from the latest $0.x$ versions improved $1.y$ versions. These versions will share the same goal as version $0.x$: producing a `cweb` TEX source file that is as close as possible to the original web source but with a documentation part of each section that reflects the changes made in the code.

- There is a long term goal that brought me to construct `web2w` in the first place: I plan to derive from the TEX sources a new kind of TEX that is influenced by the means and necessities of current software and hardware. The name for this new implementation will be HINT which is, in the usual Open Software naming schema, the acronym for "HINT is not TEX".

  For example, HINT will accept UTF-8 input files because this is the defacto standard due to its use on the world wide web. Further, the machine model will be a processor that can efficiently handle 64-Bit values and has access to large amounts of main memory (several GByte). Last not least, I assume the availability of a good modern C compiler and will leave optimizations to the compiler if possible.

  The major change however will be the separation of the TEX frontend: the processing of `.tex` files, from the TEX backend: the rendering of paragraphs and pages.

  Let's look, for example, at ebooks: Current ebooks are of minor typographic quality. Just compiling TEX sources to a standard ebook format, for example epub, does not work because a lot of information that is used by TEX to produce good looking pages is not available in these formats. So I need to cut TEX (or HINT) in two pieces: a frontend, that reads TEX input and a backend that renders pixel on a page. The frontend will not know about the final page size because the size of the output medium may change while we read—for example by turning a mobile device from landscape to portrait mode. On the other hand, the computational resources of the backend are usually limited because a mobile device has a limited supply of electrical energy. So we should do as much as we can in the frontend and postpone what needs to be postponed to the backend. In between front and back, we need a nice new file format, that is compact and efficient, and transports whatever information is necessary between both parts.

  These are the possible next steps:

  - As a first step, I will make a version of TEX that produces a file listing all the contributions and insertions that TEX sends to the page builder. Let's call this a `.hint` file. This version of TEX will become the final frontend.
  - Next, I will use a second version of TEX where I replace the reading of `.tex` files by the reading of a `.hint` file and feeding its content directly to the page builder. This version of TEX will become the final backend. Once done, I can test the equation TEX = HINT frontend + HINT backend.
  - Next, I will replace the generation of dvi files in the backend by directly displaying the results in a "viewer". The "viewer" reads in a `.hint` file and

uses it to display one single page at an arbitrary position. Using page up and page down buttons, the viewer can be used to navigate in the `.hint` file. At that point, it should be possible to change `vsize` dynamically in the viewer.

- The hardest part will be the removal of `hsize` dependencies from the frontend and moving them to the backend. I am still not sure how this will work out.
- Once the author of a TEX document can no longer specify the final `hsize` and `vsize`, he or she would probably wish to be able to write conditional text for different ranges of `hsize` and `vsize`. So if the frontend encounters such tests it needs to include all variants in its output file.
- Last not least, most people use LATEX not plain TEX. Hence, if I want many people to use HINT, it should be able to work with LATEX. As a first step, I looked at ϵ-TEX, and my cweb version of ϵ-TEX already passes the extended trip test for ϵ-TEX. But I am not sure what LATEX needs beside the extensions of ϵ-TEX. So if someone knows, please let me know.

Enough now of these fussy ideas about the future. Let's turn to the present and the conversion of TEX from `WEB` to `cweb`.

San Luis Obispo, CA
June 27, 2017 *Martin Ruckert*

Contents

List of Figures and Tables

Figures

Tables

1 Introduction

`web2w`, the program that follows, was not written following an established software engineering workflow as we teach it in our software engineering classes. Instead the development of this program was driven by an ongoing exploration of the problem at hand where the daily dose of success or failure would determine the direction I would go on the next day.

This description of my program development approach sounds a bit like "rapid prototyping". But "prototype" implies the future existence of a "final" version and I do not intend to produce such a "final" version. Actually I have no intention to finish the prototype either, and I might change it in the future in unpredictable ways. I expect, however, that the speed of its further development will certainly decrease as I move on to other problems. Instead I have documented the development process as a literate program: the pages you are just reading. So in terms of literature, this is not an epic novel with a carefully designed plot, but it's more like the diary of an explorer who sets out to travel trough yet uncharted territories.

The territory ahead of me was the program TEX written by Donald E. Knuth using the `WEB` language as a literate program. As such, it contains snippets of code in the programming language Pascal—Pascal-H to be precise. Pascal-H is Charles Hedrick's modification of a compiler for the DECsystem-10 that was originally developed at the University of Hamburg (cf. [3] see [5]). So I could not expect to find a pure "Standard Pascal". But then, the implementation of TEX deliberately does not use the full set of features that the language Pascal has to offer in order to make it easier to run TEX on a large variety of machines. At the beginning, it was unclear to me what problems I would encounter with the subset of Pascal that is actually used in TEX.

Further, the problem was not the translation of Pascal to C. A program that does this is available in the TEX Live project: `web2c`[1] translates the Pascal code that is produced using `tangle` from `tex.web` into C code. The C code that is generated this way can, however, not be regarded as human readable source code. The following example might illustrate this: Figure 1 shows the `WEB` code for the function *new_null_box*. The result of translating it to C by `web2c` can be seen in figure 3. In contrast, figure 2 shows what `web2w` will achieve.

It can be seen, that `web2c` has desugared the sweet code written by Knuth to make it unpalatable to human beings, the only use you can make of it is feeding it to a C compiler. In contrast, `web2w` tries to create source code that is as close to the original as possible but still translates Pascal to C. For example, note the last statement in the *new_null_box* function: where C has a **return** statement, Pascal

136. The *new_null_box* function returns a pointer to an *hlist_node* in which all subfields have the values corresponding to '`\hbox{}`'. The *subtype* field is set to *min_quarterword*, since that's the desired *span_count* value if this *hlist_node* is changed to an *unset_node*.

```
function new_null_box: pointer;
          { creates a new box node }
  var p: pointer;   { the new node }
  begin p ← get_node(box_node_size);
  type(p) ← hlist_node;
  subtype(p) ← min_quarterword;
  width(p) ← 0; depth(p) ← 0;
  height(p) ← 0; shift_amount(p) ← 0;
  list_ptr(p) ← null;
  glue_sign(p) ← normal;
  glue_order(p) ← normal;
  set_glue_ratio_zero(glue_set(p));
  new_null_box ← p;
  end;
```

Fig. 1: WEB *code for new_null_box*

136. The *new_null_box* function returns a pointer to an *hlist_node* in which all subfields have the values corresponding to '`\hbox{}`'. The *subtype* field is set to *min_quarterword*, since that's the desired *span_count* value if this *hlist_node* is changed to an *unset_node*.

```
pointer new_null_box(void)
  /* creates a new box node */
{ pointer p;      /* the new node */
  p = get_node(box_node_size);
  type(p) = hlist_node;
  subtype(p) = min_quarterword;
  width(p) = 0; depth(p) = 0;
  height(p) = 0; shift_amount(p) = 0;
  list_ptr(p) = null;
  glue_sign(p) = normal;
  glue_order(p) = normal;
  set_glue_ratio_zero(glue_set(p));
  return p;
}
```

Fig. 2: cweb *code for new_null_box*

```
halfword
newnullbox ( void )
{
  register halfword Result; newnullbox_regmem
  halfword p  ;
  p = getnode ( 7 ) ;
  mem [p ].hh.b0 = 0 ;
  mem [p ].hh.b1 = 0 ;
  mem [p + 1 ].cint = 0 ;
  mem [p + 2 ].cint = 0 ;
  mem [p + 3 ].cint = 0 ;
  mem [p + 4 ].cint = 0 ;
  mem [p + 5 ].hh .v.RH = -268435455L ;
  mem [p + 5 ].hh.b0 = 0 ;
  mem [p + 5 ].hh.b1 = 0 ;
  mem [p + 6 ].gr = 0.0 ;
  Result = p ;
  return Result ;
}
```

Fig. 3: The C *code for new_null_box as generated by* web2c

assigns the return value to the function name. A simple translation, sufficient for a C compiler, can just replace the function name by "`Result`" (an identifier that is not used in the implementation of TEX) and add "`return Result;`" at the end of the function (see figure 3). A translation that strives to produce nice code should, however, avoid such ugly code.

Let's look at another example:

```
function new_character(f : internal_font_number; c : eight_bits): pointer;
  label exit;
  var p: pointer;   { newly allocated node }
  begin if font_bc[f] ≤ c then
    if font_ec[f] ≥ c then
      if char_exists(char_info(f)(qi(c))) then
        begin p ← get_avail; font(p) ← f; character(p) ← qi(c);
        new_character ← p; return;
        end;
  char_warning(f, c); new_character ← null;
exit: end;
```

Fig. 4: The `WEB` *code for new_character*

```
pointer new_character(internal_font_number f, eight_bits c)
{ pointer p;      /* newly allocated node */
  if (font_bc[f] ≤ c)
    if (font_ec[f] ≥ c)
      if (char_exists(char_info(f)(qi(c)))) { p = get_avail( ); font(p) = f;
        character(p) = qi(c); return p;
      }
  char_warning(f, c); return null;
}
```

Fig. 5: The `cweb` *code for new_character*

In figure 4 there is a "**return**" in the innermost **if**. This "**return**" is actually a macro defined as "**goto** *exit*", and "*exit*" is a numeric macro defined as "10". "**return**" is a reserved word in C and "*exit*" is a function of the C standard library, so something has to be done. The example also illustrates the point that I can not always replace an assignment to the function name by a C return statement. Only if the assignment is in a tail position, that is a position where the control-flow leads directly to the end of the function body, it can be turned into a return statement as happened in figure 5. Further, if all the goto statements that lead to a given label have been eliminated, as it is the case here, the label can be eliminated as well. In figure 5 there is no "*exit*:" preceding the final "`}`".

Another seemingly small problem is the different use of semicolons in C and Pascal. While in C a semicolon follows an expression to make it into a statement, in Pascal the semicolon connects two statements into a statement sequence. For

example, if an assignment precedes an "**else**", in Pascal you have "`x:=0 else`" where as in C you have "`x=0; else`"; no additional semicolon is needed if a compound statement precedes the "**else**". When converting `tex.web`, a total of 1119 semicolons need to be inserted at the right places. Speaking of the right place: Consider the following `WEB` code:

if $s \geq str\_ptr$ **then** $s \leftarrow$ `"???"` { this can't happen }
else if $s < 256$ **then**

Where should the semicolon go? Directly preceding the "**else**"? Probably not! Alternatively, I can start the search for the right place to insert the semicolon with the assignment. But this does not work either: the assignment can be spread over several macros or modules which can be used multiple times; so the right place to insert a semicolon in one instance can be the wrong place in another instance. `web2w` places the semicolon correctly behind the assignment like this:

if $(s \geq str\_ptr)$ $s = \langle$ `"???"` 1381 $\rangle$; /* this can't happen */
else if $(s < 256)$

But look what happened to the string `"???"`. Strings enclosed in C-like double quotes receive a special treatment by `tangle`: the strings are collected in a string pool file and replaced by string numbers in the Pascal source. No such mechanism is available in `ctangle`. My first attempt was to replace the string handling of TEX and keep the C-like strings in the source code. TEXs string pool is, however, hardwired into the program and is used not only for static strings but also for strings created at runtime, for example the names of control sequences. So I tried a hybrid approach: keeping strings that are used only for output (error messages for example) and translating other strings to string numbers. There are different places where the translation of a string like `"Maybe␣you␣should␣try␣asking␣a␣human?"` to a number like 283 can take place.

1. One could add a function s to do the translation at runtime and then write s(`"Maybe␣you␣should␣try␣asking␣a␣human?"`). The advantage is simplicity and readability; the disadvantage is the overhead in time and space (the string will exist twice: as a static string and as a copy in the string pool).
2. One could use the C preprocessor to do the job. For example, I could generate a macro *Maybe_you_should_try_asking_a_human0x63* that is defined as 283 and a second macro *str_283* for the string itself. Then, I can replace the occurrence of the string in the source by the macro name that mimics the string content and initialize the *str_pool* and *str_start* array using the other macro.
3. As a third variation used below, one can use the module expansion mechanism of `ctangle`. I generate for each string a module, in the above example named $\langle$ `"Maybe␣you␣should␣try␣asking␣a␣human?"` 1234 $\rangle$, that will expand to the correct number, here 283. And as in the previous method use a macro *str_283* to initialize *str_pool* and *str_start*. The advantage is the greater flexibility and the nicer looking replacements for strings, because module names can use the full character set. (Imagine replacing `"???"` by *_0x630x630x63*.)

In retrospect, after seeing how nice method 3 works, I ponder if I should have decided to avoid the hybrid approach and use approach 3 for all strings. It would

have reduced the amount of changes to the source file considerably. I further think that approach 1 has its merits too. The overhead in space is just a few thousand byte and the overhead in time is incurred only when the strings are actually needed which is mostly during a run of `initex` and while generating output (which is slow anyway).

A mayor difference between Pascal and C is the use of subrange types. Subrange types are used to specify the range of valid indices when defining arrays. While most arrays used in TEX start with index zero, not all do. In the first case, they can be implemented as C arrays which always start at index zero; in the latter case, I define a zero based array having the right size, adding a "0" to the name. Then, I define a constant pointer initialized by the address of the zero based array plus/minus a suitable offset so that I can use this pointer as a replacement for the Pascal array.

When subrange types are used to define variables, I replace subrange types by the next largest C standard integer type as defined in `stdint.h` which works most of the time. Consider the code

```
var p: 0 .. nest_size;   { index into nest }
  ⋮
for p ← nest_ptr downto 0 do
```

where $nest\_size = 40$. Translating this to

```
uint8_t p;      /* index into nest */
  ⋮
for (p = nest_ptr; p ≥ 0; p--)
```

would result in an infinite loop because p would never become less than zero; instead it would wrap around. So in this (and 21 similar cases), I declare the variables used in for-loops to be of type **int**.

I will not go into further details of the translation process as you will find all the information in what follows below. Instead, I will take a step back now and give you the big picture, looking back at the journey that took me to this point.

The program `web2w` works in three phases: First I run the input file `tex.web` through a scanner producing tokens (see section 9). The pattern matching is done using `flex`, the action code consists of macros described here. The tokens form a doubly linked list, so that later I can traverse the source file forward and backward. During scanning, information is gathered and stored about macros, identifiers, and modules. In addition, every token has a *link* field which is used to connect related tokens. For example, I link an opening parenthesis to the matching closing parenthesis, and the start of a comment to the end of the comment.

After scanning comes parsing. The parser is generated using `bison` from a modified Pascal grammar (see section 10). To run the parser, I need to feed it with tokens, rearranged in the order that `tangle` would produce, expanding macros and modules as I go. While parsing, I gather information about the Pascal code. At the beginning, I tended to use this information immediately to rearrange the token sequence just parsed. Later, I learned the hard way (modules that were modified on

the first encounter would later be feed to the parser in the modified form) that it is better to leave the token sequence untouched and just annotate it with information needed to transform it during the next stage. A technique that proved to be very useful is connecting the key tokens of a Pascal structure using the *link* field. For example, connecting a "**case**" token with its "**do**" token makes it easy to place the expression that is between these tokens, without knowing anything about its actual structure, between " **switch** (" and ")". The final stage is the generation of `cweb` output. Here the token sequence is traversed a third time, this time again in input file order. This time, the traversal will stop at the warning signs put up during the first two passes, use the information gathered so far, and rewrite the token sequence as gentle and respectful as possible from Pascal to C.

Et voilà! `tex.w` is ready—almost at least. I apply a last patch file, for instance to adapt documentation reliant on `webmac.tex` so that it works with `cwebmac.tex`, or I make small changes that do not deserve a more general treatment. The final file is then called `ctex.w` from which I obtain `ctex.c` and `ctex.tex` simply by applying `ctangle` and `cweave`. Using "`gcc ctex.c -o ctex`" I get a running `ctex`. Running "`ctex ctex.tex`" to get `ctex.dvi` is then just a tiny step away: it is necessary to set up format and font metric files. The details on how to do that and run (and pass) the infamous trip test are described in section 11.

2 Converting WEB to cweb

web2w is implemented by a C code file:

```
#include <stdlib.h>                                                    (1)
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdbool.h>
#include <stdint.h>
#include <math.h>
#include "web2w.h"
#include "pascal.tab.h"
  ⟨ internal declarations 3 ⟩
  ⟨ global variables 11 ⟩
  ⟨ functions 13 ⟩
  int main(int argc, char *argv[])
  {
    ⟨ process the command line 213 ⟩
    ⟨ read the WEB 4 ⟩
    ⟨ parse Pascal 92 ⟩
    ⟨ generate cweb output 100 ⟩
    ⟨ show summary 12 ⟩
    return 0;
  }
```

I also create the header file web2w.h included in the above C file. It contains the external declarations and is used to share constants, macros, types, variables, and functions with other C files.

```
⟨ web2w.h 2 ⟩ ≡                                                        (2)
  ⟨ external declarations 5 ⟩
```

3 Reading the WEB

When I read the WEB, I split it into a list of tokens; this process is called "scanning". I use flex (the free counterpart of lex) to generate the function *wwlex* from the file web.l.

⟨ internal declarations 3 ⟩ ≡ (3)
```
extern int wwlex(void);                          /* the scanner */
extern FILE *wwin;                               /* the scanners input file */
extern FILE *wwout;                              /* the scanners needs an output file */
```
Used in 1.

Using this function, I can read the WEB and produce a token list.

⟨ read the WEB 4 ⟩ ≡ (4)
```
⟨ initialize token list 22 ⟩
wwlex( ); ⟨ finalize token list 66 ⟩
```
Used in 1.

Reading the WEB results in a list of tokens as used by tangle or weave. At this point, I do not need to extract the structure of the Pascal program contained in the WEB. This is left for a later stage. I need to extract the WEB specific structure: text in limbo followed by modules; modules starting with TEX text followed optionally by definitions and Pascal code. Aside from this general structure, I will later need to translate the WEB specific control sequences (starting with @) by cweb specific control sequences.

The scanner identifies tokens by matching the input against regular expressions and executing C code if a match is found. The lex file web.l is not a literate program since it's not a C file; it is given verbatim in section 9. The functions and macros used in the action parts inside the file, however, are described below.

3.1 Scanning the WEB

The scanner is written following the WEB User Manual[4].

It has three main modes: the INITIAL mode (or TEX mode), the MIDDLE mode, and the PASCAL mode; and three special modes DEFINITION, FORMAT, and NAME.

⟨ external declarations 5 ⟩ ≡ (5)
```
#define TEX  INITIAL
```
Used in 2.

The scanner starts out in TEX mode scanning the part of the file that is "in limbo" and then switches back and forth between TEX mode, MIDDLE mode, and PASCAL mode, occasionally taking a detour through DEFINITION, FORMAT, or NAME mode.

While scanning in TEX mode, I need to deal with a few special characters: the character "@", because it introduces special web commands and might introduce a change into Pascal mode; the "|" character, because it starts Pascal mode; and the "{" and "}" characters , which are used for grouping while in TEX mode. Unfortunately, these same characters also start and end comments while in Pascal mode. So finding a "}" in TEX mode might be the end of a group or the end of a comment. Everything else is just considered plain text. Text may also contain the "@", "|", "{", and "}" characters if these are preceded by a backslash.

In PASCAL mode, I match the tokens needed to build the Pascal parse tree. These are different—and far more numerous—than what I need for the TEX part which my translator will not touch at all. The MIDDLE mode is a restricted PASCAL mode that does not allow module names. Instead, a module name terminates MIDDLE mode and starts a new module.

The DEFINITION mode is used to scan the initial part of a macro definition; the FORMAT mode is a simplified version of the DEFINITION mode used for format definitions; and the NAME mode is used to scan module names.

In PASCAL mode, I ignore most spaces and match the usual Pascal tokens. The main work is left to the Pascal parser.

The switching between the scanning modes is supported by a stack (see section 3.6) because it may involve nested structures. For example inside Pascal, a comment contains TEX code and inside TEX code whatever comes between two "|" characters is considered Pascal code. A scanner produced by `flex` is very fast, but by itself not capable of tracking nested structures.

3.2 Tokens

The parser creates a representation of the WEB file as a list of tokens. Later the parser will build a parse tree with tokens as leaf nodes. Because C lacks object orientation, I define **token** as a **union** of leaf nodes and internal nodes of the tree. All instances of the type defined this way share a common *tag* field as a replacement for the class information. Every token has a pointer to the *next* token, a pointer the *previous* token, a *link* field to connect related tokens, and an *up* pointer pointing from the leafs upwards and from internal nodes upwards until reaching the root node.

```
⟨ external declarations 5 ⟩ +≡                                    (6)
  typedef struct token {
    int tag;
    struct token *next, *previous, *link, *up;
    union {
      ⟨ leaf node 7 ⟩;
      ⟨ internal node 93 ⟩;
    };
  } token;
```

Leaf nodes also contain a sequence number, enumerating stretches of contiguous Pascal code, and for debugging purposes, a line number field. There is some more token specific information, that will be explained as needed.

⟨ leaf node 7 ⟩ ≡ (7)
```
struct {
  int sequenceno;
  int lineno;
  ⟨ token specific info 8 ⟩
}
```
Used in 6.

As a first example for token specific information, I note that most tokens have a *text* field that contains the textual representation of the token.

⟨ token specific info 8 ⟩ ≡ (8)
```
char *text;
```
Used in 7.

The assignment of the *tag* numbers is mostly arbitrary. The file `pascal.y` lists all possible tags and gives them symbolic names which are shown using small caps in the following. The function *tagname*, defined in `pascal.y`, is responsible for converting the tag numbers back into readable strings.

⟨ external declarations 5 ⟩ +≡ (9)
```
extern const char *tagname(int tag);
```

Because I do not deallocate tokens, I can simply allocate them from a token array using the function *new_token*.

⟨ internal declarations 3 ⟩ +≡ (10)
```
#define MAX_TOKEN_MEM 250000
```

⟨ global variables 11 ⟩ ≡ (11)
```
static token token_mem[MAX_TOKEN_MEM] = {{0}};
static int free_tokens = MAX_TOKEN_MEM;
```
Used in 1.

⟨ show summary 12 ⟩ ≡ (12)
```
DBG(dbgbasic, "free␣tokens␣=␣%d\n", free_tokens);
```
Used in 1.

⟨ functions 13 ⟩ ≡ (13)
```
static token *new_token(int tag)
{
  token *n;
  if (free_tokens > 0) n = &token_mem[--free_tokens];
  else ERROR("token␣mem␣overflow");
  n→lineno = wwlineno; n→sequenceno = sequenceno; n→tag = tag;
  return n;
}
```
Used in 1.

The value of *wwlineno*, the current line number, is maintained automatically by the code generated from `web.l`.

⟨ external declarations 5 ⟩ +≡ (14)
```
extern int wwlineno;
```

The value of *sequenceno* is taken from a global variable.

⟨ global variables 11 ⟩ +≡ (15)
```
int sequenceno = 0;
```

I increment this variable as part of the scanner actions using the macro SEQ.

⟨ external declarations 5 ⟩ +≡ (16)
```
extern int sequenceno;
#define SEQ (sequenceno++)
```

The following function is used in the parser to verify that two tokens t and s belong to the same token sequence.

⟨ external declarations 5 ⟩ +≡ (17)
```
void seq(token *t, token *s);
```

⟨ functions 13 ⟩ +≡ (18)
```
void seq(token *t, token *s)
{
  CHECK(t→sequenceno ≡ s→sequenceno,
      "tokens␣in␣line␣%d␣and␣%d␣belong␣to␣different␣code␣sequences",
      t→lineno, s→lineno);
}
```

The list of tokens is created by the function *add_token*.

⟨ external declarations 5 ⟩ +≡ (19)
```
extern token *add_token(int tag);
```

The function creates a new token and adds it to the global list of all tokens maintaining two pointers, one to the first and one to the last token of the list.

⟨ global variables 11 ⟩ +≡ (20)
```
static token *first_token;
token *last_token;
```

⟨ external declarations 5 ⟩ +≡ (21)
```
extern token *last_token;
```

I initialize the list of tokens by creating a HEAD token, and make it the first and last token of the list.

⟨ initialize token list 22 ⟩ ≡ (22)
```
first_token = last_token = new_token(HEAD); first_token→text = "";
```
Used in 4.

⟨ functions 13 ⟩ +≡ (23)
```
token *add_token(int tag)
{
  token *n = new_token(tag);
  last_token→next = n; n→previous = last_token; last_token = n; return n;
}
```

3.3 Scanner actions

Now I am ready to explain scanner actions. Let's start with the most simple cases. There are quite a few tokens, that are just added to the token list and have a fixed literal string as textual representation. I use the macro `TOK` to do this. Making `TOK` an external declaration will write its definition into the file `web2w.h` which will be included by `web.l`.

⟨ external declarations 5 ⟩ +≡ (24)
#**define** TOK(*string*, *tag*) (*add_token*(*tag*)→*text* = *string*)

Another class of simple tokens are those that have a varying textual representation which is defined by the string found in the input file. The variable *wwtext* points to this input string after it was matched against the regular expression. Since these strings are not persistent, I need to use the string handling function *copy_string* before I can store them in the tokens *text* field. The macro `COPY` can be used together with `TOK` to achieve the desired effect.

⟨ external declarations 5 ⟩ +≡ (25)
#**define** COPY *copy_string*(*wwtext*)

The last class of tokens that I discuss before I turn my attention to the functions that actually do the string-handling are the tokens where the textual representation is build up in small increments. Three macros are used to perform the desired operations: `BOS` (Begin of String) is used to start a new string, `ADD` adds characters to the current string, and `EOS` (End of String) is used to complete the definition of the string.

⟨ external declarations 5 ⟩ +≡ (26)
#**define** BOS *new_string*( )
#**define** ADD *add_string*(*wwtext*)
#**define** EOS (*string_length*( ) > 0 ? TOK(*end_string*( ), TEXT) : 0)

More string handling functions are used to define these macros and it is time to explain the string handling in more detail.

3.4 Strings

In this section, I define the following functions:

⟨ external declarations 5 ⟩ +≡ (27)

```
extern char *new_string(void);          /* start a new string */
extern void add_string(char *str);      /* add characters to the string */
extern char *end_string(void);          /* finish the string */
extern char *copy_string(char *str);    /* all of the above */
extern int string_length(void);         /* the length of the string */
```

I use a character array called *string_mem* to store these strings. Strings in the *string_mem* are never deallocated, so memory management is simple. When the scanner has identified a string, it will add it to the current string using *add_string*. The scanner can then decide when to start a new string by calling *new_string* and when the string is ready for permanent storage by calling *end_string*. *string_length* returns the length of the current string.

Some statistics: `tex.web` contains 11195 Strings with an average of 46.6 characters per string and a maximum of 5234 characters (the text in limbo); the second largest string has 1891 characters. The total number of characters in all strings is 516646. (Scanning `etex.web` will require even more string memory.)

⟨ internal declarations 3 ⟩ +≡ (28)

```
#define MAX_STRING_MEM 800000
```

⟨ global variables 11 ⟩ +≡ (29)

```
static char string_mem[MAX_STRING_MEM];
static int free_strings = MAX_STRING_MEM;
static int current_string = 0;
```

⟨ show summary 12 ⟩ +≡ (30)

```
DBG(dbgbasic, "free␣strings␣=␣%d\n", free_strings);
```

The string currently under construction is identified by the position of its first character, the *current_string*, and its last character `MAX_STRING_MEM` − *free_strings*.

⟨ functions 13 ⟩ +≡ (31)

```
char *new_string(void)
{
  current_string = MAX_STRING_MEM - free_strings;
  return string_mem + current_string;
}
void add_string(char *str)
{
  while (free_strings > 0) {
    if (*str ≠ 0) string_mem[MAX_STRING_MEM - free_strings--] = *str++;
    else return;
  }
  ERROR("String␣mem␣overflow");
}
char *end_string(void)
{
  char *str = string_mem + current_string;
  if (free_strings > 0) string_mem[MAX_STRING_MEM - free_strings--] = 0;
  else ERROR("String␣mem␣overflow");
  current_string = MAX_STRING_MEM - free_strings; return str;
}
char *copy_string(char *str)
{ new_string(); add_string(str); return end_string(); }
int string_length(void)
{ return (MAX_STRING_MEM - free_strings) - current_string; }
```

3.5 Identifiers

To be able to parse the embedded Pascal code, I need to take special care of identifiers. I keep information related to identifiers in a table, called the *symbol_table*. The table is accessed by the string representing the identifier as a key and it returns a pointer to the table entry, called a **symbol**.

```
⟨ external declarations 5 ⟩ +≡                                    (32)
  typedef struct symbol {
    char *name;
    int tag;
    int obsolete;
    int for_ctrl;
    int value;
    struct symbol *link;
    token *type;
    token *eq;
  } symbol;
  extern int get_sym_no(char *name);
  extern symbol *symbol_table[ ];
```

```
⟨ internal declarations 3 ⟩ +≡                                     (33)
#define MAX_SYMBOL_TABLE 6007                    /* or 4001 4999, a prime */
#define MAX_SYMBOLS 5200          /* must be less than MAX_SYMBOL_TABLE */
```

```
⟨ global variables 11 ⟩ +≡                                         (34)
  symbol *symbol_table[MAX_SYMBOL_TABLE] = {NULL};
  static symbol symbols[MAX_SYMBOLS] = {{0}};
  static int free_symbols = MAX_SYMBOLS;
```

```
⟨ show summary 12 ⟩ +≡                                             (35)
  DBG(dbgbasic, "free␣symbols␣=␣%d\n", free_symbols);
```

I organize the symbol table as a hash table using double hashing as described in [7], Chapter 6.4.

```
⟨ functions 13 ⟩ +≡                                                (36)
  static int symbol_hash(char *name)
  {
    int hash = 0;
    while (*name ≠ 0) hash = hash + (*(name++) ⊕ #9E);
    return hash;
  }
  static symbol *new_symbol(void)
  {
    CHECK(free_symbols > 0, "Symbol␣table␣overflow"); free_symbols--;
    return symbols + free_symbols;
  }
  int get_sym_no(char *name)
  {
```

```
  int i, c;
  i = symbol_hash(name) % MAX_SYMBOL_TABLE;
  if (symbol_table[i] ≠ NULL) {
    if (strcmp(symbol_table[i]→name, name) ≡ 0) return i;
    if (i ≡ 0) c = 1;
    else c = MAX_SYMBOL_TABLE − i;
    while (true) {
      i = i − c;
      if (i < 0) i = i + MAX_SYMBOL_TABLE;
      if (symbol_table[i] ≡ NULL) break;
      if (strcmp(symbol_table[i]→name, name) ≡ 0) return i;
    }
  }
  symbol_table[i] = new_symbol( ); symbol_table[i]→name = new_string( );
  add_string(name); end_string( ); symbol_table[i]→tag = ID; return i;
}
```

The pointer into the symbol table can be stored inside the token in two ways: as an index into the *symbol_table* or as a direct pointer to the **symbol** structure. While scanning the WEB, I will assign the symbol number(*sym_no*), and while parsing Pascal, I will replace the symbol number by the symbol pointer (*sym_ptr*). This is necessary, because I will need to distinguish between various local symbols with the same name; these have only a single entry in the symbol table but the pointers will point to different **symbol** structures.

```
⟨ token specific info 8 ⟩ +≡                                    (37)
  int sym_no;
  struct symbol *sym_ptr;
```

This leads to the following macros:

```
⟨ external declarations 5 ⟩ +≡                                  (38)
#define SYM_PTR(name) symbol_table[get_sym_no(name)]
#define SYMBOL
  { int s = get_sym_no(yytext); add_token(symbol_table[s]→tag)→sym_no = s; }
#define SYM(t) (symbol_table[(t)→sym_no])
```

It's easy to convert such a token back to a string.

```
⟨ convert token t to a string 39 ⟩ ≡                             (39)
case ID: case PID: case PCONSTID: case PARRAYFILETYPEID:
  case PARRAYFILEID: case PFUNCID: case PPROCID: case PDEFVARID:
  case PDEFPARAMID: case PDEFREFID: case PDEFCONSTID:
  case PDEFTYPEID: case PDEFTYPESUBID: case PDEFFUNCID: case CREFID:
  case NMACRO: case OMACRO: case PMACRO:
  return SYM(t)→name;                                           Used in 99.
```

In TEX, like in most programs, I encounter two kinds of symbols: global and local symbols. While scanning, every symbol that I encounter gets entered into the global symbol table. While parsing, I will discover, that the variable *f* is a file variable in one function and an integer variable in another function. The two

occurrences of f have different scope. So I want to link different occurrences of f to different entries in the symbol table.

I use the function *localize* to create a local version of a symbol.

⟨ external declarations 5 ⟩ +≡ (40)
```
extern void localize(token *t);
```

To open a new scope, I use the function *scope_open*; to close it again, I use the function *scope_close*.

⟨ external declarations 5 ⟩ +≡ (41)
```
extern void scope_open(void);
extern void scope_close(void);
```

These functions use a small array holding all the symbol numbers of currently local symbols and another array to hold pointers to the global symbols of the same name.

⟨ global variables 11 ⟩ +≡ (42)
```
#define MAX_LOCALS 50
static int locals[MAX_LOCALS];
static symbol *globals[MAX_LOCALS];
static int free_locals = MAX_LOCALS;
```

⟨ functions 13 ⟩ +≡ (43)
```
void scope_open(void)
{
  CHECK(free_locals ≡ MAX_LOCALS,
      "Opening␣a␣new␣scope␣without␣closing␣the␣previous␣one");
}
void scope_close(void)
{
  int i;
  for (i = free_locals; i < MAX_LOCALS; i++) symbol_table[locals[i]] = globals[i];
  free_locals = MAX_LOCALS;
}
```

To localize a symbol, I create a new one and enter it, after saving the global symbol, into the symbol table.

⟨ functions 13 ⟩ +≡ (44)
```
void localize(token *t)
{
  int sym_no = t→sym_no;
  symbol *l, *g;
  l = new_symbol( ); g = symbol_table[sym_no]; l→name = g→name;
  l→tag = g→tag; l→eq = g→eq; symbol_table[sym_no] = l;
  CHECK(free_locals > 0, "Overflow␣of␣local␣symbols␣in␣line␣%d",
      t→lineno); free_locals−−; locals[free_locals] = sym_no;
  globals[free_locals] = g; t→sym_ptr = l;
}
```

3.6 Linking related tokens

So far I have considered the WEB file as one long flat list of tokens. As already mentioned above, the file has, however, also a nested structure: For example, each "{" token is related to a "}" token that ends either a TEX group or a Pascal comment. While scanning, I will need to know about this structure because it is necessary to do a correct switching of modes. Hence, I use the *link* field to connect the first token to the later token. This information is also useful at later stages, for example when I expand macros. The following table gives a list of related tokens.

Left	Right	Mode	Comment
(	)	PASCAL/PASCAL	needed for macro expansion
{	}	PASCAL/TEX/PASCAL	comments
{	}	MIDDLE/TEX/MIDDLE	comments
{	}	TEX/TEX	grouping
|	|	TEX/PASCAL/TEX	typesetting code
@<	@>		module names
=			begin of Pascal
==			begin of Pascal
	@	PASCAL	end of Pascal
	@*	PASCAL	end of Pascal
	@d	PASCAL	end of Pascal
	@f	PASCAL	end of Pascal
	@p	PASCAL	end of Pascal
"	"		list of WEB strings
@>=	@>=		continuation of module
@p	@p		continuation of program

Tab. 1: List of linked tokens

To track the nesting of structures, I need a stack:

⟨ global variables 11 ⟩ +≡ (45)

```
#define MAX_WWSTACK 200
  static token *wwstack[MAX_WWSTACK] = {0};
  static int wwsp = 0;
```

I define the functions *ww_push* and *ww_pop* to operate on the stack. When popping a token, I keep the nesting information by linking it to its matching token. The function *ww_is* can be used to test the *tag* of the token on top of the stack.

⟨ external declarations 5 ⟩ +≡ (46)

```
  extern void ww_push(token *t);
  extern token *ww_pop(token *t);
  extern int ww_is(int tag);
```

```
⟨ functions 13 ⟩ +≡                                                    (47)
  void ww_push(token *left)
  {
    CHECK(wwsp < MAX_WWSTACK, "WW␣stack␣overflow");
    DBG(dbglink, "Pushing[%d]:", wwsp);
    if (left ≠ NULL) DBG(dbglink, THE_TOKEN(left));
    wwstack[wwsp++] = left;
  }
  token *ww_pop(token *right)
  {
    token *left;
    CHECK(wwsp > 0, "Mode␣stack␣underflow"); left = wwstack[--wwsp];
    if (left ≠ NULL) left→link = right;
    DBG(dbglink, "Popping[%d]:", wwsp);
    if (left ≠ NULL) DBG(dbglink, THE_TOKEN(left));
    return left;
  }
  int ww_is(int tag)
  {
    return wwsp > 0∧wwstack[wwsp−1] ≠ NULL∧wwstack[wwsp−1]→tag ≡ tag;
  }
```

Using the stack, I can now also distinguish the use of "{" and "}" as a grouping construct in TEX from the use of starting and ending comments in Pascal. When I encounter "{" in TEX mode, it introduces a new level of grouping and I do not create a new token. Instead I push NULL on the stack. When I encounter "{" in PASCAL mode, however, it is the start of a comment; I create a token and push it. When I encounter the matching "}", I am always in TEX mode. I pop the stack and test the value: If it was NULL, I can continue in TEX mode because it was a grouping character; if it was not NULL, it is the end of a comment. I create a token for it and continue in PASCAL mode.

```
⟨ external declarations 5 ⟩ +≡                                         (48)
#define PUSH  ww_push(last_token)
#define PUSH_NULL  ww_push(NULL)
#define POP  ww_pop(last_token)
#define POP_NULL  (ADD, POP)
#define POP_MLEFT  (EOS, TOK("}", RIGHT), BEGIN(MIDDLE), POP)
#define POP_PLEFT  (EOS, TOK("}", RIGHT), BEGIN(PASCAL), POP)
#define POP_LEFT
  (ww_is(MLEFT) ? POP_MLEFT : (ww_is(PLEFT) ? POP_PLEFT : POP_NULL))
```

Besides linking matching tokens, the *link* field can also be used to build linear list of related token. One example for such a list is the list of WEB strings. The program tangle, converting a WEB to Pascal, creates a string pool file. This mechanism is no longer available in ctangle so I have to implement an alternative (see section 5.8 on replacing the TEX string pool). Here I take the first step and collect all the

strings that occur in the WEB in one linked list. For this purpose, I use a pointer to the *first_string*, and a pointer to the link field of the *last_string*. The macro WWSTRING is used in the scanner and adds the new string token to this list.

⟨ external declarations 5 ⟩ +≡ (49)
```
extern void wwstring(char *wwtext);
#define WWSTRING wwstring(wwtext)
```

⟨ functions 13 ⟩ +≡ (50)
```
void wwstring(char *wwtext)
{
  token *t = add_token(STRING);
  t->sym_no = get_sym_no(wwtext); t->text = SYM(t)->name;
  *last_string = t; last_string = &(t->link);
}
```
To make this work, it is sufficient to initialize the two pointers appropriately.

⟨ global variables 11 ⟩ +≡ (51)
```
static token *first_string = NULL, **last_string = &first_string;
```

3.7 Module names

I need to maintain information for each module. I keep this information in a table, called the module table. The table is accessed by the string representing the module name as a key. This sounds very similar to what I did for identifiers, there is, however, one main difference: Modules are sometimes referenced by incomplete module names that end with an ellipsis (...). These incomplete module names may not even be valid TEX code. For this reason, I use a binary search tree to map module names to modules. The first thing I need, therefore, is a function to compare two module names. The function *module_cmp*(n, m) will compare the name of n to the name of m; it returns a negative value if $n < m$; zero if $n = m$; and a positive value if $n > m$. m is always a full module name, n might end abruptly with an ellipsis.

⟨ functions 13 ⟩ +≡ (52)
```
static int module_name_cmp(token *n, token *m)
{
  n = n->next; m = m->next;          /* advance from "@<" to the name */
  if (n->next->tag ≡ ELIPSIS)
    return strncmp(n->text, m->text, strlen(n->text));
  else return strcmp(n->text, m->text);
}
```
I organize the module table as a binary tree and allocate new modules from a large array.

⟨ internal declarations 3 ⟩ +≡ (53)
```
#define MAX_MODULE_TABLE 1009      /* or 1009, 1231, 2017, 3001, a prime */
```

⟨ global variables 11 ⟩ +≡ (54)
```
static module module_table[MAX_MODULE_TABLE] = {{0}};
```

```
static int free_modules = MAX_MODULE_TABLE;
static module *module_root = NULL;
```

⟨ external declarations 5 ⟩ +≡ (55)

```
typedef struct module {
  token *atless;
  token *atgreater;
  struct module *left, *right;
} module;
extern void add_module(token *atless);
extern module *find_module(token *atless);
```

⟨ show summary 12 ⟩ +≡ (56)

```
DBG(dbgbasic, "free␣modules␣=␣%d\n", free_modules);
```

To look up a module in the module table I use the function *find_module*. It returns a pointer to the module given the pointer to the "`@<`" token that starts the module name. The function will allocate a new module if needed.

⟨ functions 13 ⟩ +≡ (57)

```
module *find_module(token *atless)
{
  module **m = &module_root;
  while (*m ≠ NULL) {
    int d = module_name_cmp(atless, (*m)→atless);
    if (d ≡ 0) return *m;
    else if (d < 0) m = &((*m)→left);
    else m = &((*m)→right);
  }
  CHECK(free_modules > 0, "Module␣table␣overflow");
  *m = module_table + MAX_MODULE_TABLE − free_modules−−;
  (*m)→atless = atless; return *m;
}
```

Because modules can be defined in multiple installments, I link together the closing "`@>`" tokens. This is done by calling the function *add_module* whenever I find the two tokens "`@>=`".

⟨ functions 13 ⟩ +≡ (58)

```
void add_module(token *atless)
{
  module *m = find_module(atless);
  token *atgreater = m→atgreater;
  if (atgreater ≡ NULL) m→atgreater = atless→link;
  else {
    while (atgreater→link ≠ NULL) atgreater = atgreater→link;
    atgreater→link = atless→link;
  }
}
```

Next I consider the problem of scanning module names. The name of a module starts after a "@<" token. If this token shows up, I have to do some preparations depending on the current mode: If I am in TEX mode, I need to terminate the current TEXT token; if I am in MIDDLE mode, I pop the stack and terminate the macro or format definition I were just scanning; no special preparation is needed if I am in PASCAL mode. Then I push the "@<" token on the stack, start a new TEXT token, and switch to NAME mode. When I encounter the matching "@>" or "@>=" token, I add the module to the module table—calling *find_module* to cover the case that this is the first and only complete occurrence of the module name.

⟨ external declarations 5 ⟩ +≡ (59)
#**define** AT_GREATER_EQ
  TOK("@>", ATGREATER), *add_module*(POP), TOK("=", EQ), PUSH, SEQ
#**define** AT_GREATER TOK("@>", ATGREATER), *find_module*(POP)

You may have noticed that the above AT_GREATER_EQ macro pushes the EQ token on the stack. I match this token up with the token that ends the Pascal code following the equal sign. As you will see below, I do the same for macro definitions. Further, I link all the unnamed modules together using the "@p" tokens. I add an extra EQ token to match the convention that I have established for named modules.

⟨ external declarations 5 ⟩ +≡ (60)
  extern token ∗*program*;
#**define** PROGRAM
  (*program*→*link* = *last_token*, *program* = *last_token*), TOK("", EQ)

I use the first token as list head.

⟨ global variables 11 ⟩ +≡ (61)
  token ∗*program*;

⟨ initialize token list 22 ⟩ +≡ (62)
  program = *first_token*;

3.8 Definitions

In a WEB file, the token "@d" introduces the definition of a numeric constants or a macro with or without parameter. When the scanner encounters such a token, it enters the DEFINITION mode.

The first token in DEFINITION mode is an identifier which will be stored in the symbol table. Then follows an optional macro parameter "(#)"; after the single or double equal sign, the scanner switches to MIDDLE mode, not without pushing the equal sign on the stack to be matched against the first token after the following Pascal code.

After scanning an "=" token, I know that a numeric macro is following, and I record this fact by changing the *tag* of the identifier in the token and in the symbol table.

⟨ external declarations 5 ⟩ +≡ (63)
#**define** CHGTAG(*t*, *x*) ((*t*)→*tag* = (*x*))
#**define** CHGID(*t*, *x*) (SYM(*t*)→*tag* = (*x*))
#**define** CHGTYPE(*t*, *x*) (SYM(*t*)→*type* = (*x*))

```
#define CHGVALUE(t,x) (SYM(t)→value = (x))
#define CHGTEXT(t,x) ((t)→text = (x))
```

After scanning an "==" token, I know that I have either an ordinary macro or a parametrized macro. A PARAM token tells the difference.

```
⟨ functions 13 ⟩ +≡                                                    (64)
  void def_macro(token *eq, int tag)
  {
    token *id;
    if (eq→previous→tag ≡ PARAM) {
      id = eq→previous→previous; tag = PMACRO;
    }
    else {
      id = eq→previous;
    }
    CHGTAG(id, tag); CHGID(id, tag); SYM(id)→eq = eq;
    DBG(dbgexpand, "Defining␣%s:␣%s\n", tagname(tag), SYM(id)→name);
  }
```

```
⟨ external declarations 5 ⟩ +≡                                         (65)
  extern void def_macro(token *eq, int tag);
#define DEF_MACRO(tag) def_macro(last_token, tag), SEQ
```

Similar, the token "@f" introduces a format specification switching the scanner to FORMAT mode. It then scans tokens until the first newline character brings the scanner back to MIDDLE mode.

3.9 Finishing the token list

When the scanner is done, I terminate the token list with two end of file tokens: one for Pascal and one for the WEB.

```
⟨ finalize token list 66 ⟩ ≡                                           (66)
  TOK("", ATP); PROGRAM; PUSH; TOK("", PEOF); TOK("", WEBEOF); POP;
```

Used in 4.

At this point I might want to have a complete list of all tokens and identifiers for debugging purposes.

```
⟨ finalize token list 66 ⟩ +≡                                          (67)
  if (debugflags & dbgtoken) {
    token *t = first_token;
    while (t ≠ NULL) { MESSAGE(THE_TOKEN(t)); t = t→next; }
  }
  if (debugflags & dbgid) {
    int i;
    for (i = free_symbols; i < MAX_SYMBOLS; i++)
      MESSAGE("symbol[%d]=%s\n", i, symbols[i].name);
  }
```

4 Parsing Pascal

I use `bison` (the free replacement of `yacc`) to implement the parser. Fortunately TEX does not use the full Pascal language, so the parser can be simpler. Further, I do not need to generate code, but just analyze the Pascal programs for the purpose of finding those constructions where Pascal differs from C and need a conversion. If I discover such an instance, I change the tags of the affected tokens, set the *link* field to connect related tokens, or even construct a parse tree and link to it using the *up* field. In a next sweep over the token list in section 5, these changed tokens will help us make the appropriate transformations. But before I can do this, I need to feed the parser with the proper tokens, but not in the order I find them in the `WEB` file. I have to "tangle" them to get them into Pascal program order. The function that is supposed to deliver the tangled tokens is called *pplex*. In addition, the parser expects a function *pperror* to produce error messages.

```
⟨ external declarations 5 ⟩ +≡                                    (68)
  extern int pplex(void);
  extern void pperror(const char *message);
```

The function *pperror* is very simple:

```
⟨ functions 13 ⟩ +≡                                                (69)
  void pperror(const char *message)
  {
    ERROR("Parse␣error␣line␣%d:␣%s", pplval→lineno, message); }
```

4.1 Generating the sequence of Pascal tokens

Primarily, the Pascal tokens come from the unnamed modules and then from expanding module names and macros. Because modules and macros may reference other modules and macros, I will need a stack to keep track of where to continue expansion when I have reached the end of the current expansion.

```
⟨ global variables 11 ⟩ +≡                                         (70)
#define MAX_PPSTACK 40
  static struct {
    token *next, *end, *link;
    int environment;
    token *parameter;
  } pp_stack[MAX_PPSTACK];
  static int pp_sp = MAX_PPSTACK;
```

In each stack entry, the pointers *next* and *end* point to the next and past the last token of the current replacement text. In the case of modules, where the replacement text for the module name might be defined in multiple installments, the *link* pointer is used to point to the continuation of the current replacement text.

In the *parameter* field, I store the pointer to the "`(`" token preceding the parameter of a parametrized macro; it provides us conveniently with a pointer to the parameter text with its *next* pointer and with its *link* pointer to the "`)`" token a pointer directly to the *end* of the parameter text. When I expand the parameter text of a parametrized macro, I need the *environment* variable. It points down the stack to the stack entry that contains the macro call. This is the place where I will find the replacement for a "`#`" token that might occur in the parameter text of nested parametrized macros.

The function *pp_push* will store the required information on the stack. Instead of passing the *next* and *end* pointer separately, I pass a pointer to the "`=`" token from the macro or module definition. This token conveniently contains both pointers. The function then advances the stack pointer, initializes the new stack entry, and returns the pointer to the first token of the replacement. *pp_pop* will pop the stack and again return the pointer to the next token.

```
⟨ functions 13 ⟩ +≡                                                    (71)
  token *pp_push(token *link, token *eq, int environment, token *parameter)
  {
    CHECK(pp_sp > 0, "Pascal␣lexer␣stack␣overflow"); pp_sp −−;
    pp_stack[pp_sp].link = link; pp_stack[pp_sp].next = eq→next;
    pp_stack[pp_sp].end = eq→link;
    pp_stack[pp_sp].environment = environment;
    pp_stack[pp_sp].parameter = parameter;
    DBG(dbgexpand, "Push␣pplex[%d]:␣", MAX_PPSTACK − pp_sp);
    DBGTOKS(dbgexpand, eq→next, eq→link); return pp_stack[pp_sp].next;
  }
  token *pp_pop(void)
  {
    CHECK(pp_sp < MAX_PPSTACK, "Pascal␣Lexer␣stack␣underflow");
    pp_sp++; DBG(dbgexpand, "Pop␣pplex[%d]:␣", MAX_PPSTACK − pp_sp);
    DBGTOKS(dbgexpand, pp_stack[pp_sp].next, pp_stack[pp_sp].end);
    return pp_stack[pp_sp].next;
  }
```

The function *pplex* is what I write next. In an "endless" loop, I read the next token from the stack just described, popping and pushing the stack as necessary. If I find a Pascal token—it has a *tag* value greater than FIRST_PASCAL_TOKEN—I can return its *tag* immediately to the parser. `WEB` tokens receive special treatment. When I deliver a token to the parser, *pplval*, the semantic value of the token, is the token pointer itself.

```
⟨ functions 13 ⟩ +≡                                                    (72)
  int pplex(void)
```

```
{
  token *t;
  int tag;
  t = pp_stack[pp_sp].next;
  while (true) {
    if (t ≡ pp_stack[pp_sp].end) {
      ⟨ process the end of a code segment 87 ⟩
      continue;
    }
    tag = t→tag;
  tag_known:
    if (tag > FIRST_PASCAL_TOKEN) {
      pp_stack[pp_sp].next = t→next; goto found;
    }
    else {
      switch (tag) {
        ⟨ special treatment for WEB tokens 73 ⟩
      default: ERROR("Unexpected token in pplex:"THE_TOKEN(t));
      }
    }
  }
found:
  DBG(dbgpascal, "pplex: %s->\t", tagname(tag));
  DBG(dbgpascal, THE_TOKEN(t));
  if (pascal ≠ NULL) fprintf(pascal, "%s ", token2string(t));
  pplval = t; return tag;
}
```

4.2 Simple cases for the parser

Now let's look at all the WEB tokens and what *pplex* should to do with them. Quite a lot of them can be simply skipped:

```
⟨ special treatment for WEB tokens 73 ⟩ ≡                                (73)
case NL: case INDENT:
  if (pascal ≠ NULL) fprintf(pascal, "%s", token2string(t));
case METACOMMENT: case ATT: case ATEX: case ATQM: case ATPLUS:
  case ATSLASH: case ATBACKSLASH: case ATBAR: case ATHASH:
  case ATCOMMA: case ATINDEX: case ATINDEXTT: case ATINDEX9:
  case ATAND: case ATSEMICOLON: case ATLEFT: case ATRIGHT:
  t = t→next; continue;                                  Used in 72.
```

Comments can be skipped in a single step:

```
⟨ special treatment for WEB tokens 73 ⟩ +≡                               (74)
case MLEFT: case PLEFT: t = t→link→next; continue;
```

The Pascal end-of-file token is passed to the parser which then should terminate.

```
⟨ special treatment for WEB tokens 73 ⟩ +≡                               (75)
```

case PEOF: $pp\_stack[pp\_sp].next = t \rightarrow next$; **goto** *found*;

Simple is also the translation of octal or hexadecimal constants and single character strings: I translate them as Pascal integers. The token "`@$`", it's the string pool checksum, is an integer as well.

⟨ special treatment for `WEB` tokens 73 ⟩ +≡ (76)

```
case ATDOLLAR: case OCTAL: case HEX: case CHAR:
  pp_stack[pp_sp].next = t→next; tag = PINTEGER; goto found;
```

The last simple case are identifiers. For identifiers, I find the correct tag in the symbol table which is maintained by the parser. At this point, I give tokens that still have the $tag \equiv$ ID the default tag PID and link tokens to the actual symbol structure, which might be local or global.

⟨ special treatment for `WEB` tokens 73 ⟩ +≡ (77)

```
case ID:
  {
    symbol *s = SYM(t);
    tag = s→tag;
    if (tag ≡ ID) tag = s→tag = PID;
    t→sym_ptr = s; t→tag = tag; goto tag_known;
  }
```

4.3 The macros debug, gubed, and friends

TEX does some special trickery with the pseudo keywords **debug**, **gubed**, **init**, **tini**, **stat**, and **tats**. These identifiers are used to generate different versions of TEX for debugging, initialization, and gathering of statistics. The natural way to do this in C is the use of # **ifdef**... # **endif**. It is however not possible in C to define a macro like "# **define debug** # **ifdef** `DEBUG`" because the C preprocessor performs a simple one-pass replacement on the source code. So macros are expanded and the expansion is not expanded a second time.
It would be possible to define a module ⟨ `debug` 123 ⟩ that `ctangle` expands to "# **ifdef** `DEBUG`" before the C preprocessor sees it; the other possibility is to do the expansion right now in `web2w`. The latter possibility is simple, so I do it here, but it affects the visual appearance of the converted code to its disadvantage.

There are further possibilities too: I could redefine the macro as "# **define debug** **if** (*Debug*) {" making it plain C code. Then the compiler would insert or optimize away the code in question depending on whether I say "# **define** *Debug* 1" or "# **define** *Debug* 0". The **stat**...**tats** brackets are however often used to enclose variable- or function-definitions where an "**if** (*Debug*) {" would not work.

There are, however, also instances where the "# **ifdef** `DEBUG`" approach does not work. For instance, **debug**...**gubed** is used inside the macro *succumb*. Fortunately there are only a few of these instances and I deal with them in the patch file.

As far as the parser is concerned, I just skip these tokens.

⟨ special treatment for `WEB` tokens 73 ⟩ +≡ (78)

```
case WDEBUG: case WGUBED: case WINIT: case WTINI: case WSTAT:
  case WTATS: t = t→next; continue;
```

Later, I get them back into the `cweb` file using the following code. It takes care not to replace the special keywords when they are enclosed between vertical bars and are only part of the descriptive text.

```
⟨ convert t from WEB to cweb 79 ⟩ ≡                                    (79)
case WDEBUG:
  if (t→previous→tag ≡ BAR) wputs(t→text);
  else {
    if (column ≠ 0) wput('\n');
    wputs("#ifdef␣@!DEBUG\n");
  }
  t = t→next; break;
case WINIT:
  if (t→previous→tag ≡ BAR) wputs(t→text);
  else {
    if (column ≠ 0) wput('\n');
    wputs("#ifdef␣@!INIT\n");
  }
  t = t→next; break;
case WSTAT:
  if (t→previous→tag ≡ BAR) wputs(t→text);
  else {
    if (column ≠ 0) wput('\n');
    wputs("#ifdef␣@!STAT\n");
  }
  t = t→next; break;
case WGUBED: case WTINI: case WTATS:
  if (t→previous→tag ≡ BAR) wputs(t→text);
  else {
    if (column ≠ 0) wput('\n');
    wputs("#endif\n");
  }
  t = t→next;
  if (t→tag ≡ ATPLUS ∨ t→tag ≡ ATSLASH) t = t→next;
  if (t→tag ≡ NL) t = t→next;
  break;                                                  Used in 101.
```

I ignore "`@+`" tokens that precede **debug** and friends, because their replacement should always start on the beginning of a line.

```
⟨ convert t from WEB to cweb 79 ⟩ +≡                                   (80)
case ATPLUS: t = t→next;
  if (¬following_directive(t)) wputs("@+");
  else DBG(dbgcweb, "Eliminating␣@+␣in␣line␣%d\n", t→lineno); break;
```

Because they also occur in format definitions, I mark the identifiers as obsolete.

⟨ finalize token list 66 ⟩ +≡ (81)
```
SYM_PTR("debug")→obsolete = 1; SYM_PTR("gubed")→obsolete = 1;
SYM_PTR("stat")→obsolete = 1; SYM_PTR("tats")→obsolete = 1;
SYM_PTR("init")→obsolete = 1; SYM_PTR("tini")→obsolete = 1;
```

4.4 Parsing numerical constants

I do not expand numerical macros, instead I expand the Pascal grammar to handle NMACRO tokens. This is also the right place to switch numeric macros from symbol numbers to symbol pointers. For each use of the token, I increment its *value* field in the symbol table. This will allow us later to eliminate definitions that are no longer used. The handling of WEB strings is similar.

⟨ special treatment for WEB tokens 73 ⟩ +≡ (82)
```
case NMACRO: t→sym_ptr = SYM(t);
  if (t→sym_ptr→eq→next→tag ≡ STRING) {
    token *s = t→sym_ptr→eq→next;
    s→sym_ptr = SYM(s); s→sym_ptr→value++;
    DBG(dbgstring, "Using␣numeric␣macro␣%s␣(%d)␣in␣line␣%d\n",
        s→sym_ptr→name, s→sym_ptr→value, t→lineno);
  }
  pp_stack[pp_sp].next = t→next; goto found;
case STRING: t→sym_ptr = SYM(t); t→sym_ptr→value++;
  DBG(dbgstring, "Using␣string␣%s␣(%d)␣in␣line␣%d\n", t→sym_ptr→name,
      t→sym_ptr→value, t→lineno); pp_stack[pp_sp].next = t→next;
  goto found;
```

Occasionally, I will need the ability to determine the value of a token that the Pascal parser considers an integer. The function *getval* will return this value.

⟨ external declarations 5 ⟩ +≡ (83)
```
extern int getval(token *t);
```

⟨ functions 13 ⟩ +≡ (84)
```
int getval(token *t)
{
  int n = 0;
  switch (t→tag) {
  case ATDOLLAR: n = 0; break;
  case PINTEGER: n = strtol(t→text, NULL, 10); break;
  case NMACRO: t = SYM(t)→eq; CHECK(t→tag ≡ EQEQ,
      "=␣expected␣in␣numeric␣macro␣in␣line␣%d", t→lineno);
    t = t→next;
    if (t→tag ≡ PMINUS) {
      t = t→next; n = -getval(t);
    }
    else n = getval(t);
    while (true) {
      if (t→next→tag ≡ PPLUS) {
```

```
        t = t→next→next;  n = n + getval(t);
      }
      else if (t→next→tag ≡ PMINUS) {
        t = t→next→next;  n = n − getval(t);
      }
      else break;
    }
    break;
  case OCTAL: n = strtol(t→text + 2, NULL, 8); break;
  case HEX: n = strtol(t→text + 2, NULL, 16); break;
  case CHAR: n = (int)(unsigned char) t→text[1]; break;
  case PCONSTID: n = SYM(t)→value; break;
  default: ERROR("Unable to get value for tag %s in line %d", TAG(t),
      t→lineno);
  }
  return n;
}
```

Notice that I assume that tokens which are tagged as constant identifiers are expected to have a value stored in the symbol table. We write this value using the macro `SETVAL`.

⟨ external declarations 5 ⟩ +≡ (85)

```
#define SETVAL(t, val) SYM(t)→value = val
```

4.5 Expanding module names and macros

Now let's turn to the more complicated operations, for example the expansion of module names. I know that I hit a module name when I encounter an "`@<`" token. At this point, I advance the current token pointer past the end of the module name, look up the module in the module table, and push its first code segment.

⟨ special treatment for `WEB` tokens 73 ⟩ +≡ (86)

```
case ATLESS:
  {
    token *eq, *atgreater;
    atgreater = find_module(t)→atgreater;
    CHECK(atgreater ≠ NULL, "Undefined module @<%s...@> in line %d",
        token2string(t→next), t→lineno);
    DBG(dbgexpand, "Expanding module @<%s@> in line %d\n",
        token2string(t→next), t→lineno);  eq = atgreater→next;
    pp_stack[pp_sp].next = t→link→next;
    t = pp_push(atgreater→link, eq, 0, NULL);  continue;
  }
```

When I reach the end of the code segment, I can check the link field to find its continuation.

⟨ process the end of a code segment 87 ⟩ ≡ (87)

```
  token *link = pp_stack[pp_sp].link;
```

```
if (link ≠ NULL) {
  token *eq;
  eq = link→next; link = link→link; pp_pop( );
  t = pp_push(link, eq, 0, NULL);
}
else t = pp_pop( );
```

Used in 72.

Slightly simpler are ordinary macros.

⟨ special treatment for WEB tokens 73 ⟩ +≡ (88)

```
case OMACRO:
  {
    token *eq;
    eq = SYM(t)→eq; pp_stack[pp_sp].next = t→next;
    DBG(dbgexpand, "Expanding␣ordinary␣macro␣%s␣in␣line␣%d\n",
        token2string(t), t→lineno); t = pp_push(NULL, eq, 0, NULL); continue;
  }
```

There are a few macros, that are special; I do not want to expand them but instead generate special tokens in `web.l` and expand the Pascal grammar to cope with them directly. It remains, however, to mark them as obsolete to remove the macro definitions form the `cweb` output.

⟨ finalize token list 66 ⟩ +≡ (89)

```
SYM_PTR("return")→obsolete = 1; SYM_PTR("endcases")→obsolete = 1;
SYM_PTR("othercases")→obsolete = 1; SYM_PTR("mtype")→obsolete = 1;
SYM_PTR("final_end")→obsolete = 1;
```

4.6 Expanding macros with parameters

Now I come to the most complex case: parametrized macros. When the `WEB` invokes a parametrized macro as part of the Pascal code, the macro identifier is followed by a "`(`" token, the parameter tokens, and a matching "`)`" token. The `WEB` scanner has also set the *link* field of the "`(`" token to point to the "`)`" token. The replacement text for the macro is found in the same way as for ordinary macros above. The replacement text, however, may now contain a "`#`" token, asking for another replacement by the parameter tokens. The whole process can be nested because the parameter tokens may again contain a "`#`" token. Hence, I need to store the parameter tokens on the stack as well as a reference to the enclosing environment. I store a reference to the "`(`" token on the stack, because from it, I can get the first token and the last token of the replacement text.

I can write now the code to expand a parametrized macro. To cope with cases like `font(x)`, `font == type` and `type(#)=mem[#]`, I call *pplex* to find the opening parenthesis before pushing the macro expansion and its parameter. (Note: I expand `font` as an ordinary macro; then find `type` which is a parametrized macro and end up in the "**case** PMACRO:" below. The "`(`" token is not the next token after `type` because I am still expanding `font`. Calling *pplex* will reach the end of the expansion, pop the stack, and then find the "`(`" token.)

⟨ special treatment for WEB tokens 73 ⟩ +≡ (90)

```
case PMACRO:
  {
    token *open, *eq;
    int popen;
    DBG(dbgexpand, "Expanding␣parameter␣macro␣%s␣in␣line␣%d\n",
        token2string(t), t→lineno); eq = SYM(t)→eq;
    pp_stack[pp_sp].next = t→next; popen = pplex( );
    CHECK(popen ≡ POPEN, "expected␣(␣after␣macro␣with␣parameter");
    open = pplval; pp_stack[pp_sp].next = open→link→next;
    ⟨ count macro parameters 144 ⟩
    t = pp_push(NULL, eq, pp_sp, open); continue;
  }
```

While traversing the replacement text, I may find a "`#`" token. In this case, I find on the *pp_stack* the pointer to the *parameter* and, in case the *parameter* contains again a "`#`" token, its *environment*.

```
⟨ special treatment for WEB tokens 73 ⟩ +≡                    (91)
case HASH:
  {
    token *parameter = pp_stack[pp_sp].parameter;
    int environment = pp_stack[pp_sp].environment;
    pp_stack[pp_sp].next = t→next; t = pp_push(NULL, parameter,
        pp_stack[environment].environment, pp_stack[environment].parameter);
    continue;
  }
```

4.7 The function *ppparse*

The function *ppparse* is implemented in the file `pascal.y` which must be processed by `bison` (the free version of `yacc`) to produce `pascal.tab.c` and `pascal.tab.h`. The former contains the definition of the parser function *ppparse* which I call after initializing the *pp_stack* in preparation for the first call to *pplex*.

```
⟨ parse Pascal 92 ⟩ ≡                                         (92)
  program = first_token→link; pp_push(program→link, program→next, 0, NULL);
  ppparse( );                                                 Used in 1.
```

The function *ppparse* occasionally builds a parse tree out of internal nodes for the Pascal program; this parse tree is then used to accomplish the transformations needed to turn the Pascal code into C code.

```
⟨ internal node 93 ⟩ ≡                                        (93)
  struct {
    int value;
  }                                                           Used in 6.
```

Internal nodes are constructed using the function *join*.

```
⟨ external declarations 5 ⟩ +≡                                (94)
  token *join(int tag, token *left, token *right, int value);
```

⟨ functions 13 ⟩ +≡ (95)

```
token *join(int tag, token *left, token *right, int value)
{
  token *n = new_token(tag);
  n→previous = left; n→next = right; n→value = value;
  DBG(dbgjoin, "tree:␣"); DBGTREE(dbgjoin, n); return n;
}
```

5 Writing the cweb

5.1 cweb output routines

The basic function to write the cweb file is the function *wprint*, along with its simpler cousins *wput* and *wputs*, and the more specialized member of the family *wputi*. While most of the work of converting the visual representation of tokens to cweb is left to the function *token2string*, the basic functions take care of inserting spaces after a comma and to prevent adjacent tokens from running together.

The variables *alfanum* and *comma* indicate that the last character written was alphanumeric or a comma; the variable *column* counts the characters on the current line.

⟨ global variables 11 ⟩ +≡ (96)

```
static int alfanum = 0;
static int comma = 0;
static int column = 0;
```

⟨ functions 13 ⟩ +≡ (97)

```
static void wput(char c)
{
  fputc(c, w); alfanum = isalnum(c); comma = c ≡ ',';
  if (c ≡ '\n') column = 0; else column++;
}
static void wputs(char *str)
{
  while (*str ≠ 0) wput(*str++);
}
static void wputi(int i)
{
  if (alfanum ∨ comma) fputc(' ', w), column++;
  column += fprintf(w, "%d", i); alfanum = true; comma = false;
}
static void wprint(char *str)
{
  if ((alfanum ∨ comma) ∧ isalnum(str[0])) fputc(' ', w);
  wputs(str);
}
```

Most tokens have their string representation stored in the *info.text* field, so I sketch the function *token2string* here and describe the details of conversion later.

⟨ internal declarations 3 ⟩ +≡ (98)
```
static char *token2string(token *t);
```

⟨ functions 13 ⟩ +≡ (99)
```
static char *token2string(token *t)
{
  CHECK(t ≠ NULL, "Unable to convert NULL token to a string");
  switch (t→tag) {
  default:
    if (t→text ≠ NULL) return t→text;
    else return "";
    ⟨ convert token t to a string 39 ⟩
  }
}
```

5.2 Traversing the WEB

After these preparations, I am ready to traverse the list of tokens again; this time not in Pascal order but in the order given in the WEB file because I want the cweb file to be as close as possible to the original WEB file.

The main loop can be performed by the function *wprint_to*. It traverses the token list until a given *last_token* is found. Using this function I can generate the whole cweb file simply by starting with the *first_token* and terminating with the *last_token*.

⟨ generate cweb output 100 ⟩ ≡ (100)
```
wprint_to(first_token, last_token);
```
Used in 1.

The function *wprint_to* delegates all the work to *wtoken* which in turn uses *wprint* and *token2string* after converting the tokens from WEB to cweb as necessary. Besides writing out the token, *wtoken* also advances past the written token and returns a pointer to the token immediately following it. The function *wtoken* will be called recursively. For debugging purposes, it maintains a counter of its nesting *level*.

⟨ functions 13 ⟩ +≡ (101)
```
static token *wtoken(token *t)
{
  static int level = 0;
  level++; DBG(dbgcweb, "wtoken[%d] %s (%s) line %d\n", level, TAG(t),
      token2string(t), t→lineno);
  switch (t→tag) {
    ⟨ convert t from WEB to cweb 79 ⟩
  default: wprint(token2string(t)); t = t→next; break;
  }
  level--; return t;
}
```

wprint_to is complemented by the function *wskip_to* which suppresses the printing of tokens.

```
⟨ internal declarations 3 ⟩ +≡ (102)
  static token *wprint_to(token *t, token *end);
  static token *wskip_to(token *t, token *end);
```

```
⟨ functions 13 ⟩ +≡ (103)
  static token *wprint_to(token *t, token *end)
  {
    while (t ≠ end) t = wtoken(t);
    return t;
  }
  static token *wskip_to(token *t, token *end)
  {
    while (t ≠ end) t = t→next;
    return t;
  }
```

5.3 Simple cases of conversion

Quite a few tokens serve a syntactical purpose in Pascal but are simply ignored when generating C code.

```
⟨ convert t from WEB to cweb 79 ⟩ +≡ (104)
case CIGNORE: case PPROGRAM: case PLABEL: case PCONST: case PVAR:
  case PPACKED: case POF: case ATQM: case ATBACKSLASH:
  t = t→next; break;
```

The parser will change a *tag* to CIGNORE by using the `IGN` macro.

```
⟨ external declarations 5 ⟩ +≡ (105)
#define IGN(t) ((t)→tag = CIGNORE)
```

TEX uses the control sequence "`@t\2@>`" after "**forward**;". It needs to be removed together with the forward declaration, because it does confuse `cweb`.

```
⟨ convert t from WEB to cweb 79 ⟩ +≡ (106)
case PFORWARD:
  if (t→next→tag ≡ PSEMICOLON) wput(';'), t = t→next→next;
  else wprint("forward"), t = t→next;
  if (t→tag ≡ ATT) t = t→next;
  break;
```

The meta-comments of `WEB` are translated to plain C comments if they are just a single line and to `#if 0...#endif` otherwise.

```
⟨ convert t from WEB to cweb 79 ⟩ +≡ (107)
case METACOMMENT:
  {
    char *c;
    wputs("/*");
    for (c = t→text + 2; c[0] ≠ '@' ∨ c[1] ≠ '}'; c++) wput(*c);
```

```
    wputs("*/"); t = t→next;
  }
  break;
case ATLEFT:
  if (column ≠ 0) wput('\n');
  wputs("#if␣0\n"); t = t→next; break;
case ATRIGHT:
  if (column ≠ 0) wput('\n');
  wputs("#endif\n"); t = t→next; break;
```

Some tokens just need a slight adjustment of their textual representation. In other cases, the parser changes the tag of a token, for example to PSEMICOLON, without changing the textual representation of that token. All these tokens are listed below.

```
⟨ convert t from WEB to cweb 79 ⟩ +≡                                    (108)
case PLEFT: case MLEFT: wputs("␣/*"); t = t→next; break;
case RIGHT: wputs("*/␣"); t = t→next; break;
case PSEMICOLON: wputs(";"); t = t→next; break;
case PCOMMA: wputs(","); t = t→next; break;
case PMOD: wput('%'); t = t→next; break;
case PDIV: wput('/'); t = t→next; break;
case PNIL: wprint("NULL"); t = t→next; break;
case POR: wputs("||"); t = t→next; break;
case PAND: wputs("&&"); t = t→next; break;
case PNOT: wputs("!"); t = t→next; break;
case PIF: wprint("if␣("); t = t→next; break;
case PTHEN: wputs(")␣"); t = t→next; break;
case PASSIGN: wput('='); t = t→next; break;
case PNOTEQ: wputs("!="); t = t→next; break;
case PEQ: wputs("=="); t = t→next; break;
case EQEQ: wput('\t'); t = t→next; break;
case OCTAL: wprint("0"); wputs(t→text + 2); t = t→next; break;
case HEX: wprint("0x"); wputs(t→text + 2); t = t→next; break;
case PTYPEINT: wprint("int"); t = t→next; break;
case PTYPEREAL: wprint("double"); t = t→next; break;
case PTYPEBOOL: wprint("bool"); t = t→next; break;
case PTYPECHAR: wprint("unsigned␣char"); t = t→next; break;
```

I convert "**begin**" to "{". In most cases, I want an "@+" to follow; except of course if a preprocessor directive is following.

```
⟨ convert t from WEB to cweb 79 ⟩ +≡                                    (109)
case PBEGIN: wput('{'), t = t→next;
  if (¬following_directive(t)) wputs("@+");
  break;
```

```
⟨ internal declarations 3 ⟩ +≡                                          (110)
  static bool following_directive(token *t);
```

```
⟨ functions 13 ⟩ +≡                                                    (111)
  static bool following_directive(token *t)
  {
    while (true)
      if (WDEBUG ≤ t→tag ∧ t→tag ≤ WGUBED) return true;
      else if (t→tag ≡ ATPLUS ∨ t→tag ≡ ATEX ∨ t→tag ≡
              ATSEMICOLON ∨ t→tag ≡ NL ∨ t→tag ≡ INDENT) t = t→next;
      else return false;
  }
```

After the conversion, the Pascal token "`..`" will still occur in the file as part of code between vertical bars. To make it print nicely in the TEX output, it is converted to an identifier, "`dotdot`", that is used nowhere else.

```
⟨ convert t from WEB to cweb 79 ⟩ +≡                                   (112)
case PDOTDOT: wprint("dotdot"); t = t→next; break;
```

Using the patch file, I instruct `cweave` to treat this identifier in a special way and print it like "`. .`".

5.4 Pascal division

In some cases build-in functions of Pascal can be replaced by a suitably defined macro in C. The most simple solution was to add these definitions to the module "⟨ Compiler directives ⟩" using the patch file. Using Macros instead of inline replacement has the advantage that the visual appearance of the original code remains undisturbed. A not so simple case is the Pascal division.

The Pascal language has two different division operators: "**div**" divides two integers and gives an integer result; it can be replaced by "/" in the C language. The Pascal operator "/" divides **integer** and **real** values and converts both operands to type **real** before doing so; replacing it simply by the C operator "/" will give different results if both operands are **integer** values because in this case C will do an integer division truncating the result. So expressions of the form "X/Y" should be replaced by "$X/$(**double**)(Y)" to force a floating point division in C

Fortunately, all expressions in the denominator have the form *total_stretch*[*o*], *total_shrink*[*o*], *glue_stretch*(*r*), *glue_shrink*(*r*), or *float_constant*(*n*). So no parentheses around the denominator are required and inserting a simple (**double**) after the / is sufficient. Further, the macro *float_constant* is already a cast to **double**, so I can check for the corresponding identifier and omit the extra cast.

```
⟨ global variables 11 ⟩ +≡                                             (113)
  static int float_constant_no;

⟨ initialize token list 22 ⟩ +≡                                        (114)
  float_constant_no = predefine("float_constant", ID, 0);

⟨ convert t from WEB to cweb 79 ⟩ +≡                                   (115)
case PSLASH: wput('/');
  if (t→next→tag ≠ PMACRO ∨ t→next→sym_no ≠ float_constant_no) {
    wprint("(double)");
    DBG(dbgslash, "Inserting␣(double)␣after␣/␣in␣line␣%d\n", t→lineno);
```

```
  }
  t = t→next; break;
```

5.5 Identifiers

Before I can look at the identifiers, I have to consider the "@!" token which can precede an identifier and will cause the identifiers to appear underlined in the index. The "@!" token needs a special treatment. When I convert Pascal to C, I have to rearrange the order of tokens and while I am doing so, a "@!" token that precedes an identifier should stick to the identifier and move with it. I accomplish this by suppressing the output of the "@!" token if it is followed by an identifier, and insert it again when I output the identifier itself.

```
⟨ convert t from WEB to cweb 79 ⟩ +≡                                    (116)
case ATEX: t = t→next;
  if (t→tag ≠ ID ∧ t→tag ≠ PID ∧ t→tag ≠ PFUNCID ∧
  t→tag ≠ PDEFVARID ∧ t→tag ≠ PDEFPARAMID ∧ t→tag ≠ PDEFTYPEID ∧
  t→tag ≠ OMACRO ∧ t→tag ≠ PMACRO ∧ t→tag ≠ NMACRO ∧
  t→tag ≠ CINTDEF ∧ t→tag ≠ CSTRDEF ∧ t→tag ≠ PDIV ∧
  t→tag ≠ WDEBUG ∧ t→tag ≠ WINIT ∧ t→tag ≠ WSTAT) {
    wputs("@!"); DBG(dbgbasic, "Tag␣after␣@!␣is␣%s␣in␣line␣%d\n",
        tagname(t→tag), t→lineno);
  }
  break;
```

Identifier tokens are converted by using their name. I use a simple function to do the name lookup and take care of adding the "@!" token if necessary.

```
⟨ internal declarations 3 ⟩ +≡                                          (117)
  static void wid(token *t);
```

```
⟨ functions 13 ⟩ +≡                                                     (118)
  void wid(token *t)
  {
    if (alfanum ∨ comma) wput('␣');
    if (t→previous→tag ≡ ATEX) wputs("@!");
    wputs(SYM(t)→name);
  }
```

I use this function like this:

```
⟨ convert t from WEB to cweb 79 ⟩ +≡                                    (119)
case ID: case PID: case OMACRO: case PMACRO: case NMACRO: wid(t);
  t = t→next; break;
```

Some identifiers that TEX uses are reserved words in C or loose their special meaning. So after I finish scanning the WEB, I change the names of these identifiers.

```
⟨ finalize token list 66 ⟩ +≡                                           (120)
  SYM_PTR("xclause")→name = "else";
  SYM_PTR("switch")→name = "get_cur_chr";
  SYM_PTR("continue")→name = "resume";
```

```
SYM_PTR("exit")→name = "end"; SYM_PTR("free")→name = "is_free";
SYM_PTR("int")→name = "i"; SYM_PTR("remainder")→name = "rem";
SYM_PTR("register")→name = "internal_register";
```

A special case is the the field identifier **int**. It can not be used in C because it is a very common (if not the most common) reserved word. I replace it with i which does not conflict with the variable i because field names have their own name-space in C.

5.6 Strings

Pascal strings need some more work. I translate them to characters or C strings. Note that the parser occasionally converts STRING or CHAR tokens to PSTRING tokens.

⟨ convert t from `WEB` to `cweb` 79 ⟩ +≡ (121)

```
case PCHAR:
  {
    char *str = t→text;
    wput('␣'); wput('\''), str++;
    if (str[0] ≡ '\'') wputs("\\'");
    else if (str[0] ≡ '\\') wputs("\\\\");
    else if (str[0] ≡ '@') wputs("@@");
    else wput(str[0]);
    wput('\''); wput('␣');
  }
  t = t→next; break;
case PSTRING:
  {
    char *str = t→text;
    wput('"'), str++;
    while (*str ≠ 0) {
      if (str[0] ≡ '\'' ∧ str[1] ≡ '\'') wput('\''), str++;
      else if (str[0] ≡ '"' ∧ str[1] ≡ '"') wputs("\\\""), str++;
      else if (str[0] ≡ '\\') wputs("\\\\");
      else if (str[0] ≡ '\'' ∧ str[1] ≡ 0) wput('"');
      else if (str[0] ≡ '"' ∧ str[1] ≡ 0) wput('"');
      else if (str[0] ≡ '"') wput('\\'), wput('"');
      else wput(str[0]);
      str++;
    }
  }
  t = t→next; break;
```

5.7 Module names

I have removed newlines and extra spaces from module names; now I have to insert newlines if the module names are too long.

⟨ convert t from WEB to cweb 79 ⟩ +≡ (122)

```
case ATLESS: wputs("@<"); t = t→next;
  CHECK(t→tag ≡ TEXT, "Module␣name␣expected␣instead␣of␣%s␣in␣line␣%d",
      token2string(t), t→lineno);
  {
    char *str = t→text;
    do if (str[0] ≡ '@' ∧ str[1] ≡ ',') str = str + 2;
                          /* control codes are forbidden in section names */
    else if (column > 80 ∧ isspace(*str)) wput('\n'), str++;
    else wput(*str++); while (*str ≠ 0);
  }
  t = t→next;
  if (t→tag ≡ ELIPSIS) wputs("..."), t = t→next;
  CHECK(t→tag ≡ ATGREATER, "@>␣expected␣instead␣of␣%s␣in␣line␣%d",
      token2string(t), t→lineno); wputs("@>"); t = t→next;
  if (t→tag ≡ ATSLASH) wputs("@;"), t = t→next;
  else if (t→tag ≡ PELSE ∨ (t→tag ≡ NL)) wputs("@;");
  break;
```

Note that I replace an "`@/`" after the module name by an "`@;`" Because in most places this is enough to cause the requested new line and causes the correct indentation.

5.8 Replacing the WEB string pool file

WEB strings need more work because I have to replace the WEB string pool file. Before I start, I finish two easy cases. Single character strings are replaced by C character constants. The string pool checksum is simply replaced by zero, because I will not use it.

⟨ convert t from WEB to cweb 79 ⟩ +≡ (123)

```
case CHAR:
  {
    char c = t→text[1];
    wput('\'');
    if (c ≡ '\'' ∨ c ≡ '\\') wput('\\');
    wput(c); wput('\''); t = t→next; break;
  }
case ATDOLLAR: wputs("0"); t = t→next; break;
```

Of course it would be possible to generate suitable initializations for the variables *str_pool* and *str_start* and replace each string with its corresponding index in the *str_start* array. The goal of my project is, however, to generate readable source code and replacing for example `"Maybe␣you␣should␣try␣asking␣a␣human?"` by 283 is not very readable. Instead, I will create for each string a module, in the

above example named ⟨ "Maybe you should try asking a human?" 1234 ⟩, that will expand to the correct number, here 283.

⟨ convert t from WEB to cweb 79 ⟩ +≡ (124)

```
case STRING:
  {
    ⟨ convert some strings to macro names 125 ⟩
    else {
      wputs("@[@<|"); wputs(SYM(t)→name); wputs("|@>@]"); }
    t = t→next; break;
  }
```

There are some exceptions to the general rule, for example for the empty string. I define an appropriate constant *empty_string* and use it instead of a module (which would have a rather unsightly name). More instances of this scheme follow below.

⟨ convert some strings to macro names 125 ⟩ ≡ (125)

```
if (t→sym_no ≡ empty_string_no) wprint("empty_string");
```

Used in 124.

To define all the other new modules, I add some code at the very end of the output file.

⟨ generate cweb output 100 ⟩ +≡ (126)

```
wputs("\n@␣Appendix:␣Replacement␣of␣the␣string␣pool␣file.\n");
{
  token *str_k;
  int i, k;
  ⟨ generate definitions for the first 256 strings 127 ⟩
  for (str_k = first_string; str_k ≠ NULL; str_k = str_k→link)
    ⟨ generate definition for string k 129 ⟩
  ⟨ generate string pool initializations 140 ⟩
}
```

The first 256 strings in the string pool are the printable replacements for the single character strings for all character codes from 0 to 255.

⟨ generate definitions for the first 256 strings 127 ⟩ ≡ (127)

```
wputs("@d␣str_0_255␣");
for (k = 0; k < 256; k++) {
  if ((k & #F) ≡ 0) wputs("\t\"");
  if ((⟨ Character k cannot be printed 128 ⟩)) {
    wputs("^^");
    if (k < °100 ∧ k + °100 ≡ '@') wputs("@@");
    else if (k < °100 ∧ k + °100 ≡ '\\') wputs("\\\\");
    else if (k < °100) wput(k + °100);
    else if (k < °200 ∧ k − °100 ≡ '@') wputs("@@");
    else if (k < °200) wput(k − °100);
#define HEXDIGIT(x) ((x) < 10 ? ((x) + '0') : ((x) − 10 + 'a'))
    else wput(HEXDIGIT(k/16)), wput(HEXDIGIT(k % 16));
  }
  else if (k ≡ '"') wputs("\\\"");
```

```
    else if (k ≡ '\\') wputs("\\\\");
    else if (k ≡ '@') wputs("@@");
    else wput(k);
    if ((k & #F) ≡ #F) wputs("\"@/\n");
  }
  wputs("@d␣str_start_0_255"); i = 0;
  for (k = 0; k < 256; k++) {
    if ((k & #F) ≡ 0) wput('\t');
    wputi(i);
    if ((⟨ Character k cannot be printed 128 ⟩)) {
      if (k < °100) i = i + 3;
      else if (k < °200) i = i + 3;
      else i = i + 4;
    }
    else i = i + 1;
    wput(',');
    if ((k & #F) ≡ #F) wputs("@/\n");
  }
```

Used in 126.

This condition is taken from `tex.web`:

⟨ Character k cannot be printed 128 ⟩ ≡ (128)

```
(k < '␣') ∨ (k > '~')
```

Used in 127.

⟨ generate definition for string k 129 ⟩ ≡ (129)

```
{
  symbol *s = SYM(str_k);
  if (s→value > 0) {
    s→value = 0; wputs("@␣\n");
    wputs("@d␣str_"), wputi(k), wput('␣'), wputs(s→name), wput('\n');
    ⟨ generate macros for some strings 137 ⟩
    else wputs("@<|"), wputs(s→name), wputs("|@>=@+"), wputi(k);
    wput('\n'); k++;
  }
}
```

Used in 126.

There are, however, a few more exceptions to the general procedure. Many of the `WEB` strings are used simply for printing with the procedure *print*. There is actually no need to enter all these strings into the string pool. Instead I add a procedure *print_str* that prints plain zero terminated C strings.

Now I can convert the STRING argument of the procedure *print* to a PSTRING by calling the following procedures in the parser.

⟨ external declarations 5 ⟩ +≡ (130)

```
extern void pstring_args(token *id, token *arg);
extern void pstring_assign(token *id, token *val);
```

The function *pstring_args* is called with the *id* of the function. The *arg* token points to the argument list. A few other functions just pass their arguments to *print*. By replacing their call to *print* by a call to *print_str*, I can convert those

arguments as well. For example the function *overflow* expects two arguments of which the first one is the STRING token. The other functions are: *prompt_file_name*, *print_nl*, and *fatal_error*.

If a STRING token is found, its *value* in the symbol table is decremented. This *value* counts the number of occurrences; if it goes down to zero, the STRING token is no longer used and no module needs to be generated for it.

⟨ functions 13 ⟩ +≡ (131)

```
static int convert_arg(token *arg)
{
  if (arg→tag ≡ STRING) {
    symbol *s = symbol_table[arg→sym_no];
    s→value −−;
    DBG(dbgstring, "Eliminating␣string␣argument␣%s␣(%d)␣in␣line␣%d\n",
        s→name, s→value, arg→lineno); arg→tag = PSTRING; return 1;
  }
  else if (arg→tag ≡ CHAR) {
    arg→tag = PSTRING; return 1;
  }
  return 0;
}
void pstring_args(token *id, token *arg)
{
  if (arg→tag ≡ PCOLON ∨ arg→tag ≡ CREFID) return;
  if (id→sym_no ≡ overflow_no ∨ id→sym_no ≡ prompt_file_name_no) {
    CHECK(arg→tag ≡ PCOMMA,
        "function␣should␣have␣two␣arguments␣in␣line␣%d", id→lineno);
    convert_arg(arg→previous);
  }
  else if (id→sym_no ≡ print_no) {
    if (convert_arg(arg)) id→sym_no = print_str_no;
  }
  else if (id→sym_no ≡ print_str_no∨id→sym_no ≡ print_nl_no∨id→sym_no ≡
        fatal_error_no) convert_arg(arg);
}
```

The function *pstring_assign* is used when STRING tokens are assigned to the variable *help_line*, which I redefine as a variable containing character pointers instead of string numbers.

⟨ functions 13 ⟩ +≡ (132)

```
void pstring_assign(token *id, token *val)
{
  if (id→tag ≡ PID ∧ (id→sym_no ≡ help_line_no ∨ id→sym_no ≡
        max_reg_help_line_no)) {
    SYM(val)→value −−; DBG(dbgstring,
        "Eliminating␣string␣assignment␣%s␣(%d)␣in␣line␣%d\n",
```

```
        SYM(val)→name, SYM(val)→value, val→lineno);
    val→tag = PSTRING;
  }
  else DBG(dbgstring, "No␣string␣assignment␣%s␣(%d)␣in␣line␣%d\n",
        SYM(val)→name, SYM(val)→value, val→lineno);
}
```

Note: *max_reg_help_line* is used in ϵ-TEX.

I have used these variables:

⟨ global variables 11 ⟩ +≡ (133)

```
int print_no, print_str_no, overflow_no, print_err_no, print_nl_no,
    fatal_error_no, prompt_file_name_no, help_line_no, empty_string_no,
    max_reg_help_line_no;
```

The variables are initialized like this:

⟨ functions 13 ⟩ +≡ (134)

```
int predefine(char *name, int tag, int value)
{
  int sym_no = get_sym_no(name);
  symbol *s = symbol_table[sym_no];
  s→tag = tag; s→value = value; return sym_no;
}
```

⟨ initialize token list 22 ⟩ +≡ (135)

```
print_str_no = predefine("print_str", PPROCID, 0);
empty_string_no = predefine("\"\"", PID, 1);
help_line_no = predefine("help_line", ID, 0);
print_no = predefine("print", PPROCID, 0);
overflow_no = predefine("overflow", PPROCID, 0);
print_err_no = predefine("print_err", PPROCID, 0);
print_nl_no = predefine("print_nl", PPROCID, 0);
fatal_error_no = predefine("fatal_error", PPROCID, 0);
prompt_file_name_no = predefine("prompt_file_name", PPROCID, 0);
max_reg_help_line_no = predefine("max_reg_help_line", ID, 0);
```

There are still a few remaining problems. First, STRING tokens occur occasionally as part of a macro replacement text. There I can not substitute a module name for them. By having introduced the function *print_str*, some of them are now plain C strings: `"pool␣size"` in line 1184 (*overflow*), `"!␣"` in line 1750 (*print_nl*) `"save␣size"` in line 5910 (*overflow*), `"input␣stack␣size"` in line 6940 (*overflow*), `"Font␣"` in line 10927 (*print_err*), `"␣at␣"` in line 10930 (*print*), `"pt"` in line 10930 (*print*), `"␣scaled␣"` in line 10933 (*print*), and `"␣plus␣"` in line 19250 (*print*).

Most other macros are numeric macros, and I just generate these instead of module names: `"TeXinputs:"` in line 9992 (*TEX_area*), `"TeXfonts:"` in line 9994 (*TEX_font_area*), `".fmt"` in line 10082 (*format_extension*), the *empty_string* in line 10928, and `"0234000122*4000133**3**344*0400400*000000234000111*\ 1111112341011"` in line 15049 (*math_spacing*).

I have shown already some of the handling of the empty string; the rest follows now:

⟨ convert some strings to macro names 125 ⟩ +≡ (136)
```
else if (t→sym_no ≡ TeXinputs_no) wprint("TEX_area");
  else if (t→sym_no ≡ TeXfonts_no) wprint("TEX_font_area");
  else if (t→sym_no ≡ fmt_no) wprint("format_extension");
  else if (t→sym_no ≡ math_spacing_no) wprint("math_spacing");
```

⟨ generate macros for some strings 137 ⟩ ≡ (137)
```
if (str_k→sym_no ≡ empty_string_no) wputs("@d␣empty_string␣"), wputi(k);
else if (str_k→sym_no ≡ TeXinputs_no) wputs("@d␣TEX_area␣"), wputi(k);
else if (str_k→sym_no ≡ TeXfonts_no)
  wputs("@d␣TEX_font_area␣"), wputi(k);
else if (str_k→sym_no ≡ fmt_no) wputs("@d␣format_extension␣"), wputi(k);
else if (str_k→sym_no ≡ math_spacing_no)
  wputs("@d␣math_spacing␣"), wputi(k);
```
Used in 129.

⟨ global variables 11 ⟩ +≡ (138)
```
int TeXinputs_no, TeXfonts_no, fmt_no, math_spacing_no;
```

⟨ initialize token list 22 ⟩ +≡ (139)
```
TeXinputs_no = predefine("\"TeXinputs:\"", PID, 0);
TeXfonts_no = predefine("\"TeXfonts:\"", PID, 0);
fmt_no = predefine("\".fmt\"", PID, 0);
math_spacing_no = predefine("\"0234000122*4000133**3**344*0400400*00\
    0000234000111*1111112341011\"", PID, 1);
```

I am left with the macro *ensure_dvi_open*, containing `".dvi"` in line 10284, `"file␣name␣for␣output"` in line 10286, and `".dvi"` in line 10286, which I simply turn into a module of the same name using the patch file.

I conclude the generation of `cweb` output by generating initializations for the *str_pool* and *str_start* array.

⟨ generate string pool initializations 140 ⟩ ≡ (140)
```
wputs("\n@␣All␣the␣above␣strings␣together␣make␣up␣the␣string␣pool.\
    \n""@<|str_pool|␣initialization@>=\n""str_0_255\n");
for (i = 256; i < k; i++) {
  wputs("str_"), wputi(i);
  if ((i & 7) ≡ 7) wputs("@/\n"); else wput('␣');
}
wputs("\n\n@␣@<|str_start|␣initialization@>=\n""str_start_0_255\n");
for (i = 256; i < k; i++) {
  wputs("str_start_"), wputi(i), wput(',');
  if ((i & 3) ≡ 3) wput('\n'); else wput('␣');
}
wputs("str_start_"), wputi(k); wputs("\n\n"
"@␣We␣still␣need␣to␣define␣the␣start␣locations␣of␣the␣strings.\n"
"@<prepare␣for␣string␣pool␣initialization@>=\n"
```

```
"typedef␣enum␣{\n"
"str_start_256=sizeof(str_0_255)-1,\n");
for (i = 257; i ≤ k; i++)
  wputs("str_start_"), wputi(i), wputs("=str_start_"), wputi(i − 1),
      wputs("+sizeof(str_"), wputi(i − 1), wputs(")-1,@/\n");
wputs("str_start_end␣}␣str_starts;\n"
"\n@␣@<|pool_ptr|␣initialization@>=␣str_start_"), wputi(k), wputs("\n"
"\n@␣@<|str_ptr|␣initialization@>=␣"), wputi(k), wput('\n');
```

Used in 126.

5.9 Macro and format declarations

When I convert a macro, I first check if the translation has made it obsolete in which case I skip it. Otherwise, I output the initial part of the macro declaration up to the equal sign. From here on, I go different routes for the different types of declarations.

```
⟨ convert t from WEB to cweb 79 ⟩ +≡
case ATD:
  {
    token *eq = t→next→next;
    DBG(dbgcweb, "Macro␣definition␣in␣line␣%d\n", t→lineno);
    if (SYM(t→next)→obsolete) t = wskip_to(t, eq→link);
    else {
      wputs("@d␣"), t = t→next; wprint(SYM(t)→name);
      if (t→tag ≡ NMACRO) ⟨ convert NMACRO from WEB to cweb 143 ⟩
      else if (t→tag ≡ OMACRO) ⟨ convert OMACRO from WEB to cweb 142 ⟩
      else if (t→tag ≡ PMACRO) ⟨ convert PMACRO from WEB to cweb 145 ⟩
      else ERROR("Macro␣name␣expected␣in␣line␣%d", t→lineno);
    }
    DBG(dbgcweb, "End␣Macro␣definition␣in␣line␣%d\n", t→lineno); break;
  }
case ATF:
  {
    token *eq = t→next→next;
    DBG(dbgcweb, "Format␣definition␣in␣line␣%d\n", t→lineno);
    if (SYM(t→next)→obsolete) t = wskip_to(t, eq→link);
    else {
      wputs("@f␣"), t = t→next; wprint(SYM(t)→name);
      t = wprint_to(eq→next, eq→link);
    }
    break;
  }
```

(141)

Ordinary parameterless macros map directly to C style macros.

```
⟨ convert OMACRO from WEB to cweb 142 ⟩ ≡
  {
    wput('\t'); t = eq→next;
```

(142)

```
}
```
Used in 141.

WEB features numeric macros that are evaluated to a numeric value by WEB before they are inserted into the final Pascal program. When converting such macros to C style macros, I have to make sure that a replacement text containing operators is evaluated as one expression. For example when TEX defines *single_base* ≡ *active_base* + 256, where *active_base* ≡ 1, then *print_esc*(*p* − *single_base*) should be evaluated as *print_esc*(*p* − (1 + 256)) not*print_esc*(*p* − 1 + 256). So I add an extra pair of parentheses around the replacement text in case it contains a plus sign or a minus sign.

⟨ convert NMACRO from WEB to cweb 143 ⟩ ≡ (143)
```
{
  int has_operators;
  wput('\t'); has_operators = 0;
  for (t = eq→next; t ≠ eq→link ∧ t→tag ≠ MLEFT ∧ t→tag ≠ NL;
       t = t→next)
    if (t→tag ≡ PPLUS ∨ t→tag ≡ PMINUS) {
      has_operators = 1; break; }
  if (has_operators) wput('(');
  for (t = eq→next; t ≠ eq→link ∧ t→tag ≠ MLEFT ∧ t→tag ≠ NL;
       t = wtoken(t)) continue;
  if (has_operators) wput(')');
}
```
Used in 141.

Parametrized macros in WEB can use any number of arguments. In C, typical parametrized macros have a fixed number of arguments, variadic macros being the exception rather than the rule. Therefore, I count the number of macro arguments each time I expand a macro. Since TEX uses macros with a fixed number of arguments only for 1, 2, or 3 arguments, I use the value 4, for variadic macros.

⟨ count macro parameters 144 ⟩ ≡ (144)
```
{
  token *p;
  int count = 1;
  if (open→next→tag ≡ HASH) {
    DBG(dbgmacro, "Counting␣%s␣parameters␣(#)␣in␣line␣%d\n",
        SYM(t)→name, t→lineno);
  }
  else {
    for (p = open→next; p ≠ open→link; p = p→next)
      if (p→tag ≡ PCOMMA) count++;
      else if (p→tag ≡ POPEN) p = p→link;
    if (SYM(t)→value ≡ 0) SYM(t)→value = count;
    else if (SYM(t)→value ≠ count) SYM(t)→value = 4;
    DBG(dbgmacro, "Counting␣%s␣parameters␣%d␣line␣%d\n",
        SYM(t)→name, SYM(t)→value, t→lineno);
  }
```

```
}
```
Used in 90.

Now that I know the number of arguments, I can construct the macro definition.

⟨ convert PMACRO from WEB to cweb 145 ⟩ ≡ (145)
```
{
  static char *params[4] = {"X", "X", "X, Y", "X, Y, Z"};
                                   /* if I have no information, I assume 1 */
  char *param;
  eq = eq→next;                    /* account for the (#) token */
  if (SYM(t)→value > 3) {
    param = "..."; hash_str = "__VA_ARGS__"; }
  else hash_str = param = params[SYM(t)→value];
  wput('('), wputs(param), wputs(")\t"), t = eq→next;
}
```
Used in 141.

⟨ global variables 11 ⟩ +≡ (146)
```
static char *hash_str;
```

⟨ convert t from WEB to cweb 79 ⟩ +≡ (147)
```
case HASH: wprint(hash_str), t = t→next; break;
```

5.10 Labels

In C, labels are identifiers and labels do not need a declaration. So in the parser, I mark the tokens belonging to a label declaration with the tag CIGNORE and they will be ignored when the cweb file is written.

The tag CLABEL is used now to mark the labels when they are used. In most cases the labels in TEX are numeric macros. So I use the name of the macro as the name of the CLABEL token and mark the definition of the numeric macro as obsolete (Occasionally these label names are modified by adding an integer). In the rare cases where the label is indeed an integer, I use the tag CLABELN. In this case I add the prefix "label" to the numeric value to make it a C identifier. Further, I count the number of times a label is used. Later transformation might render a label as unused and I can remove also the target label. The whole bookkeeping is achieved by calling the function *clabel* at appropriate places in the parser.

⟨ external declarations 5 ⟩ +≡ (148)
```
extern void clabel(token *t, int use);
```

⟨ functions 13 ⟩ +≡ (149)
```
void clabel(token *t, int use)
{
  if (t→tag ≡ NMACRO) {
    SYM(t)→obsolete = true; SYM(t)→value += use; t→tag = CLABEL;
  }
  else if (t→tag ≡ CLABEL) SYM(t)→value += use;
  else if (t→tag ≡ PRETURN) SYM(t)→value += use;
  else {
    if (t→tag ≡ PINTEGER) t→tag = CLABELN;
```

```
      return;
    }
    DBG(dbgstring, "Using␣label␣%s␣(%d)␣in␣line␣%d\n", SYM(t)→name,
        SYM(t)→value, t→lineno);
  }
```

A very special case is the **return** macro of TEX; it is defined as **goto** *exit*. I need to deal with it in a special way, because it usually follows the assignment of a function return value and therefore can be converted to a C **return** statement. In the scanner, I create the PRETURN token and set its symbol number to the *exit* symbol.

⟨ external declarations 5 ⟩ +≡ (150)

```
  extern int exit_no;
#define TOK_RETURN
  {
    token *t = add_token(PRETURN);
    t→sym_no = exit_no;
  }
```

⟨ global variables 11 ⟩ +≡ (151)

```
  int exit_no;
```

⟨ initialize token list 22 ⟩ +≡ (152)

```
  exit_no = get_sym_no("exit");
```

While parsing, I replace the symbol number by the symbol pointer to reflect local *exit* labels.

⟨ special treatment for WEB tokens 73 ⟩ +≡ (153)

```
case PRETURN: t→sym_ptr = SYM(t); pp_stack[pp_sp].next = t→next;
  goto found;
```

The output of the C-style labels is done with the following code. In the case of a CLABEL, I check the use-count in *value* and eliminate unused labels; I also check for a plus sign and a second number (remember labels in Pascal are numeric values) and if found append the number to the label name.

⟨ convert t from WEB to cweb 79 ⟩ +≡ (154)

```
case CLABEL:
  if (t→sym_ptr→value ≤ 0) {
    t = t→next;
    if (t→tag ≡ PPLUS) t = t→next→next;
    if (t→tag ≡ PCOLON) {
      t = t→next;
      if (t→tag ≡ CSEMICOLON) t = t→next;
    }
  }
  else {
    wprint(SYM(t)→name); t = t→next;
    if (t→tag ≡ PPLUS) {
```

```
      t = t→next; wputs(t→text); t = t→next;
    }
  }
  break;
case CLABELN: wprint("label"); wputs(t→text); t = t→next; break;
case PEXIT: wprint("exit(0)"); t = t→next; break;
case PRETURN: wprint("goto␣end"); t = t→next; break;
```

5.11 Constant declarations

In TEX there are only two types of constant declarations: integers and strings. I also observe, that the integer declarations are followed by at most one string declaration. While parsing, I change the tag of the identifier getting defined to CINTDEF or CSTRDEF. I convert the constant declarations into an enumeration type or a **const char** ∗.

```
⟨ convert t from WEB to cweb 79 ⟩ +≡                                    (155)
case CINTDEF: wputs("enum␣{@+"), wid(t), wput('='):
  t = wprint_to(t→link→next, t→link→link); wputs("@+};"); t = t→next;
  break;
case CSTRDEF: wprint("const␣char␣*"), wid(t); t = t→next; break;
```

I have used above a technique that I will use often in the following code. While parsing, I use the link filed of the tokens to connect key tokens of a certain Pascal constructions. Using these links, I can find the different parts (including the intervening WEB tokens) and rearrange them as needed. Linking tokens is achieved with the following macro which also checks that the link stays within the same code sequence.

```
⟨ external declarations 5 ⟩ +≡                                          (156)
#define LNK(from, to) ((from) ? (seq((from), (to)), (from)→link = (to)) : 0)
```

5.12 Variable declarations

When I parse variable declarations, I replace the *tag* of the first variable identifier by PDEFVARID and link all the variables following it together. The last variable is linked to the token separating the identifier from the type, a PCOLON token which the parser has changed to a CIGNORE token. The former PCOLON token itself is then linked to the PSEMICOLON that terminates the variable declaration. In the special case of array variables, I have to insert the variable identifiers inside the type definition. To accomplish this, I set the global variable *varlist* to point to the PDEFVARID token, and continue after printing the type with whatever is left from this list. Note the special precautions taken to get the type of variables right that are used to control **for**-loops; I deal with this problem in section 5.16.

```
⟨ internal declarations 3 ⟩ +≡                                          (157)
  static token ∗varlist;

⟨ global variables 11 ⟩ +≡                                              (158)
  static token ∗varlist = NULL;
```

Using this information I can convert the variable declaration.

```
⟨ convert t from WEB to cweb 79 ⟩ +≡                                    (159)
case PDEFVARID:
  {
    token *type = t→link;
    varlist = t;
    DBG(dbgcweb, "Converting␣variable␣list␣in␣line␣%d\n", t→lineno);
    while (type→tag ≡ PID) type = type→link;
    {
      int replace = 0;
      ⟨ decide whether to replace a subrange type for loop control variables 184 ⟩
      if (replace) {
        wprint("int");
        DBG(dbgfor, "\tReplacing␣subrange␣type␣by␣int\n");
      }
      else wprint_to(type, type→link);
    }
    DBG(dbgcweb, "Finished␣variable␣type␣in␣line␣%d\n", t→lineno);
    if (varlist→tag ≡ PDEFVARID) {
      wid(varlist); varlist = varlist→next; }
    wprint_to(varlist, type); t = type→link;
    DBG(dbgcweb, "Finishing␣variable␣list␣in␣line␣%d\n", t→lineno);
    break;
  }
```

5.13 Types

Pascal type declarations start with the keyword **type**, then follows a list of declarations each one starting with a type identifier. While parsing Pascal, I change the *tag* of the identifier being defined to PDEFTYPEID. I link this token to the first token of the type, and link the first token of the type to the semicolon terminating the type. When I encounter these *tags* now a second time, I can convert them into C **typedef**'s.

```
⟨ convert t from WEB to cweb 79 ⟩ +≡                                    (160)
case PDEFTYPEID:
  {
    token *type_name = t;
    token *type = type_name→link;
    DBG(dbgcweb, "Defining␣type␣%s␣in␣line␣%d\n", token2string(t),
        t→lineno); wprint("typedef␣"); t = wprint_to(type, type→link);
    wprint(token2string(type_name)); break;
  }
```

The above code just uses *wprint_to* to print the type itself. Some types need a little help to print correctly. For instance, subrange types are converted by changing the PEQ token after the new type identifier to a CTSUBRANGE token, with an *up*-link to the parse tree for the subrange. Since C does not have this

kind of subrange types, I approximate them by the standard integer types found in `stdint.h`.

```
⟨ convert t from WEB to cweb 79 ⟩ +≡                                    (161)
case CTSUBRANGE:
  {
    int lo = t→up→previous→value;
    int hi = t→up→next→value;
    DBG(dbgcweb, "Defining␣subrange␣type␣%d..%d\n", lo, hi);
    if (lo < 0 ∧ INT8_MIN ≤ lo ∧ hi ≤ INT8_MAX) wprint("int8_t");
    else if (0 ≤ lo ∧ hi ≤ UINT8_MAX) wprint("uint8_t");
    else if (lo < 0 ∧ INT16_MIN ≤ lo ∧ hi ≤ INT16_MAX) wprint("int16_t");
    else if (0 ≤ lo ∧ hi ≤ UINT16_MAX) wprint("uint16_t");
    else if (lo < 0 ∧ INT32_MIN ≤ lo ∧ hi ≤ INT32_MAX) wprint("int32_t");
    else if (0 ≤ lo ∧ hi ≤ UINT32_MAX) wprint("uint32_t");
    else ERROR("unable␣to␣convert␣subrange␣type␣%d..%d␣in␣line␣%d\n",
        lo, hi, t→lineno);
    t = t→link; break;
  }
```

To set *up*-links in the parser, I use the following macro:

```
⟨ external declarations 5 ⟩ +≡                                          (162)
#define UP(from, to) ((from)→up = (to))
```

Record types get converted into C structures; the variant parts of records become C unions.

```
⟨ convert t from WEB to cweb 79 ⟩ +≡                                    (163)
case PRECORD:
  {
    DBG(dbgcweb, "Converting␣record␣type␣in␣line␣%d\n", t→lineno);
    wprint("struct␣{␣"); t = wprint_to(t→next, t→link);
    DBG(dbgcweb, "Finished␣record␣type␣in␣line␣%d\n", t→lineno);
    wprint("}␣"); break;
  }
case CUNION:
  {
    DBG(dbgcweb, "Converting␣union␣type␣in␣line␣%d\n", t→lineno);
    wprint("union␣{␣"); t = wprint_to(t→next, t→link); wprint("};");
    DBG(dbgcweb, "Finished␣union␣type␣in␣line␣%d\n", t→lineno); break;
  }
```

The conversion of the field declarations of a record type assumes that the Pascal parser has changed the first PID token to a PDEFVARID token and linked it to the following PCOLON token; then linked the PCOLON token to the PSEMICOLON or PEND token that follows the type.

Arrays also need special conversion. Pascal arrays specify a subrange type while C arrays are always zero based and specify a size. Common to both is the specification

of an element type. TEX does not use named array types. Array types only occur in the definition of variables.

I link the PARRAY token to the PSQOPEN token, which I link to either the PDOTDOT token or the type identifier, which I link to the PSQCLOSE token, which I link to the POF token, which is finally linked to the PSEMICOLON following the element type.

The *up* pointer of the PARRAY token points to the parse tree for the subrange of the index type.

```
⟨ convert t from WEB to cweb 79 ⟩ +≡                                    (164)
case PARRAY:
  if (t→up ≡ NULL)
                /* happens for example code which is not part of the program */
    wputs(t→text), t = t→next;
  else {
    token *from = t→link;
    token *index = from→link;
    token *to = index→link;
    token *element_type = to→link;
    token *subrange = t→up;
    int lo, hi, zero_based;
    if (subrange→tag ≡ PID) subrange = subrange→sym_ptr→type;
    lo = subrange→previous→value;
    hi = subrange→next→value; zero_based = (subrange→previous→tag ≡
        PINTEGER ∧ lo ≡ 0) ∨ subrange→previous→tag ≡ PTYPECHAR;
    DBG(dbgarray, "Converting␣array[%d..%d]␣type␣in␣line␣%d\n", lo, hi,
        t→lineno); t = wprint_to(element_type, element_type→link);
    while (true) {
      CHECK(varlist ≠ NULL,
          "Nonempty␣variable␣list␣expected␣in␣line␣%d", varlist→lineno);
      DBG(dbgarray, "Generating␣array␣variable␣%s␣in␣line␣%d\n",
          varlist→sym_ptr→name, varlist→lineno); wid(varlist);
      if (¬zero_based) wput('0');          /* add a zero to the array name */
      wput('['); ⟨ generate array size 165 ⟩ wput(']');
      if (¬zero_based)    /* now I need the array with the appropriate offset */
      {
        DBG(dbgarray,
            "Generating␣array␣pointer␣%s[%s=%d..␣]␣in␣line␣%d\n",
            varlist→sym_ptr→name, token2string(from→next), lo,
            varlist→lineno); wputs(",␣*const␣"); wid(varlist);
        wputs("␣=␣"); wid(varlist), wput('0'); ⟨ generate array offset 166 ⟩;
      }
      varlist = varlist→link;
      if (varlist→tag ≡ PDEFVARID ∨ varlist→tag ≡ PID) wput(',');
      else break;
    }
```

```
    DBG(dbgarray, "Finished␣array␣type␣in␣line␣%d\n", t→lineno);
  }
  break;
```

⟨ generate array size 165 ⟩ ≡ (165)

```
{
  int hi, size;
  hi = generate_constant(subrange→next, 0, 0);
  size = generate_constant(subrange→previous, '-', hi); size = size + 1;
  if (size < 0) wput('-'), wputi(−size);
  else if (size > 0) {
    if (subrange→previous→tag ≠ PTYPECHAR ∧ (subrange→previous→tag ≠
          PINTEGER ∨ subrange→next→tag ≠ PINTEGER)) wput('+');
    wputi(size);
  }
}
```

Used in 164.

⟨ generate array offset 166 ⟩ ≡ (166)

```
{
  int lo = generate_constant(subrange→previous, '-', 0);
  if (lo < 0) wput('-'), wputi(−lo);
  else if (lo > 0) wput('+'), wputi(lo);
}
```

Used in 164.

I use the following function to generate a symbolic expression for the given parse tree representing a constant integer value. The expression contains only plus or minus operators. Parentheses are eliminated using the *sign* parameter. The function returns the numeric value that needs to be printed after all the symbolic constants, accumulating literal constants on its way.

⟨ functions 13 ⟩ +≡ (167)

```
int generate_constant(token *t, char sign, int value)
{
  if (t→tag ≡ PTYPECHAR ∨ t→tag ≡ PINTEGER) {
    if (sign ≡ '-') return value − t→value;
    else return value + t→value;
  }
  else if (t→tag ≡ NMACRO ∨ t→tag ≡ PCONSTID) {
    if (sign ≠ 0) wput(sign);
    wprint(token2string(t→previous)); return value;
  }
  if (t→tag ≡ PPLUS) {
    if (t→previous ≠ NULL)
      value = generate_constant(t→previous, sign, value);
    if (sign ≡ 0) sign = '+';
    return generate_constant(t→next, sign, value);
  }
  if (t→tag ≡ PMINUS) {
```

```
    if (t→previ ous ≠ NULL)
      value = generate_constant(t→previous, sign, value);
    if (sign ≡ 0 ∨ sign ≡ '+') sign = '-';
    else sign = '+';
    return generate_constant(t→next, sign, value);
  }
  else ERROR("Unexpected␣tag␣%s␣while␣generating␣a␣co\
       nstant␣expression␣in␣line␣%d", TAG(t), t→lineno);
}
```

⟨ internal declarations 3 ⟩ +≡ (168)

```
int generate_constant(token *t, char sign, int value);
```

5.14 Files

The Pascal idea of a file, let's say "*fmt_file*: **file of** *memory_word*", is a combination of two things: the file itself and the file's buffer variable capable of holding one data item, in this case one *memory_word*. In C, I can simulate such a Pascal file by a structure containing both: **FILE** $*f$, the file in the C sense; and *memory_word d*, the data item.

⟨ convert t from WEB to cweb 79 ⟩ +≡ (169)

```
case PFILE:
  {
    DBG(dbgcweb, "Converting␣file␣type␣in␣line␣%d\n", t→lineno);
    wprint("struct␣{@+FILE␣*f;@+"); t = wprint_to(t→next, t→link);
    wprint("@,d;@+}␣");
    DBG(dbgcweb, "Finished␣file␣type␣in␣line␣%d\n", t→lineno); break;
  }
```

As I will show in section 5.18, it is also convenient that TEX always passes files, and only files, by reference to functions or procedures. Now I can transcribe *get*(*fmt_file*) into *fread*(&*fmt_file.d*, **sizeof** (*memory_word*), 1, *fmt_file.f*). I put these "rewrite rules" as macros in the patch file; it has the advantage that the rewriting does not disturb the visual appearance of the program code.

Access to the file's buffer variable, in Pascal written as $f\hat{}$ becomes simply $f.d$.

⟨ convert t from WEB to cweb 79 ⟩ +≡ (170)

```
case PUP: wputs(".d"); t = t→next; break;
```

5.15 Structured statements

Some of the structured statements are easy to convert. For example the **if** statement just needs an extra pair of parentheses around the controlling expression. These small adjustment are made when dealing with the PIF and PTHEN token. The **while** statement is similarly simple, but the PDO token may also be part of a **for**-loop. So the parser links the PWHILE token to the PDO token to insert the parentheses.

⟨ convert t from WEB to cweb 79 ⟩ +≡ (171)

```
case PWHILE: wprint("while␣");
```

```
if (t→link ≠ NULL) {
  wput('('); t = wprint_to(t→next, t→link); wputs(")␣");
}
t = t→next; break;
```

Other structured statements need more work.

Let's start with the Pascal **case** statement. Adding parentheses around the controlling expression is not as simple, because I lack a unique second keyword; instead I have a POF token which occurs at various places and is usually ignored. So I link the PCASE token to the corresponding POF token while parsing and generate a **switch** statement.

```
⟨ convert t from WEB to cweb 79 ⟩ +≡                                    (172)
case PCASE:
  if (t→link ≡ NULL) {
    wprint(t→text); t = t→next;
  }
  else {
    wprint("switch␣("); t = wprint_to(t→next, t→link); wputs(")␣{");
  }
  break;
```

The case labels are converted while parsing. While Pascal requires a list of labels followed by a semicolon and a statement, C needs the keyword "**case**" preceding a single label, a colon, and a statement list usually ending with "**break**;". When faced with this problem, I tried a new strategy: inserting new tokens. I insert a CCASE token before each Pascal case label and replace the PCOMMA between labels by a CCOLON. (While it worked quite well, I still wished, I would have solved the problem without modifying the token list).

To insert the CCASE tokens, the parser uses the function *winsert_after*.

```
⟨ external declarations 5 ⟩ +≡                                          (173)
  extern token *winsert_after(token *t, int tag, char *text);

⟨ functions 13 ⟩ +≡                                                     (174)
  token *winsert_after(token *t, int tag, char *text)
  {
    token *n;
    DBG(dbgcweb, "Inserting␣token␣%s␣after␣%s␣in␣line␣%d\n",
        tagname(tag), TAG(t), t→lineno); n = new_token(tag);
    n→next = t→next; n→next→previous = n; n→previous = t; t→next = n;
    n→sequenceno = t→sequenceno; n→lineno = t→lineno; n→text = text;
    return n;
  }
```

Further, the parser replaces the semicolons separating the Pascal case elements by a CBREAK token.

```
⟨ convert t from WEB to cweb 79 ⟩ +≡                                    (175)
case CBREAK:
```

```
if (t→previous→tag ≠ PSEMICOLON ∧ t→previous→tag ≠
        CSEMICOLON ∧ t→previous→tag ≠ PEND) wputs("@;");
if (¬dead_end(t→up, t→lineno)) wprint("@+break;");
t = t→next; break;
```

The semicolon that might be necessary before the "**break**" is inserted using a general procedure described in section 5.17.

TEX often terminates the statement following the case label with a **goto** statement. In this case of course it looks silly to add a **break** statement. I can test this by calling the *dead_end* function

```
⟨ external declarations 5 ⟩ +≡                                    (176)
  int dead_end(token *t, int lineno);

⟨ functions 13 ⟩ +≡                                               (177)
  int dead_end(token *t, int lineno)
  {
    DBG(dbgbreak, "Searching␣for␣dead␣end␣in␣line␣%d:\n", lineno);
    while (true) {
      DBG(dbgbreak, "\t%s\n", TAG(t));
      if (t→tag ≡ PGOTO ∨ t→tag ≡ PEXIT ∨ t→tag ≡ CPROCRETURN)
        return true;
      else if (t→tag ≡ PCOLON) t = t→next;
      else if (t→tag ≡ PBEGIN) t = t→previous;
      else if (t→tag ≡ PSEMICOLON ∨ t→tag ≡ CCASE) {
        if (t→next→tag ≡ CEMPTY) t = t→previous;
        else t = t→next;
      }
      else return false;
    }
  }
```

The "**others**" label can be replaced by "**default**".

```
⟨ convert t from WEB to cweb 79 ⟩ +≡                              (178)
case POTHERS: wprint("default:"); t = t→next; break;
```

I suspect that a *case_list* always ends with either a semicolon or POTHERS without a semicolon. It could be better to generate also a **break** statement at the end of the last case element—especially if the order of cases gets rearranged by rearranging or adding modules.

Finally I convert the **repeat-until** statement. The "**repeat**" becomes "**do** {" and the "**until**" becomes "} **while**". All that is left is to enclose the expression following the "**until**" in a pair of parentheses and add a ¬ operator. The opening parenthesis follows the "**while**"; but where should the closing parenthesis's go? Here I use the fact that in TEX the condition after the "**until**" is either followed directly by a semicolon, or by the start of a new section.

```
⟨ convert t from WEB to cweb 79 ⟩ +≡                              (179)
case PREPEAT: wprint("@/do@+{"); t = t→next; break;
case PUNTIL:
```

```
{
  int sequenceno, lineno;
  token *end;
  wputs("}@+␣while␣(!("); sequenceno = t→sequenceno;
  lineno = t→lineno; end = t→next;
  while (true) {
    if (end→tag ≡ PSEMICOLON ∨ end→tag ≡ CSEMICOLON ∨ end→tag ≡
        PELSE) break;
    else if (end→tag ≡ ATSPACE) {
      while (true) {
        int tag = end→previous→tag;
        if (tag > FIRST_PASCAL_TOKEN ∨ tag ≡ OMACRO ∨ tag ≡
            NMACRO ∨ tag ≡ CHAR) break;
        end = end→previous;
      }
      break;
    }
    end = end→next;
  }
  CHECK(sequenceno ≡ end→sequenceno,
      "until:␣end␣of␣expression␣not␣found␣in␣line␣%d", lineno);
  t = wprint_to(t→next, end); wputs("))"); break;
}
```

5.16 for-loops

To convert the **for** statement, I link the PFOR token to the PTO or PDOWNTO token respectively, which is then linked to the PDO token. The rest seems simple but it hides a surprising difficulty.

⟨ convert t from WEB to cweb 79 ⟩ +≡ (180)

```
case PFOR:
  {
    token *id = t→next;
    token *to = t→link;
    if (to ≡ NULL) {
      wprint("for"); t = t→next; break;
    }
    wprint("for␣("); wprint_to(id, to); wputs(";␣"); wid(id);
    if (to→tag ≡ PTO) wputs("<=");
    else if (to→tag ≡ PDOWNTO) wputs(">=");
    else ERROR("to␣or␣downto␣expected␣in␣line␣%d", to→lineno);
    wprint_to(to→next, to→link); wputs(";␣"); wid(id);
    if (to→tag ≡ PTO) wputs("++");
    else wputs("--");
    wputs(")␣"); t = to→link→next; break;
```

```
}
```

The above code checks that there is actually a link to the PTO token. This link will exist only if the **for**-loop was parsed as part of the Pascal program; it will not exists if the code segment was just part of an explanation (see for example section 823). In this case, I need to deal with the PTO and PDO separately.

Given a Pascal variable "**var** i: 0..255;" the **for**-loop "**for** $i := 255$ **downto** 0 **do**..." will work as expected. If I translate the variable definition to "**uint8_t** i;" the translated **for**-loop "**for** $(i = 255;\ i \geq 0;\ i{-}{-})$..." will not terminate because the loop control variable will never be smaller than 0, instead it will wrap around. If the variable i is used in such a **for**-loop, I should define it simply as "**int** i;".

The first step is the analysis of **for**-loops in the Pascal parser. To do so, I call the function *mark_for_variable* with three parameters: *id*, the loop control variable; *lineno*, the line number for debugging purposes; *value*, the value of the limit terminating the loop; and *direction*, indicating the type of loop. For the *direction*, I distinguish three cases: `TO_LOOP`, `DOWNTO_LOOP`, and loops where the loops limit is a variable (`VAR_LOOP`).

⟨ external declarations 5 ⟩ +≡ (181)

```
#define VAR_LOOP 0
#define TO_LOOP 1
#define DOWNTO_LOOP 2
```

The function then tries to decide whether the type of the for loop control variable should be changed from a subrange type to a plain integer.

If the limit controlling the loop is a variable, I can not ensure (without reasoning about program semantics) that the limit will not coincide with the limit of the subrange type of the control variable. In this case I stay on the safe side and replace the subrange type.

If the limit controlling the loop is a constant, I check its value and replace the type of the control variable only if the value coincides with the upper (or lower) limit of the subrange type used for the control variable. The comparison of the given loop limit with the variables possible range limit is postponed until I generate the variable declaration. For now, I just determine the minimum number of *bits* needed for a suitable variable type.

⟨ functions 13 ⟩ +≡ (182)

```
void mark_for_variable(token *id, int lineno, int value, int direction)
{
  int replace = 0;
  int bits = 0;
  if (direction ≡ VAR_LOOP) replace = 1;
  else if (direction ≡ DOWNTO_LOOP) {
    if (value ≥ 0) bits = 0;              /* lower limit of all unsigned types */
    else if (value > INT8_MIN) bits = 6;
    else if (value > INT16_MIN) bits = 14;
    else bits = 15;
  }
```

```
    else                                                    /* TO_LOOP */
    {
      if (value < 0) bits = 0;
      else if (value < INT8_MAX) bits = 6;
      else if (value < UINT8_MAX) bits = 7;
      else if (value < INT16_MAX) bits = 14;
      else if (value < UINT16_MAX) bits = 15;
      else if (value < INT32_MAX) bits = 31;
      else bits = 32;
    }
    SYM(id)→for_ctrl = FOR_CTRL_PACK(lineno, replace, direction, bits);
  }
```

I pack the result of my analysis into the *for_ctrl* field of the variables symbol table entry using the following macros.

⟨ external declarations 5 ⟩ +≡ (183)

```
  extern void mark_for_variable(token *id, int lineno, int value, int direction);
#define FOR_CTRL_PACK(lineno, replace, direction, bits)
  ((lineno ≪ 16) | ((replace & #1) ≪ 15) | ((direction & #3) ≪ 13) | (bits & #1FFF))
#define FOR_CTRL_LINE(X) (((X) ≫ 16) & #FFFF)
#define FOR_CTRL_REPLACE(X) (((X) ≫ 15) & 1)
#define FOR_CTRL_DIRECTION(X) (((X) ≫ 13) & #3)
#define FOR_CTRL_BITS(X) ((X) & #1FFF)
```

When I finally come to the place where I generate a variable declaration, I can decide whether to replace a subrange type for loop control variables. To do this I iterate over the list of variables and if I find in the list one variable that requires replacement, I change the type of the whole list (this is a bit more than necessary, but it does no harm either).

⟨ decide whether to replace a subrange type for loop control variables 184 ⟩ ≡ (184)

```
  {
    token *subrange = NULL;
    if (type→tag ≡ CTSUBRANGE) subrange = type→up;
    else if (type→tag ≡ CIGNORE ∧ type→next→tag ≡
          PID ∧ type→next→sym_ptr→type ≠
        NULL ∧ type→next→sym_ptr→type→tag ≡ PDOTDOT)
      subrange = type→next→sym_ptr→type;      /* subrange type identifier */
    if (subrange ≠ NULL) {
      token *id = t;
      while (id ≠ type ∧ ¬replace) {
        if (id→sym_ptr→for_ctrl ≠ 0) {
          int lo = subrange→previous→value;
          int hi = subrange→next→value;
          int bits = FOR_CTRL_BITS(id→sym_ptr→for_ctrl);
          int direction = FOR_CTRL_DIRECTION(id→sym_ptr→for_ctrl);
          int lineno = FOR_CTRL_LINE(id→sym_ptr→for_ctrl);
```

```
        replace = FOR_CTRL_REPLACE(id→sym_ptr→for_ctrl);
        DBG(dbgfor, "Subrange␣for␣marked␣variable␣%s␣in␣line␣%d\n",
            token2string(id), id→lineno);
        DBG(dbgfor, "\tRange␣%d␣to␣%d,␣limit␣%d␣bits,␣direct\
            ion␣%d␣in␣line␣%d\n", lo, hi, bits, direction, lineno);
        if (direction ≡ DOWNTO_LOOP) {
          if (lo ≥ 0 ∧ bits ≡ 0) replace = true;
          else if (lo < 0 ∧ INT8_MIN ≤ lo ∧ hi ≤ INT8_MAX ∧ bits ≥ 7)
            replace = true;
          else if (lo < 0 ∧ INT16_MIN ≤ lo ∧ hi ≤ INT16_MAX ∧ bits ≥ 15)
            replace = true;
        }
        else if (direction ≡ TO_LOOP) {
          if (lo < 0 ∧ INT8_MIN ≤ lo ∧ hi ≤ INT8_MAX ∧ bits ≥ 7)
            replace = true;
          else if (0 ≤ lo ∧ hi ≤ UINT8_MAX ∧ bits ≥ 8) replace = true;
          else if (lo < 0 ∧ INT16_MIN ≤ lo ∧ hi ≤ INT16_MAX ∧ bits ≥ 15)
            replace = true;
          else if (0 ≤ lo ∧ hi ≤ UINT16_MAX ∧ bits ≥ 16) replace = true;
        }
      }
      id = id→link;
    }
  }
}
```

Used in 159.

5.17 Semicolons

In C, the semicolon is used to turn an expression, for example an assignment, into a statement; while in Pascal semicolons are used to separate statements in a statement sequence. This difference is important, because C will need in certain cases, for example, preceding an "**else**" or a "`}`" a semicolon, where Pascal must not have one.

The simpler case is the semicolon that in Pascal quite frequently follows an **end**. In C this semicolon often does no harm (it indicates an empty statement), but looks kind of strange, in other cases, for example following a procedure body, it must be eliminated. So I test for it and eliminate it wherever I find it.

```
⟨ convert t from WEB to cweb 79 ⟩ +≡                                (185)
case PEND: wputs("}␣"), t = t→next;
  if (t→tag ≡ PSEMICOLON) t = t→next;
  break;
```

Now let's turn to the more difficult case where C needs a semicolon and Pascal does not have one: preceding an "**else**", at the end of a *case_element*, and at the end of a statement sequence (preceding an "**end**" or "**until**"). Adding a semicolon directly before such an "**else**" would in many cases not look very nice. For instance when the code preceding it is in a different module. The semicolon should instead

follow immediately after the last preceding Pascal token. I insert a CSEMICOLON token just there using the function *wsemicolon*. The function has two parameters: t, the token that might require a preceding semicolon; and p, the pointer to the parse tree preceding the token pointed to by t.

I first check the parse tree whether a semicolon is indeed needed, and if so, I search for the proper place to insert the semicolon. The function *wneeds_semicolon* descends into the parse tree, finds its rightmost statement, and determines whether it needs a semicolon. The function *wback* searches backward to the earliest token that is relevant for the C parser.

The situation is slightly different for `ctangle`. Its pattern matching algorithm does not work good, if the material, for example preceding an else, does not look like a statement, for example because the closing semicolon is hidden in a module or a macro. In these cases it is appropriate to insert a "`@;`" token. I do this by looking at the token preceding the "**else**", skipping over index entries, newlines, indents and such stuff, until finding the end of a module, or macro and insert the "`@;`" there.

⟨ functions 13 ⟩ +≡ (186)

```
bool wneeds_semicolon(token *p)
{
  while (p ≠ NULL) {
    switch (p→tag) {
    case PCASE: case PBEGIN: case CIGNORE: return false;
    case PSEMICOLON: case CCASE: case PELSE: p = p→next; continue;
    case PIF: case PWHILE: case PFOR: case PCOLON:
      p = p→previous; continue;
    case PASSIGN: case PFUNCID: case PCALLID: case PREPEAT:
      case PRETURN: case CRETURN: case CPROCRETURN: case PGOTO:
      case PEXIT: case CEMPTY: default: return true;
    }
  }
  return false;
}
static token *wback(token *t)
{
  while (true) {
    CHECK(t→previous ≠ NULL, "Error␣searching␣backward");
    t = t→previous;
    switch (t→tag) {
    case PSEMICOLON: case CSEMICOLON: case PEND: return t;
    case RIGHT:
      while (t→tag ≠ PLEFT ∧ t→tag ≠ MLEFT) t = t→previous;
      break;
    case ATGREATER: case EQ: case HASH: case ATDOLLAR:
      case NMACRO: case OMACRO: case OCTAL: case HEX: case CHAR:
      case STRING: case PRETURN: case CEMPTY: return t;
```

```
      case CIGNORE: continue;
      default: break;
      }
      if (t→tag > FIRST_PASCAL_TOKEN) return t;
    }
  }
  void wsemicolon(token *p, token *t)
  {
    t = wback(t);
    if (t→tag ≠ PSEMICOLON ∧ t→tag ≠ CSEMICOLON ∧ t→tag ≠ PEND) {
      if (wneeds_semicolon(p)) {
        DBG(dbgsemicolon, "inserting ; in line %d\n", t→lineno);
        if (t→next→tag ≡ ATSEMICOLON) {
          t→next→tag = CSEMICOLON; t→next→text = ";";
        }
        else winsert_after(t, CSEMICOLON, ";");
      }
      else if (t→next→tag ≠ ATSEMICOLON ∧ t→next→tag ≠ PSEMICOLON) {
        DBG(dbgsemicolon, "inserting @; in line %d\n", t→lineno);
        winsert_after(t, ATSEMICOLON, "@;");
      }
    }
  }
```

In procedures, I eliminate a final "*exit*:" because I have replaced "**goto** *exit*" by "**return**".

⟨ functions 13 ⟩ +≡ (187)

```
  void wend(token *p, token *t)
  {
    if (p→tag ≡ PSEMICOLON ∧ p→next→tag ≡
          PCOLON ∧ p→next→next→tag ≡ CEMPTY ∧ p→next→previous→tag ≡
          CLABEL ∧ p→next→previous→sym_no ≡ exit_no) {
      token *label = p→next→previous;
      DBG(dbgreturn, "Trailing exit: found preceding line %d\n",
          t→lineno); label→tag = CIGNORE; SYM(label)→value = −1000;
      CHECK(label→next→tag ≡ PCOLON,
          "Expected colon after label in line %d\n", label→lineno);
      label→next→tag = CIGNORE; p→next→tag = CIGNORE;
    }
    else DBG(dbgreturn, "No trailing exit: found preceding line %d\n",
          t→lineno);
  }
```

⟨ external declarations 5 ⟩ +≡ (188)

```
  extern void wsemicolon(token *p, token *t);
  extern void wend(token *p, token *t);
```

The inserted semicolons have the tag CSEMICOLON. These tokens are printed but—and this is new—hidden from the Pascal parser. This idea might be useful also for other inserted tokens.

⟨ convert token t to a string 39 ⟩ +≡ (189)
```
case CSEMICOLON: return ";";
```

⟨ special treatment for WEB tokens 73 ⟩ +≡ (190)
```
case CSEMICOLON: t = t→next; continue;
```

5.18 Procedures

While parsing, I link the PPROCEDURE token to the PSEMICOLON or POPEN following the procedure name. The PSEMICOLON following the heading is always changed to a CIGNORE.

⟨ convert t from WEB to cweb 79 ⟩ +≡ (191)
```
case PPROCEDURE:
  DBG(dbgcweb, "Converting␣procedure␣heading␣in␣line␣%d\n", t→lineno);
  wprint("void"); t = wprint_to(t→next, t→link);
  if (t→tag ≠ POPEN) wputs("(void)");
  break;
```

The list of parameter identifiers spans from the beginning parenthesis that is pointed to by t to the closing parenthesis pointed to by t→*link*. It is handled similar to a variable declaration. The type identifier, however, needs to be repeated for each parameter in a list. The parser has converted the parameter identifiers to either PDEFPARAMID or PDEFREFID, linked the identifiers together with the final link pointing to the PCOLON preceding the type, and it linked the PCOLON to the PSEMICOLON or PCLOSE following the type. This information is sufficient to convert the parameter list.

⟨ convert t from WEB to cweb 79 ⟩ +≡ (192)
```
case PDEFPARAMID: case PDEFREFID:
  {
    token *varlist = t, *type = t→link;
    DBG(dbgcweb, "Converting␣parameter␣list␣in␣line␣%d\n", t→lineno);
    while (type→tag ≡ PDEFPARAMID ∨ type→tag ≡ PDEFREFID)
      type = type→link;
    while (true) {
      wprint_to(type, type→link);
      if (varlist→tag ≡ PDEFREFID) wputs("␣*");
      wid(varlist); varlist = varlist→link;
      if (varlist ≠ type) wput(',');
      else break;
    }
    t = type→link;
    DBG(dbgcweb, "Finishing␣parameter␣list␣in␣line␣%d\n", t→lineno);
    break;
  }
```

The parser changes the use of a reference variable to a CREFID token, and when I find one now, I dereference it.

⟨ convert t from WEB to cweb 79 ⟩ +≡ (193)
case CREFID: *wputs*("(*"), *wid*(t), *wput*(')'); $t = t\rightarrow next$; **break**;

Now consider a procedure call. The most complex part about it is the argument list. If a procedure has no parameters, there is no argument list in Pascal but there is an empty argument list in C. Further, the use of reference parameters complicates the processing. I need to add a "&" in front of a variable that is passed by reference in C. To accomplish this, the parser constructs for every procedure a *param_mask* and stores it in the *value* field of the procedure identifiers entry in the symbol table. A value of 1 means "no parameter list"; all the other bits correspond from left to right to up to 31 parameters; a bit is set if the corresponding parameter is a reference parameter. I use these definitions:

⟨ external declarations 5 ⟩ +≡ (194)
extern unsigned int *param_mask*, *param_bit*;
#**define** SIGN_BIT (∼(((**unsigned int**) ∼0) ≫ 1))
#**define** START_PARAM (*param_mask* = 0, *param_bit* = SIGN_BIT)
#**define** NEXT_PARAM
(*param_bit* = *param_bit* ≫ 1, CHECK(*param_bit* ≠ 0, "Too␣many␣parameters"))
#**define** REF_PARAM (*param_mask* = *param_mask* | *param_bit*)

⟨ global variables 11 ⟩ +≡ (195)
unsigned int *param_mask*, *param_bit*;

Due to forward declarations, procedure calls can occur before the procedure definition. Therefore I can not apply my knowledge about reference parameters when I parse the procedure call, I have to wait for the second pass, when I convert the WEB to cweb. In TEX the procedure identifier (for example *print*) can be a macro, so the procedure identifier is not necessarily preceding the argument list. Hence I have to process the procedure identifier and the argument list separately.

Let's start with the procedure identifier. When I find it, I check for its *value*, and if the value indicates that there is an empty argument list, I add it.

⟨ convert t from WEB to cweb 79 ⟩ +≡ (196)
case PCALLID: DBG(*dbgcweb*, "Converting␣call␣in␣line␣%d\n", $t\rightarrow lineno$);
wid(t);
if (SYM(t)$\rightarrow value \equiv 1$) *wputs*("()");
$t = t\rightarrow next$; **break**;

At a possibly different place in the WEB file, I will encounter the POPEN token that starts the argument list. It is linked to the corresponding PCLOSE token, and the parser takes care of setting its *up* pointer to the corresponding PCALLID token if there are reference parameters in the argument list.

⟨ convert t from WEB to cweb 79 ⟩ +≡ (197)
case POPEN: *wput*('(');
if ($t\rightarrow up \equiv$ NULL ∨ SYM($t\rightarrow up$)$\rightarrow value \equiv 0$) t = *wprint_to*($t\rightarrow next, t\rightarrow link$);
else {

```
    int param_mask = SYM(t→up)→value;
    token *close = t→link;
    t = t→next;
    if (param_mask < 0) wput('&');
    param_mask = param_mask ≪ 1;
    while (t ≠ close) {
      if (t→tag ≡ PCOMMA) {
        wputs(", "); t = t→next;
        if (param_mask < 0) wput('&');
        param_mask = param_mask ≪ 1;
      }
      else t = wtoken(t);
    }
  }
  break;
```

5.19 Functions

Functions are slightly more complicated than procedures because they feature a return type and a return value. Let's start with the function header. To find the return type, the parser links the end of the parameter list to the colon and the colon to the end of the return type.

⟨ convert *t* from WEB to cweb 79 ⟩ +≡ (198)

```
case PFUNCTION:
  {
    token *param = t→link;
    token *type;
    DBG(dbgcweb, "Converting function heading in line %d\n", t→lineno);
    if (param→tag ≡ POPEN) type = param→link→link;
    else type = param;
    wprint_to(type, type→link); wprint_to(t→next, t→link);
    if (param→tag ≠ POPEN) wputs("(void)");
    else wprint_to(param, param→link→next);
    t = type→link; break;
  }
```

Functions in Pascal return values by assigning them to the function identifier somewhere within the body of the function. In contrast, C uses a return statement, which also terminates the execution of the function immediately. The **return** statement is equivalent to the Pascal assignment only if the assignment is in the tail position of the function. While parsing, I build a tree of the statements. This tree is then searched for assignments to the function identifier in tail positions and these assignments can be converted to **return** statements.

I start with a function that determines whether a part of the parse tree is a "tail", that is it leads directly to the function return.

⟨ functions 13 ⟩ +≡ (199)

```
static bool wtail(token *t)
{
  CHECK(t ≠ NULL,
      "Unexpected␣NULL␣token␣while␣searching␣for␣tail␣statements");
  switch (t→tag) {
  case PSEMICOLON: case PELSE: case CCASE:
    return wtail(t→next) ∧ wtail(t→previous);
  case PCOLON: return wtail(t→next);
  case PRETURN: case CIGNORE: case CEMPTY: return true;
  case PASSIGN: case PCALLID: case PFUNCID: case CRETURN:
    case CPROCRETURN: case PWHILE: case PREPEAT: case PFOR:
    case PEXIT: case PGOTO: return false;
  case PBEGIN: case PIF: case PCASE: return wtail(t→previous);
  default:
    ERROR("Unexpected␣tag␣%s␣while␣searching␣for␣tail␣statements",
        TAG(t));
  }
}
```

The function *wreturn* accomplishes the main task. It is called by the parser, when it has completed the parsing of the function body with parameter t pointing to the parse tree of the entire body. The parameter *tail*, which tells us if the parse tree t is in a tail position, is then set to true. The link parameter, pointing to a possible PRETURN token, is NULL.

⟨ external declarations 5 ⟩ +≡ (200)

```
extern void wreturn(token *t, int tail, token *link);
```

The function *wreturn* calls itself recursively to find and convert all instances where a C return statement is appropriate. If I convert the TEX macro "**return**" to a C **return** statement, I decrement its use-count. If at the end it is zero, I can omit the label *end* marking the end of the function body.

⟨ functions 13 ⟩ +≡ (201)

```
void wreturn(token *t, int tail, token *link)
{
  CHECK(t ≠ NULL, "Unexpected␣NULL␣token␣while␣searching␣f\
      or␣return␣statements");
  switch (t→tag) {
  case PSEMICOLON:
    if (t→next→tag ≡ PRETURN) wreturn(t→previous, true, t→next);
    else {
      wreturn(t→next, tail, link);
      if (wtail(t→next)) wreturn(t→previous, tail, link);
      else wreturn(t→previous, false, NULL);
    }
    return;
  case PCOLON: wreturn(t→next, tail, link); return;
```

```
    case PASSIGN: case PCALLID: case PRETURN: case PEXIT: case PGOTO:
      case CIGNORE: case CEMPTY: return;
    case PWHILE: case PREPEAT: case PFOR:
      wreturn(t→previous, false, NULL); return;
    case PELSE: case CCASE: wreturn(t→next, tail, link);
      wreturn(t→previous, tail, link); return;
    case PCASE: case PIF: case PBEGIN: wreturn(t→previous, tail, link);
      return;
    case PFUNCID:
      if (tail) {
        DBG(dbgreturn, "Converting␣assignment␣to␣function␣in␣line␣%d\n",
            t→lineno); t→tag = CRETURN; IGN(t→next);
        if (link ≠ NULL) {
          link→sym_ptr→value −−; t→sym_ptr = link→sym_ptr;
          IGN(link), IGN(link→next);
          DBG(dbgreturn, "Eliminating␣label␣%s␣(%d)␣in␣line␣%d\n",
              link→sym_ptr→name, link→sym_ptr→value, t→lineno);
        }
      }
      return;
    case CRETURN:      /* this happened when the return; is inside a macro */
      if (t→sym_ptr ≠ NULL) {
        t→sym_ptr→value −−;
        DBG(dbgreturn, "Eliminating␣label␣%s␣(%d)␣in␣line␣%d\n",
            t→sym_ptr→name, t→sym_ptr→value, t→lineno);
      }
      return;
    default: ERROR("Unexpected␣tag␣%s␣in␣line␣%d""␣while␣searching␣fo\
        r␣return␣statements", TAG(t), t→lineno);
    }
  }
```

After these precautions, there are only two functions left: *x_over_n* in line 2273 and *xn_over_d* in line 2306. These need a special local variable matching the function name in the assignment and a trailing **return** statement.

I have two global variables to hold the symbol numbers of the two function names.

```
⟨ external declarations 5 ⟩ +≡                                    (202)
  extern int x_over_n, xn_over_d;

⟨ global variables 11 ⟩ +≡                                        (203)
  int x_over_n, xn_over_d;

⟨ initialize token list 22 ⟩ +≡                                   (204)
  x_over_n = get_sym_no("x_over_n"); xn_over_d = get_sym_no("xn_over_d");
```

While parsing, I check for these two function names and change the initial PBEGIN to an PFBEGIN and the trailing PEND to PFEND, setting the *sym_no* of

these tokens to the symbol number of the function name. Now I can generate the definition of a local variable with the same name as the function (shadowing the function name) at the beginning and a matching return statement at the end.

⟨ convert t from `WEB` to `cweb` 79 ⟩ +≡ (205)

```
case PFBEGIN:
  DBG(dbgcweb, "Adding␣scaled␣%s;␣in␣line␣%d\n", SYM(t)→name, t→lineno);
  wprint("scaled"); wid(t); wputs(";␣"); t = t→next; break;
case PFEND:
  DBG(dbgcweb, "Adding␣return␣%s;␣in␣line␣%d\n", SYM(t)→name, t→lineno);
  wprint("return"); wid(t); wputs(";}"); t = t→next; break;
```

While converting the token list, I check for PFUNCID and CRETURN tokens

⟨ convert t from `WEB` to `cweb` 79 ⟩ +≡ (206)

```
case PFUNCID:
  DBG(dbgcweb, "function␣%s␣in␣line␣%d␣assigns␣result␣variable\n",
      SYM(t)→name, t→lineno); wid(t); t = t→next; break;
case CRETURN:
  DBG(dbgcweb, "Converted␣function␣return␣%s␣in␣line␣%d\n",
      SYM(t)→name, t→lineno); wprint("return"); t = t→next; break;
case CPROCRETURN:
  if (t→sym_ptr→value ≤ 0) wprint("return");
  else wprint("goto␣end");
  t = t→next; break;
```

5.20 The *main* program

While parsing the Pascal program, I change the PBEGIN token starting the main program to a CMAIN token. Now I replace it by the heading of the main program. Similarly I deal with the PEND ending the main program.

⟨ convert t from `WEB` to `cweb` 79 ⟩ +≡ (207)

```
case CMAIN: wprint("int␣main(void)␣{"); t = t→next; break;
case CMAINEND: wprint("return␣0;␣}"); t = t→next; break;
```

6 Predefined symbols in Pascal

I put predefine function- and constant-names of Pascal into the symbol table. I omit predefined symbols that are not used in TEX.

⟨ initialize token list 22 ⟩ +≡ (208)

```
predefine("put", PPROCID, 0); predefine("get", PPROCID, 0);
predefine("reset", PPROCID, 0); predefine("rewrite", PPROCID, 0);
predefine("abs", PFUNCID, 0); predefine("odd", PFUNCID, 0);
predefine("eof", PFUNCID, 0); predefine("eoln", PFUNCID, 0);
predefine("round", PFUNCID, 0); predefine("ord", PFUNCID, 0);
predefine("chr", PFUNCID, 0); predefine("close", PPROCID, 0);
predefine("read", PPROCID, 0); predefine("read_ln", PPROCID, 0);
predefine("write", PPROCID, 0); predefine("write_ln", PPROCID, 0);
predefine("break", PPROCID, 0); predefine("break_in", PPROCID, 0);
predefine("erstat", PFUNCID, 0); predefine("false", PCONSTID, 0);
predefine("true", PCONSTID, 1);
```

7 Processing the command line

The *usage* function explains command line parameters and options.

⟨ functions 13 ⟩ +≡ (209)

```
void usage(void)
{
  fprintf(stderr, "Usage: web2w [parameters] filename.web\n"
  "Parameters:\n"
  "\t -p    \t generate a pascal output file\n"
  "\t -o file    \t specify an output file name\n"
  "\t -l    \t redirect stderr to a log file\n"
  "\t -y    \t generate a trace while parsing pascal\n"
  "\t -d XX\t hexadecimal debug value. OR together these values:\n"
  "\t \t    \t XX=1    basic debugging\n"
  "\t \t    \t XX=2    flex debugging\n"
  "\t \t    \t XX=4    link debugging\n"
  "\t \t    \t XX=8    token debugging\n"
  "\t \t    \t XX=10   identifier debugging\n"
  "\t \t    \t XX=20   pascal tokens debugging\n"
  "\t \t    \t XX=40   expansion debugging\n"
  "\t \t    \t XX=80   bison debugging\n"
  "\t \t    \t XX=100 pascal parser debugging\n"
  "\t \t    \t XX=200 cweb debugging\n"
  "\t \t    \t XX=400 join debugging\n"
  "\t \t    \t XX=800 string pool debugging\n"
  "\t \t    \t XX=1000 for variables debugging\n"
  "\t \t    \t XX=2000 for real division debugging\n"
  "\t \t    \t XX=4000 for macro debugging\n"
  "\t \t    \t XX=8000 for array debugging\n"
  "\t \t    \t XX=10000 for return debugging\n"
  "\t \t    \t XX=20000 for semicolon debugging\n"
  "\t \t    \t XX=40000 for break debugging\n"
  ); exit(1);
}
```

The different debug values are taken from an enumeration type.

⟨ external declarations 5 ⟩ +≡ (210)

```
typedef enum {
```

dbgnone = #0, *dbgbasic* = #1, *dbgflex* = #2, *dbglink* = #4, *dbgtoken* = #8,
dbgid = #10, *dbgpascal* = #20, *dbgexpand* = #40, *dbgbison* = #80,
dbgparse = #100, *dbgcweb* = #200, *dbgjoin* = #400, *dbgstring* = #800,
dbgfor = #1000, *dbgslash* = #2000, *dbgmacro* = #4000, *dbgarray* = #8000,
dbgreturn = #10000, *dbgsemicolon* = #20000, *dbgbreak* = #40000
} **debugmode**;

Processing the command line looks for options and then sets the basename.

⟨ external declarations 5 ⟩ +≡ (211)
```
extern FILE *logfile;
extern int ww_flex_debug;
extern debugmode debugflags;
```

⟨ global variables 11 ⟩ +≡ (212)
```
#define MAX_NAME 256
static char basename[MAX_NAME];
static FILE *w = NULL;
static FILE *pascal = NULL;
FILE *logfile = NULL;
debugmode debugflags = dbgnone;
```

⟨ process the command line 213 ⟩ ≡ (213)
```
{
  int mk_logfile = 0, mk_pascal = 0, baselength = 0;
  char *w_file_name = NULL;
  ww_flex_debug = 0; ppdebug = 0;
  if (argc < 2) usage( );
  argv++;                                   /* skip the program name */
  while (*argv ≠ NULL) {
    if ((*argv)[0] ≡ '-') {
      char option = (*argv)[1];
      switch (option) {
      default: usage( );
      case 'p': mk_pascal = 1; break;
      case 'o': argv++; w_file_name = *argv; break;
      case 'l': mk_logfile = 1; break;
      case 'y': ppdebug = 1; break;
      case 'd':
        argv++;
        if (*argv ≡ NULL) usage( );
        debugflags = strtol(*argv, NULL, 16);
        if (debugflags & dbgflex) ww_flex_debug = 1;
        if (debugflags & dbgbison) ppdebug = 1;
        break;
      }
    }
    else {
```

```
        strncpy(basename, *argv, MAX_NAME − 1);
        baselength = strlen(basename) − 4;
        if (baselength < 1 ∨ strncmp(basename + baselength, ".web", 4) ≠ 0)
          usage( );
        basename[baselength] = 0;
        if (*(argv + 1) ≠ NULL) usage( );
      }
      argv++;
    }
    ⟨ open the files 214 ⟩
  }
```

Used in 1.

After the command line has been processed, four file streams need to be opened: *win*, the input file; *w*, the output file; *logfile*, if a log file is asked for; and *pascal*, if the output of the pascal code is requested. For technical reasons, the scanner generated by `flex` needs an output file *wwout*. The log file is opened first because this is the place where error messages should go while the other files are opened.

```
⟨ open the files 214 ⟩ ≡                                              (214)
  if (mk_logfile) {
    basename[baselength] = 0; strcat(basename, ".log");
    logfile = freopen(basename, "w", stderr);
    if (logfile ≡ NULL) {
      fprintf(stderr, "Unable to open logfile %s", basename);
      logfile = stderr;
    }
    wwout = logfile;
  }
  else {
    logfile = stderr; wwout = stderr;
  }
  basename[baselength] = 0; strcat(basename, ".web");
  wwin = fopen(basename, "r");
  if (wwin ≡ NULL) ERROR("Unable to open input file %s", basename);
  if (w_file_name ≡ NULL) {
    w_file_name = basename; basename[baselength] = 0;
    strcat(basename, ".w");
  }
  w = fopen(w_file_name, "w");
  if (w ≡ NULL) ERROR("Unable to open output file %s", w_file_name);
  if (mk_pascal) {
    basename[baselength] = 0; strcat(basename, ".pas");
    pascal = fopen(basename, "w");
    if (pascal ≡ NULL) ERROR("Unable to open pascal file %s", basename);
  }
```

Used in 213.

8 Error handling and debugging

There is no good program without good error handling. To print messages or indicate errors I define the following macros:

⟨ external declarations 5 ⟩ +≡ (215)

```
#include <stdlib.h>
#include <stdio.h>
#define MESSAGE(...) (fprintf(logfile, __VA_ARGS__), fflush(logfile))
#define ERROR(...) (fprintf(logfile, "ERROR: "), MESSAGE(__VA_ARGS__),
    fprintf(logfile, "\n"), exit(1))
#define CHECK(condition, ...) (¬(condition) ? ERROR(__VA_ARGS__) : 0)
```

To display the content of a token I can use `THE_TOKEN`.

⟨ external declarations 5 ⟩ +≡ (216)

```
#define THE_TOKEN(t) "%d\t%d: %s\t[%s]\n", t→lineno, t→sequenceno,
    token2string(t), tagname(t→tag)
```

The amount of debugging depends on the debugging flags.

⟨ external declarations 5 ⟩ +≡ (217)

```
#define DBG(flags, ...)
  { if (debugflags & flags) MESSAGE(__VA_ARGS__); }
#define DBGTOKS(flags, from, to)
  {
    if (debugflags & flags) {
      token *t = from;
      MESSAGE("<<");
      while (t ≠ to) {
        MESSAGE("%s", token2string(t)); t = t→next;
      }
      MESSAGE(">>\n");
    }
  }
#define TAG(t) (t ? tagname(t→tag) : "NULL")
#define DBGTREE(flags, t)DBG (flags, "%s -> %s | %s | %d\n", TAG(t),
    TAG(t→previous), TAG(t→next), t→value)
```

9 The scanner

```
%{
#include "web2w.h"
#include "pascal.tab.h"
%}

%option prefix="ww"
%option noyywrap yylineno nounput noinput batch
%option debug

%x PASCAL MIDDLE DEFINITION FORMAT NAME

CONTROL         [^@\n]*
ID              [a-zA-Z][a-zA-Z0-9_]*
SP              [[:blank:]]*
STARSECTION     @\*{SP}(\\\[[0-9a-z]+\])?
SPACESECTION    @[[:space:]]{SP}
REAL            [0-9]+(\.[0-9]+(E[+-]?[0-9]+)?|E[+-]?[0-9]+)
DDD             {SP}\.\.\.{SP}
%%
                        /* WEB codes, see WEB User Manual page 7 ff*/
<INITIAL>{
{SPACESECTION}          EOS;TOK("@ ",ATSPACE);BOS;
{STARSECTION}           EOS;TOK("@*",ATSTAR); BOS;
@[dD]                   EOS;TOK("@d ",ATD);BEGIN(DEFINITION);
@[fF]                   EOS;TOK("@f ",ATF);BEGIN(FORMAT);
@[pP]                   EOS;TOK("@p",ATP);PROGRAM;PUSH;SEQ;BEGIN(PASCAL);
@\<{SP}                 EOS;TOK("@<",ATLESS);PUSH;BOS;BEGIN(NAME);

\{                      ADD;PUSH_NULL;
\}                      POP_LEFT;

\|                      EOS;TOK("|",BAR);PUSH;BEGIN(PASCAL);

@'[0-7]+                EOS;TOK(COPY,OCTAL);BOS;
@\"[0-9a-fA-F]+         EOS;TOK(COPY,HEX);BOS;
@\^{CONTROL}@\>         EOS;TOK(COPY,ATINDEX); BOS;
@\.{CONTROL}@\>         EOS;TOK(COPY,ATINDEXTT); BOS;
@\:{CONTROL}@\>         EOS;TOK(COPY,ATINDEX9); BOS;
@!                      EOS;TOK("@!",ATEX); BOS;
```

```
@\?                     EOS;TOK("@?",ATQM); BOS;
@@                      EOS;TOK("@@",ATAT);BOS;

\n                      ADD;
([^\\%@|{}.\n])*        ADD; /* we do not analyze TEX parts any further */
\\[\\%@|{}]             ADD;
\\                      ADD;
\%.*                    ADD;
.                       ADD;

<<EOF>>                 EOS;TOK("",WEBEOF);return 0;
}

<MIDDLE>{
{SPACESECTION}          TOK("@ ",ATSPACE);POP;BOS;BEGIN(TEX);
{STARSECTION}           TOK("@*",ATSTAR); POP;BOS;BEGIN(TEX);
@[dD]                   TOK("@d ",ATD);POP;BEGIN(DEFINITION);
@[fF]                   TOK("@f ",ATF);POP;BEGIN(FORMAT);
@[pP]                   TOK("@p",ATP);POP;PROGRAM;PUSH;SEQ;BEGIN(PASCAL);
@\<{SP}                 TOK("@<",ATLESS);POP;PUSH;BOS;BEGIN(NAME);
\{                      TOK(" {",MLEFT);PUSH;BEGIN(TEX);BOS;
}

<DEFINITION>{
{ID}                    SYMBOL;
\(#\)                   TOK("(#)",PARAM);
=                       TOK("=",EQEQ);PUSH;DEF_MACRO(NMACRO);BEGIN(MIDDLE);
==                      TOK("==",EQEQ);PUSH;DEF_MACRO(OMACRO);BEGIN(MIDDLE);
[[:space:]]             ;
}

<FORMAT>{
begin                   TOK("if",PIF);
end                     TOK("if",PIF);
{ID}                    SYMBOL;
==                      TOK("==",EQEQ);PUSH;
\{                      TOK(" {",MLEFT);PUSH;BEGIN(TEX);BOS;
\n                      TOK("\n",NL);BEGIN(MIDDLE);
[[:space:]]             ;
}

<NAME>{
{SP}@\>                 EOS;AT_GREATER;BEGIN(PASCAL);
{DDD}@\>                EOS;TOK("...",ELIPSIS);AT_GREATER;BEGIN(PASCAL);
{SP}@\>{SP}=            EOS;AT_GREATER_EQ;BEGIN(PASCAL);
{DDD}@\>{SP}=           EOS;TOK("...",ELIPSIS);AT_GREATER_EQ;BEGIN(PASCAL);
[[:space:]]+            add_string(" ");
.                       ADD;
}
```

```
<PASCAL>{
{SPACESECTION}        TOK("@ ",ATSPACE);POP;BOS;BEGIN(TEX);
{STARSECTION}         TOK("@*",ATSTAR);POP;BOS;BEGIN(TEX);
@\<{SP}               TOK("@<",ATLESS);PUSH;BOS;BEGIN(NAME);
\{                    TOK(" {",PLEFT);PUSH;BEGIN(TEX);BOS;
}

<MIDDLE,PASCAL>{
<<EOF>>               TOK("",WEBEOF);POP;return 0;

@'[0-7]+              TOK(COPY,OCTAL);
@\"[0-9a-fA-F]+       TOK(COPY,HEX);
@!                    TOK("@!",ATEX);
@\?                   TOK("@?",ATQM);
\|                    TOK("|",BAR);POP;BEGIN(TEX);BOS;
@t{CONTROL}@\>        TOK(COPY,ATT);
@={CONTROL}@\>        TOK(COPY,ATEQ);

\}                    ERROR("Unexpected }");
\(                    TOK("(",POPEN);PUSH;
\)                    TOK(")",PCLOSE);POP;
#                     TOK("#",HASH);/* used in macros */

                      /* non Pascal tokens */
\n                    TOK("\n",NL);
@\^{CONTROL}@\>       TOK(COPY,ATINDEX);
@\.{CONTROL}@\>       TOK(COPY,ATINDEXTT);
@\:{CONTROL}@\>       TOK(COPY,ATINDEX9);
@\$                   TOK("@$",ATDOLLAR);
@\{                   TOK("@{",ATLEFT);
@\}                   TOK("@}",ATRIGHT);
@\{[^\n]*@\}          TOK(COPY,METACOMMENT);
@\&                   TOK("@&",ATAND);
@\\                   TOK("@\\",ATBACKSLASH);
@,                    TOK("@,",ATCOMMA);
@\/                   TOK("@/",ATSLASH);
@\|                   TOK("@|",ATBAR);
@\#                   TOK("@#",ATHASH);
@\+                   TOK("@+",ATPLUS);
@\;                   TOK("@;",ATSEMICOLON);

                      /* Pascal tokens */
=                     TOK("=",PEQ);
\+                    TOK("+",PPLUS);
\-                    TOK("-",PMINUS);
\*                    TOK("*",PSTAR);
\/                    TOK("/",PSLASH);
\<\>                  TOK(" <> ",PNOTEQ);
\<                    TOK(" < ",PLESS);
```

```
\>                  TOK(" > ",PGREATER);
\<=                 TOK(" <= ",PLESSEQ);
\>=                 TOK(" >= ",PGREATEREQ);
\[                  TOK("[",PSQOPEN);
\]                  TOK("]",PSQCLOSE);
:=                  TOK(":=",PASSIGN);
\.                  TOK(".",PDOT);
\.\.                TOK("..",PDOTDOT);
,                   TOK(",",PCOMMA);
;                   TOK(";",PSEMICOLON);
:                   TOK(":  ",PCOLON);
\^                  TOK("^",PUP);

                    /* special coding trick in line 676 of tex.web */
t@&y@&p@&e          TOK("type",PTYPE);

                    /* pascal keywords */
"mod"               TOK("mod",PMOD);
"div"               TOK("div",PDIV);
"nil"               TOK("nil",PNIL);
"in"                TOK("in",PIN);
"or"                TOK("or",POR);
"and"               TOK("and",PAND);
"not"               TOK("not",PNOT);
"if"                TOK("if",PIF);
"then"              TOK("then",PTHEN);
"else"              TOK("else",PELSE);
"case"              TOK("case",PCASE);
"of"                TOK("of",POF);
"others"            TOK("others",POTHERS);
"forward"           TOK("forward",PFORWARD);
"repeat"            TOK("repeat",PREPEAT);
"until"             TOK("until",PUNTIL);
"while"             TOK("while",PWHILE);
"do"                TOK("do",PDO);
"for"               TOK("for",PFOR);
"to"                TOK("to",PTO);
"downto"            TOK("downto",PDOWNTO);
"begin"             TOK("begin",PBEGIN);
"end"               TOK("end",PEND);
"with"              TOK("with",PWITH);
"goto"              TOK("goto",PGOTO);
"const"             TOK("const",PCONST);
"var"               TOK("var",PVAR);
"array"             TOK("array",PARRAY);
"record"            TOK("record",PRECORD);
"set"               TOK("set",PSET);
```

```
"file"                TOK("file",PFILE);
"function"            TOK("function",PFUNCTION);
"procedure"           TOK("procedure",PPROCEDURE);
"label"               TOK("label",PLABEL);
"packed"              TOK("packed",PPACKED);
"program"             TOK("program",PPROGRAM);
"char"                TOK("char",PTYPECHAR);
"integer"             TOK("integer",PTYPEINT);
"real"                TOK("real",PTYPEREAL);
"boolean"             TOK("boolean",PTYPEBOOL);

"endcases"            TOK("endcases",PEND);
"othercases"          TOK("othercases",POTHERS);
"mtype"               TOK("type",PTYPE);
"final_end"           TOK("final_end",PEXIT);

"return"              TOK_RETURN;

"debug"               TOK("debug",WDEBUG);
"gubed"               TOK("debug",WGUBED);
"stat"                TOK("stat",WSTAT);
"tats"                TOK("tats",WTATS);
"init"                TOK("init",WINIT);
"tini"                TOK("tini",WTINI);

{ID}                  SYMBOL;
\"([^"\n]|\"\")\"     TOK(COPY,CHAR);  /* single character string */
\"([^"\n]|\"\")*\"    WWSTRING; /* multiple character string */
\'([^'\n]|\'\'|@@)\'  TOK(COPY,PCHAR);
\'([^'\n]|\'\')*\'    TOK(COPY,PSTRING);
[0-9]+                TOK(COPY,PINTEGER);
{REAL}                TOK(COPY,PREAL);

^[[:space:]]+         TOK(COPY,INDENT);
[[:space:]]           ; /* in Pascal mode we ignore spaces */

}

                      /* anything that gets to this line
                      is an illegal character */
<*>.                  { ERROR("Illegal %c (0x%02x) in line %d mode %d",
                      yytext[0],yytext[0],yylineno, YY_START);}

%%
```

10 The parser

The following code is contained in the file `pascal.y`. It represents a modified grammar for the Pascal language. Here and throughout of this document, terminal symbols, or tokens, are shown using a small caps font; for nonterminal symbols I use a slanted font.

```
%{
#include <stdio.h>
#include "web2w.h"

/* the tag=token number of the left hand side symbol of a rule */
#define LHSS (yyr1[yyn]+FIRST_PASCAL_TOKEN-3)

static int function=0;

%}

%code requires {
#define PPSTYPE token *
#define YYSTYPE PPSTYPE

extern int ppparse(void);
extern int ppdebug;
}

%token-table
%defines
%error_verbose
%debug
%define api.prefix "pp"
%expect 1

%token PEOF 0 "end of file"
%token WEBEOF "end of web"
%token HEAD
%token BAR
%token PLEFT
%token MLEFT
%token RIGHT
%token OPEN
%token CLOSE
```

```
%token TEXT
%token NL
%token HASH
%token NMACRO
%token OMACRO
%token PMACRO
%token PARAM
%token EQ
%token EQEQ
%token ATSTAR
%token ATSPACE
%token ATD
%token ATF
%token ATLESS
%token ATGREATER
%token ELIPSIS
%token ATP
%token OCTAL
%token HEX
%token ATAT
%token ATDOLLAR
%token ATLEFT
%token ATRIGHT
%token ATINDEX
%token ATINDEXTT
%token ATINDEX9
%token ATT
%token ATEQ
%token ATAND
%token ATBACKSLASH
%token ATEX
%token ATQM
%token ATCOMMA
%token ATSLASH
%token ATBAR
%token ATHASH
%token ATPLUS
%token ATSEMICOLON
%token STRING
%token CHAR
%token INDENT
%token METACOMMENT
%token CSEMICOLON
%token ID

%token WDEBUG
```

```
%token WSTAT
%token WINIT
%token WTINI
%token WTATS
%token WGUBED

%token PRETURN "return"

%token FIRST_PASCAL_TOKEN

%token PPLUS "+"
%token PMINUS "-"
%token PSTAR "*"
%token PSLASH "/"
%token PEQ "="
%token PNOTEQ "<>"
%token PLESS "<"
%token PGREATER ">"
%token PLESSEQ "<="
%token PGREATEREQ ">="
%token POPEN "("
%token PCLOSE ")"
%token PSQOPEN "["
%token PSQCLOSE "]"
%token PASSIGN ":="
%token PDOT "."
%token PCOMMA ","
%token PSEMICOLON ";"
%token PMOD "mod"
%token PDIV "div"
%token PNIL "nil"
%token POR "or"
%token PAND "and"
%token PNOT "not"
%token PIF "if"
%token PTHEN "then"
%token PELSE "else"
%token PREPEAT "repeat"
%token PUNTIL "until"
%token PWHILE "while"
%token PDO "do"
%token PFOR "for"
%token PTO "to"
%token PDOWNTO "downto"
%token PBEGIN "begin"
%token PEND "end"
%token PGOTO "goto"
```

```
%token PINTEGER "0-9"
%token PREAL "real"
%token POTHERS "others"
%token PSTRING "'...'"
%token PCHAR "'.'"
%token PTYPECHAR "char type"
%token PTYPEBOOL "bool type"
%token PTYPEINT "integer type"
%token PTYPEREAL "real type"
%token PTYPEINDEX "index type"

%token PID "identifier"
%token PDEFVARID "variable definition"
%token PDEFPARAMID "parameter definition"
%token PDEFREFID "reference parameter definition"
%token PCONSTID "constant"
%token PDEFCONSTID "constant definition"
%token PDEFTYPEID "typename definition"
%token PDEFTYPESUBID "subrange typename definition"
%token PARRAYFILETYPEID "array of file type"
%token PARRAYFILEID "array of file name"
%token PFUNCID "functionname"
%token PDEFFUNCID "functionname definition"
%token PPROCID "procedurename"
%token PCALLID "call"
%token PRETURNID "return value"

%token PEXIT "final_end"
%token PFBEGIN "function begin"
%token PFEND "function end"
%token PDOTDOT ".."
%token PCOLON ":"
%token PUP "^"
%token PIN "in"
%token PCASE "case"
%token POF "of"
%token PWITH "with"
%token PCONST "const"
%token PVAR "var"
%token PTYPE "type"
%token PARRAY "array"
%token PRECORD "record"
%token PSET "set"
%token PFILE "file"
%token PFUNCTION "function"
%token PPROCEDURE "procedure"
%token PLABEL "label"
```

```
%token PPACKED "packed"
%token PPROGRAM "program"
%token PFORWARD "forward"

%token CIGNORE
%token CLABEL
%token CLABELN
%token CINTDEF
%token CSTRDEF
%token CMAIN
%token CMAINEND
%token CUNION
%token CTSUBRANGE
%token CINT
%token CREFID "reference variable"
%token CRETURN "C function return"
%token CPROCRETURN "C procedure return"
%token CCASE "C case"
%token CCOLON "C :"
%token CBREAK "break"
%token CEMPTY "empty statement"
%%
program : programheading globals
          PBEGIN statements PEND PDOT
          { CHGTAG($3,CMAIN); CHGTAG($5,CMAINEND); IGN($6);
            wsemicolon($4,$5);
          }
        ;

programheading : PPROGRAM PID PSEMICOLON { IGN($2); IGN($3); }
               ;

globals : labels constants types variables procedures
        ;

labels :
       | PLABEL labellist PSEMICOLON { IGN($3); }
       ;

labellist : labeldecl
          | labellist PCOMMA labeldecl { IGN($2); }
          ;

labeldecl : NMACRO { IGN($1); SYM($1)->obsolete=1; }
          | PINTEGER { IGN($1); }
          | PEXIT { IGN($1); }
          | labeldecl PPLUS PINTEGER { IGN($2); IGN($3); }
          ;
```

```
constants :
          | PCONST constdefinitions
          | PCONST constdefinitions conststringdefinition
          ;

constdefinitions : constdefinition
                 | constdefinitions constdefinition
                 ;

constdefinition : PID PEQ PINTEGER PSEMICOLON { LNK($1,$2); LNK($2,$4);
                    SETVAL($1,getval($3)); CHGID($1,PCONSTID);
                    CHGTAG($1,CINTDEF); }
                ;

conststringdefinition : PID PEQ PSTRING PSEMICOLON
                      { seq($1,$4); CHGID($1,PCONSTID);
                        CHGTAG($1,CSTRDEF);CHGTAG($2,PASSIGN); }
                      ;

types :
      | PTYPE typedefinitions { IGN($1); }
      ;

typedefinitions : typedefinition
                | typedefinitions typedefinition
                ;

typedefinition : PID PEQ subrange PSEMICOLON
                { DBG(dbgparse,"New Subrange Type: %s\n",
                  SYM($1)->name);
                  LNK($1,$2); IGN($2);LNK($2,$4);
                  CHGTYPE($1,$3);
                  CHGTAG($1,PDEFTYPEID);
                  CHGTAG($2,CTSUBRANGE); UP($2,$3);
                }
               | PID PEQ type PSEMICOLON
                { DBG(dbgparse,"New Type: %s\n",
                  SYM($1)->name);
                  LNK($1,$2); IGN($2); LNK($2,$4);
                  CHGTYPE($1,$3); LNK($3,$4);
                  CHGTAG($1,PDEFTYPEID);
                }
               ;
```

```
subrange : iconst PDOTDOT iconst
           { $$=join(PDOTDOT,$1,$3,$3->value-$1->value+1); }
         | PTYPECHAR
           { $$=join(PDOTDOT,join(PTYPECHAR,$1,$1,0),
             join(PTYPECHAR,$1,$1,255),256); }
         ;

iconst : signed_iconst { $$=$1; }
       | iconst PPLUS simple_iconst
         { $$=join(PPLUS,$1,$3,$1->value+$3->value); }
       | iconst PMINUS simple_iconst
         { $$=join(PMINUS,$1,$3,$1->value-$3->value); }
       ;

signed_iconst : simple_iconst { $$=$1; }
              | PPLUS simple_iconst { $$=join(PPLUS,NULL,$2,$2->value); }
              | PMINUS simple_iconst
                { $$=join(PMINUS,NULL,$2,-($2->value)); }
              ;

simple_iconst : PINTEGER { $$=join(PINTEGER,$1,NULL,getval($1)); }
              | NMACRO { $$=join(NMACRO,$1,NULL,getval($1)); }
              | PCONSTID { $$=join(PCONSTID,$1,NULL,getval($1)); }
              ;

file_type : packed PFILE POF typename { $$=$2; }
          | packed PFILE POF subrange { $$=$2; }
          ;

packed : PPACKED
       |
       ;

typename : PTYPEINT { $$=NULL; }
         | PTYPEREAL { $$=NULL; }
         | PTYPEBOOL { $$=NULL; }
         | PID { $$=NULL; }
         ;

record_type : packed PRECORD fields PEND { LNK($2,$4); LNK($3,$4);
                if ($3) CHGTAG($4,PSEMICOLON); else IGN($4); $$=NULL; }
            | packed PRECORD variant_part PEND
              { LNK($2,$4); LNK($3,$4); IGN($4); $$=NULL; }
            | packed PRECORD fields PSEMICOLON variant_part PEND
              { LNK($2,$6); LNK($3,$4); LNK($5,$6); IGN($6); $$=NULL; }
            ;

fields : recordsection { $$=$1; }
       | fields PSEMICOLON recordsection { LNK($1,$2); $$=$3; }
       ;
```

```
/*  in a recordsection the first PID links to the PCOLON, the recordsection
    points to the PCOLON    */
recordsection : { $$=NULL; }
              | recids PCOLON type { LNK($1,$2); IGN($2); $$=$2; }
              | recids PCOLON subrange
                { LNK($1,$2); CHGTAG($2,CTSUBRANGE); UP($2,$3); $$=$2; }
              ;

/*  recids point to the first PID which is changed to PDEFVARID    */
recids : PID { $$=$1; CHGTAG($1,PDEFVARID); }
       | recids PCOMMA PID { $$=$1; }
       ;

variant_part : PCASE PID POF variants { IGN($1);IGN($2);
                  CHGTAG($3,CUNION); $$=$3; }
             ;

variants : variant
         | variants variant
         ;

variant : PINTEGER PCOLON POPEN recordsection PCLOSE PSEMICOLON
          { IGN($1); IGN($2); IGN($3);
            LNK($4,$5);
            IGN($5); }
        | PINTEGER PCOLON POPEN recordsection PSEMICOLON
          recordsection PCLOSE PSEMICOLON
          { IGN($1); IGN($2); CHGTAG($3,PRECORD);
            LNK($3,$8); LNK($4,$5); LNK($6,$7); CHGTAG($7,PSEMICOLON); }
        ;

type : typename
     | file_type
     | record_type
     ;
```

```
array_type : packed PARRAY PSQOPEN iconst PDOTDOT iconst PSQCLOSE
             POF type { LNK($2,$3);
               UP($2,join(PDOTDOT,$4,$6,$6->value-$4->value+1));
               LNK($3,$5); LNK($5,$7); LNK($7,$8);$$=$8; }
           | packed PARRAY PSQOPEN iconst PDOTDOT iconst PSQCLOSE
             POF subrange { LNK($2,$3);
               UP($2,join(PDOTDOT,$4,$6,$6->value-$4->value+1));
               LNK($3,$5); LNK($5,$7); LNK($7,$8);
               CHGTAG($8,CTSUBRANGE); UP($8,$9);$$=$8; }
           | packed PARRAY PSQOPEN PID PSQCLOSE POF type { LNK($2,$3);
             UP($2,$4);LNK($3,$4); LNK($4,$5); LNK($5,$6);$$=$6; }
           | packed PARRAY PSQOPEN PID PSQCLOSE POF subrange
             { LNK($2,$3); UP($2,$4); LNK($3,$4); LNK($4,$5);
               LNK($5,$6); CHGTAG($6,CTSUBRANGE); UP($6,$7);$$=$6; }
           | packed PARRAY PSQOPEN PTYPECHAR PSQCLOSE
             POF type { LNK($2,$3); UP($2,join(PDOTDOT,
               join(PTYPECHAR,$1,$1,0),join(PTYPECHAR,$1,$1,255),256));
               $3->link=join(PTYPECHAR,$3,$5,256); $3->link->link=$5;
               /* the PTYPECHAR comes from a macroexpansion, so we can not
               link it directly */ LNK($5,$6); $$=$6; }
           ;

variables :
          | PVAR vardeclarations { IGN($1); }
          ;

vardeclarations : vardeclaration
                | vardeclarations vardeclaration
                ;

vardeclaration : varids PCOLON type PSEMICOLON { LNK($1,$2);
                   IGN($2); LNK($2,$4); }
               | varids PCOLON array_type PSEMICOLON { LNK($1,$2);
                   IGN($2); LNK($3,$4); LNK($2,$4); }
               | varids PCOLON subrange PSEMICOLON { LNK($1,$2);
                   CHGTAG($2,CTSUBRANGE); UP($2,$3); LNK($2,$4); }
               ;

varids : entire_var { CHGTAG($1,PDEFVARID); $$=$1; }
       | varids PCOMMA entire_var { LNK($1,$3);$$=$3; }
       ;

procedures :
           | procedures procedure
           | procedures function
           ;
```

```
locals : PVAR localvardeclarations { CHGTAG($1,PBEGIN); }
       | PLABEL locallabellist PSEMICOLON localvariables
         { CHGTAG($1,PBEGIN); IGN($3); }
       ;

locallabellist : locallabeldecl
               | locallabellist PCOMMA locallabeldecl { IGN($2); }
               ;

locallabeldecl : NMACRO { IGN($1); SYM($1)->obsolete=1; localize($1); }
               | PINTEGER { IGN($1); }
               | labeldecl PPLUS PINTEGER { IGN($2); IGN($3); }
               ;

localvariables :
               | PVAR localvardeclarations { IGN($1); }
               ;

localvardeclarations : localvardeclaration
                     | localvardeclarations localvardeclaration
                     ;

localvardeclaration : localvarids PCOLON type PSEMICOLON
                      { LNK($1,$2); IGN($2); LNK($2,$4); }
                    | localvarids PCOLON array_type PSEMICOLON
                      { LNK($1,$2); IGN($2); LNK($3,$4); LNK($2,$4); }
                    | localvarids PCOLON subrange PSEMICOLON
                      { LNK($1,$2); CHGTAG($2,CTSUBRANGE);
                        UP($2,$3); LNK($2,$4); }
                    ;

localvarids : localentire_var { CHGTAG($1,PDEFVARID); $$=$1; }
            | localvarids PCOMMA localentire_var { LNK($1,$3);$$=$3; }
            ;

localentire_var : PID { $$=$1; localize($1); }
                | CREFID { $$=$1; CHGTAG($1,PID);
                    CHGID($1,PID); localize($1); }
                ;

procedure : pheading locals PBEGIN statements PEND PSEMICOLON
            { IGN($3); IGN($6); wend($4,$5); wsemicolon($4,$5);
              scope_close(); }
          | pheading PBEGIN statements PEND PSEMICOLON
            { IGN($5); wend($3,$4); wsemicolon($3,$4); scope_close(); }
          | pheading PFORWARD PSEMICOLON { scope_close(); }
          ;
```

```
function : fheading PBEGIN { function=1; }statements PEND PSEMICOLON
           { function=0; wreturn($4, 1,NULL); IGN($6);
             wsemicolon($4,$5); scope_close(); }
         | fheading locals PBEGIN { function=1; }
           statements PEND PSEMICOLON
           { int f_no=$1->sym_no;
             function=0;
             if (f_no==x_over_n || f_no==xn_over_d)
             { DBG(dbgcweb,"Discovered function %s; in line %d\n",
               SYM($1)->name,$1->lineno);
               CHGTAG($3,PFBEGIN); $3->sym_no=f_no;
               CHGTAG($6,PFEND); $6->sym_no=f_no;
             }
             else
             { IGN($3);
               wreturn($5,1,NULL);
             }
             wsemicolon($5,$6);
             IGN($7);
             scope_close();
           }
         ;

pid : PID { scope_open(); $$=$1; START_PARAM; }
    | PPROCID { scope_open(); $$=$1; START_PARAM; }
    | PFUNCID { scope_open(); $$=$1; START_PARAM; }
    ;

pheading : PPROCEDURE pid PSEMICOLON
           { LNK($1,$3); CHGID($2,PPROCID); CHGVALUE($2,1); IGN($3); }
         | PPROCEDURE pid POPEN formals PCLOSE PSEMICOLON
           { LNK($1,$3); CHGID($2,PPROCID); CHGVALUE($2,param_mask);
             LNK($4,$5); IGN($6); }
         ;

fheading : PFUNCTION pid PCOLON typename PSEMICOLON
           { $$=$2; LNK($1,$3); CHGID($2,PFUNCID);
             CHGVALUE($2,1);IGN($3); LNK($3,$5); IGN($5); }
         | PFUNCTION pid POPEN formals PCLOSE
           PCOLON typename PSEMICOLON { $$=$2; LNK($1,$3);
             CHGID($2,PFUNCID); CHGVALUE($2,param_mask);
             LNK($4,$5); LNK($5,$6); IGN($6); LNK($6,$8); IGN($8); }
         ;
```

```
formals : formalparameters { $$=$1; }
        | formals PSEMICOLON formalparameters
          { LNK($1,$2); CHGTAG($2,PCOMMA); $$=$3; }
        ;

formalparameters : params PCOLON typename
                   { LNK($1,$2); IGN($2); $$=$2; }
                 ;

params : param { $$=$1; }
       | params PCOMMA param { LNK($1,$3);$$=$3; }
       ;

param : entire_var { NEXT_PARAM; CHGTAG($1,PDEFPARAMID); $$=$1; }
      | PVAR entire_var { REF_PARAM; NEXT_PARAM; IGN($1);
          CHGTAG($2,PDEFREFID);CHGID($2,CREFID); $$=$2; }
      ;

proc_stmt : PPROCID POPEN args PCLOSE { CHGTAG($1,PCALLID); $$=$1;
              UP($2,$1); pstring_args($1,$3); }
          | PCALLID POPEN args PCLOSE
            { $$=$1; UP($2,$1); pstring_args($1,$3); }
          | PPROCID { CHGTAG($1,PCALLID); $$=$1; }
          | PCALLID { $$=$1; }
          ;

function_call : PFUNCID POPEN args PCLOSE
                { CHGTAG($1,PCALLID); $$=$4; UP($2,$1); }
              | PCALLID POPEN args PCLOSE { $$=$4; UP($2,$1); }
              | PFUNCID { CHGTAG($1,PCALLID);$$=$1; }
              | PCALLID { $$=$1; }
              ;

args : arg { $$=$1; }
     | args PCOMMA arg
       { if ($3==NULL) $$=$1; else if ($1==NULL) $$=$3;
         else $$=join(PCOMMA,$1,$3,0); }
     ;

arg : expression { $$=$1; }
    | write_arg { $$=$1; }
    | STRING { $$=$1; }
    | CHAR { $$=$1; }
    ;

write_arg : expression PCOLON expression { $$=$2; }
          ;
```

```
statements : statement { $$=$1; }
           | statements PSEMICOLON statement
             { $$=join(PSEMICOLON,$1,$3,0); }
           ;

statement : stmt { $$=$1; }
          | label PCOLON stmt { clabel($1,0);$$=join(PCOLON,$1,$3,0); }
          | PEXIT PCOLON stmt
            { IGN($1); IGN($2); $$=join(PCOLON,$1,$3,0); }
          ;

goto_stmt : PGOTO label { clabel($2,1); $$=join(PGOTO,$2,NULL,0); }
          | PGOTO PEXIT { IGN($1); $$=$2; }
          | CIGNORE PEXIT { $$=$2; }
          | PRETURN { if (function) clabel($1,1);
              else { CHGTAG($1,CPROCRETURN);$1->sym_ptr->value++; }
              $$=$1; }
          ;

label : PINTEGER
      | NMACRO
      | CLABEL
      | NMACRO PPLUS PINTEGER { seq($1,$3); $$=$1; }
      ;

stmt : simple_stmt
     | structured_stmt
     ;

simple_stmt : empty_stmt
            | assign_stmt
            | return_stmt
            | goto_stmt
            | proc_stmt
            ;

empty_stmt : { $$=join(CEMPTY,NULL,NULL,0); }
           ;

assign_stmt : variable PASSIGN expression { $$=$2; }
            | variable PASSIGN STRING { $$=$2; pstring_assign($1,$3); }
            | variable PASSIGN POPEN STRING PCLOSE
              { $$=$2; pstring_assign($1,$4); }
            ;
```

```
return_stmt : PFUNCID PASSIGN expression { $$=$1; }
            | CRETURN CIGNORE expression { $$=$1; }
            | CRETURN CIGNORE expression CIGNORE CIGNORE
              { $$=join(CRETURN,NULL,NULL,0); }
            | CRETURN { $$=$1; }
            | CPROCRETURN { $$=$1; }
            ;

structured_stmt : compound_stmt
                | conditional_stmt
                | repetitive_stmt
                ;

compound_stmt : PBEGIN statements PEND
                { $$=join(PBEGIN,$2,NULL,0); wsemicolon($2,$3); }
              ;

conditional_stmt : if_stmt
                 | case_stmt
                 ;

if_stmt : PIF expression PTHEN statement { $$=join(PIF,$4,NULL,0); }
        | PIF expression PTHEN statement PELSE statement
          { wsemicolon($4,$5); $$=join(PELSE,$4,$6,0); }
        ;

case_stmt : PCASE expression POF case_list PEND { LNK($1,$3);
              wsemicolon($4,$5);$$=join(PCASE,$4,NULL,0); }
          | PCASE expression POF case_list PSEMICOLON PEND
            { LNK($1,$3);$$=join(PCASE,$4,NULL,0); }
          ;

case_list : case_element
          | case_list PSEMICOLON case_element { $$=join(CCASE,$1,$3,0);
              wsemicolon($1,$2); CHGTAG($2,CBREAK); UP($2,$1); }
          | case_list CBREAK case_element
            { $$=join(CCASE,$1,$3,0); /* etex parses same module twice */ }
          ;

case_element : case_labels PCOLON statement { $$=$3; }
             | POTHERS statement { $$=$2; }
             ;

case_labels : case_label
            | case_labels PCOMMA case_label
              { CHGTAG($2,CCOLON); CHGTEXT($2,": "); }
            | case_labels CCOLON case_label
            ;
```

```
case_label : PINTEGER { winsert_after($1->previous,CCASE,"case "); }
           | NMACRO { winsert_after($1->previous,CCASE,"case "); }
           | PINTEGER PPLUS NMACRO
             { winsert_after($1->previous,CCASE,"case "); }
           | NMACRO PPLUS NMACRO
             { winsert_after($1->previous,CCASE,"case "); }
           | NMACRO PPLUS PINTEGER
             { winsert_after($1->previous,CCASE,"case "); }
           | CCASE NMACRO
           | CCASE PINTEGER
           | CCASE NMACRO PPLUS NMACRO
           | NMACRO PMINUS NMACRO PPLUS NMACRO
             { winsert_after($1->previous,CCASE,"case "); /* etex */ }
           ;

repetitive_stmt : while_stmt
                | repeat_stmt
                | for_stmt
                ;

while_stmt : PWHILE expression PDO statement
             { LNK($1,$3); $$=join(PWHILE,$4,NULL,0); }
           ;

repeat_stmt : PREPEAT statements PUNTIL expression
              { wsemicolon($2,$3); $$=join(PREPEAT,$2,NULL,0); }
            ;

for_stmt : PFOR PID PASSIGN expression PTO varlimit PDO statement
           { mark_for_variable($2,$1->lineno,0,VAR_LOOP);
             DBG(dbgfor,"for variable %s, limit variable in line %d\n",
             SYM($2)->name,$2->lineno);
             $$=join(PFOR,$8,NULL,0);LNK($1,$5);LNK($5,$7); }
         | PFOR PID PASSIGN expression PTO iconst PDO statement
           { mark_for_variable($2,$1->lineno,$6->value,TO_LOOP);
             DBG(dbgfor,"for variable %s, limit up in line %d\n",
             SYM($2)->name,$2->lineno);
             $$=join(PFOR,$8,NULL,0);LNK($1,$5);LNK($5,$7); }
         | PFOR PID PASSIGN expression PDOWNTO iconst PDO statement
           { mark_for_variable($2,$1->lineno,$6->value,DOWNTO_LOOP);
             DBG(dbgfor,"for variable %s, limit down in line %d\n",
             SYM($2)->name,$2->lineno);
             $$=join(PFOR,$8,NULL,0);LNK($1,$5);LNK($5,$7); }
         ;
```

varlimit : *variable*
| *variable* PMINUS *expression*
| *variable* PPLUS *expression*
| *iconst* PSTAR *expression*
;

variable : PID
| CREFID
| *indexed_var*
| *field_var*
| *file_var*
;

entire_var : PID `{ $$=$1; }`
| CREFID `{ $$=$1; CHGTAG($1,PID); CHGID($1,PID); }`
;

indexed_var : *variable* PSQOPEN *expression* PSQCLOSE
| *variable* PSQOPEN STRING PSQCLOSE
| PARRAYFILEID PSQOPEN *expression* PSQCLOSE
;

field_var : *variable* PDOT PID
;

file_var : *variable* PUP
;

expression : *simple_expr* `{ $$=$1; }`
| *simple_expr relop simple_expr* `{ $$=$3; }`
| *simple_expr* PEQ STRING `{ $$=$3; }`
;

relop : PEQ
| PNOTEQ
| PLESS
| PLESSEQ
| PGREATER
| PGREATEREQ
;

simple_expr : *term* `{ $$=$1; }`
| *sign term* `{ $$=$2; }`
| *simple_expr addop term* `{ $$=$3; }`
| *simple_expr addop sign term* `{ $$=$4; }`
;

sign : PPLUS
| PMINUS
;

```
addop : PPLUS
      | PMINUS
      | POR
      ;

term : factor { $$=$1; }
     | term mulop factor { $$=$3; }
     ;

mulop : PSTAR
      | PSLASH { DBG(dbgslash,"Pascal / in line %d\n",$1->lineno); }
      | PDIV
      | PMOD
      | PAND
      ;

factor : variable
       | unsigned_const
       | POPEN expression PCLOSE { $$=$3; }
       | function_call
       | PNOT factor { $$=$2; }
       ;

unsigned_const : real
               | PINTEGER
               | NMACRO
               | PSTRING
               | PCHAR
               | PCONSTID
               ;

real : PREAL
     | PINTEGER PDOT PINTEGER { $$=$3; /* used in line 2361 */}
     ;

%%

const char *tagname(int tag)
{ return yytname[YYTRANSLATE(tag)];
}
```

11 Generating TEX, Running TEX, and Passing the Trip Test

Here I give a step by step instruction on how to get TEX up and running and finally, how to pass Donald Knuth's trip test.

I assume that you have a Unix/Linux system with a terminal window but other operating systems might work as well as long as you have access to the internet (I need files from `www.ctan.org`), an `unzip` program (because packages on `www.ctan.org` come in `.zip` files), and a C compiler.

The recommended, short, and easy way is to start with the file `ctex.w` the `cweb` version of `tex.web`. After all, this is the reason for the whole `web2w` project: to provide you with a `cweb` version of TEX that is much easier to use than the original `WEB` version of TEX. But if you insist, there is also a subsection below that explains how to get `web2w` up and running and use it to generate the `ctex.w` file.

11.1 Generating TEX

1. Download the `web2w` package from `www.ctan.org` and expand the files. Open a terminal window and navigate to the root directory of the package. This directory will be called the `web2w` directory in the following. It contains a `Makefile` that contains most of the commands that are explained in the following.
2. In the `web2w` directory are the files `ctex.c` and `ctex.tex`. If you want to use them, go to step 7; if you want to build them yourself, continue with the next step.
3. TEX and `web2w` are written as literate programs. To use them, you need the `cweb` tools `ctangle` and `cweave` that I build now.

   Since the TEX program is a pretty big file, you can not use the standard configuration even if you have `ctangle` and `cweave` already installed.

   Now download the `cweb` package from `www.ctan.org` and expand the files in the `web2w` directory creating the subdirectory `cweb`.

   Change to this subdirectory and try `make`. If it builds `ctangle` and `cweave` (using the preinstalled programs) skip the next step.
4. If it complains that it can not find `ctangle` then it's trying to bootstrap `ctangle` from `ctangle.w` without having `ctangle` to begin with. Try `touch *.c` and try `make` again. This time it should try to make `ctangle` from `ctangle.c` and `common.c`, running:

   ```
   cc -g -c -o ctangle.o ctangle.c
   ```

```
cc -g -DCWEBINPUTS="/usr/local/lib/cweb" -c common.c
cc -g -o ctangle ctangle.o common.o
```

Now you should have `ctangle`. Then building `cweave` should be no problem by running `make`.

5. Next I need to patch `ctangle.w`, `cweave.w`, and `common.w` to enlarge the settings for various parameters. Change to the `cweb` subdirectory and run the commands

```
patch --verbose cweave.w ../cweave.patch
patch --verbose ctangle.w ../ctangle.patch
patch --verbose common.w ../common.patch
make
```

If you do not have the `patch` program, look at the patch files and read them as instructions how to change the settings in `ctangle.w`, `cweave.w`, and `common.w`; you can do these small changes easily with any text editor yourself.

The final `make` should produce a new `ctangle` and `cweave` by running the old `ctangle` on the new `ctangle.w`, `cweave.w`, and `common.w`. The `cweb` directory contains change files to adapt the programs to particular operating systems and it might be a good idea to use them. On an Win32 machine, for example, you might want to write

```
./ctangle ctangle.w ctang-w32.ch
./ctangle cweave.w cweav-w32.ch
./ctangle common.w comm-w32.ch
```

Then run the C compiler again as in the previous step.

6. Now you use your extra powerful `ctangle` and `cweave` from step 5, return to the `web2w` directory, and generate `ctex.c` and `ctex.tex` simply by running the commands

```
cweb/ctangle ctex.w
cweb/cweave ctex.w
```

7. Compiling `ctex.c` is pretty easy: use the command

```
cc ctex.c -lm -o ctex
```

The `-lm` tells it to link in the C math library. You may add other options like `-g` or `-O3` as you like. What you have now is the virgin TEX program (also called `VIRTEX`).

8. If you have TEX on your system, you can generate the documentation with the command

`tex ctex.tex` or `pdftex ctex.tex`.

Otherwise, you will have to wait until step 16.

Note that the above commands will need the files `ctex.idx` and `ctex.scn`. These are part of the `web2w` package and are produced as a side effect of running `cweave` on `ctex.w`.

11.2 Running TEX

9. Producing "Hello world!" with `ctex`.

There are some differences between the plain TEX that you have generated now and the TEX that you get if you install one of the large and convenient

TEX distributions. First, there is no sophisticated searching for font files, formats, and tex input files (as usually provided by the `kpathsea` library), instead files are looked up in the current directory or in the subdirectories `TeXfonts`, `TeXformats`, and `TeXinputs`. Second, the plain TEX that you have now does not come with preloadable format files, you have to generate them first. So let's get started with populating the subdirectories just mentioned with the necessary files from the `www.ctan.org` archives.

The first file is the `plain.tex` file. You find it on `www.ctan.org` in the `lib` subdirectory of `systems/knuth/dist/`. This file defines the plain TEX format; save it to the `TeXinputs` subdirectory.

Now, do the same for the file `hyphen.tex` (same source same destination directory) containing basic hyphenation patterns.

10. Next, you need the TEX font metric files. Download the package "`cm-tfm`— Metric files for the Computer Modern fonts" from `www.ctan.org` and unpack the files in `tfm.zip` into the `TeXfonts` subdirectory.

11. Now you need to create `cinitex`, a special version of TEX that is able to initialize all its internal data structures and therefore does not depend on format files; instead it can be used to create format files. Special versions of `ctex` can be created by defining the C macros `DEBUG`, `INIT`, or `STAT` on the command line. So (compare step 7) run the command
    ```
    cc -DINIT ctex.c -lm -o cinitex
    ```

12. Ready? Start `cinitex` and see what happens. The dialog with `cinitex` should follow the outline below. TEX's output is shown in typewriter style, your input is shown in italics.
    ```
    This is TeX, Version 3.14159265 (HINT) (INITEX)
    **plain
    (TeXinputs/plain.tex Preloading the plain format:  codes,
    registers, parameters, fonts, more fonts, macros,
    math definitions, output routines,
    hyphenation (TeXinputs/hyphen.tex))
    *Hello world!

    *\end
    [1]
    Output written on plain.dvi (1 page, 224 bytes).
    Transcript written on plain.log.
    ```
    Well that's it. You should now have a file `plain.dvi` which you can open with any run-of-the-mill dvi-viewer.

13. To do the same with the virgin `ctex` program, you need a `plain.fmt` file which I produce next. Start `cinitex` again. This time your dialog should be as follows:
    ```
    This is TeX, Version 3.14159265 (HINT) (INITEX)
    **plain \dump
    (TeXinputs/plain.tex Preloading the plain format:  codes,
    registers, parameters, fonts, more fonts, macros,
    math definitions, output routines,
    ```

```
hyphenation (TeXinputs/hyphen.tex))
Beginning to dump on file plain.fmt
 (preloaded format=plain 1776.7.4)
1338 strings of total length 8447
4990 memory locations dumped; current usage is 110&4877
926 multiletter control sequences
\font\nullfont=nullfont
```

⋮

```
14707 words of font info for 50 preloaded fonts
14 hyphenation exceptions
Hyphenation trie of length 6075 has 181 ops out of 500
 181 for language 0
No pages of output.
Transcript written on plain.log
```

Now you should have a file `plain.fmt`. Move it to the `TeXformats/` subdirectory, where plain `ctex` will find it, and you are ready for the final "Hello world!" step.

14. Start the virgin `ctex` program and answer as follows:

```
This is TeX, Version 3.14159265 (HINT) (no format preloaded)
**&plain
*Hello world!
*\end
[1]
Output written on texput.dvi (1 page, 224 bytes).
Transcript written on texput.log
```

The "&" preceding "*plain*" tells TEX that this is a format file. Your dvi output is now in the `texput.dvi` file.

15. If you have `ctex.tex` from step 6, `ctex` from step 7, and `plain.fmt` from step 13, producing `ctex.dvi` using `ctex` itself seems like a snap. Running `ctex` on `ctex.tex` will, however, need the include file `cwebmac.tex` which you should have downloaded already with the `cweb` sources in step 3; copy it to the `TeXinputs/` subdirectory. Then `ctex.tex` will further need the `logo10.tfm` file from the mflogo fonts package. Download the file from the `fonts/mflogo/tfm` directory (part of the mflogo package) on `www.ctan.org` and place it in the `TeXfonts` subdirectory.

    Unfortunately TEX is a real big program and you need not only a super `ctangle` and `cweave`, you need also a super TEX to process it. The out-of-the box `ctex` will end with a "`! TeX capacity exceeded, sorry [main memory size=30001].`"

    So the next step describes how to get this super TEX.

16. Take your favorite text editor and open the file `ctex.w`. Locate the line (this should be line 397) where it says `enum {@+@!mem_max=30000@+};` and change the size to 50000. (You see how easy it is to change the code of TEX now?) It remains to run `ctangle` and `cc` to get the super `ctex`:

```
cweb/ctangle ctex.w
cc ctex.c -lm -o ctex
```

Now start super `ctex` and answer *&plain ctex*. You should get `ctex.dvi`

11.3 Passing the Trip Test

17. Passing the trip test is the last proof of concept!

    Download the package `tex.zip` from `www.ctan.org` which contains the files of `systems/knuth/dist/tex` (this is the original TEX distribution by Donald E. Knuth) and extract the files into the `tex` subdirectory of `web2w` (see also step 21 below).

    Perform all the steps described in `tripman.tex` in the `tex` subdirectory (you might want to create a dvi file with `ctex` before reading it) replacing "`tex.web`" by "`ctex.w`" and "`tangle`" by "`ctangle`". You should encounter no difficulties (if yes, let me know) if you observe the following hints:

    - Make a copy of `ctex.w` and modify the setting of constants as required by step 2 of Knuths instructions. If you have the `patch` program, you might want to use the file `triptest.patch` to get these changes.
    - After generating `ctex.c` from the modified `ctex.w` by running `ctangle`, compile `ctex.c` with the options `-DINIT` and `-DSTAT` like this:
      ```
      cc -DINIT -DSTAT ctex.c -lm -o cinitex
      ```
      Instead of setting **init** and **stats** in `ctex.w`, use the `-D` command line options.

11.4 Generating `ctex.w` from `tex.web`

18. To create `ctex.w` from `tex.web`, you need to build `web2w`, which is written as a literate program. So you can start building it from the file `web2w.w` or use the file `web2w.c` which comes with the `web2w` package. In the latter case, you can skip the next step.

19. You create `web2w.c` and `web2w.h` from `web2w.w` by running
    ```
    ctangle web2w.w   or   cweb/ctangle web2w.w
    ```
    Any `ctangle` program should work here, but it doesn't harm if you use your own `ctangle` created in step 5.

    I do not describe how to produce `web2w.pdf` from `web2w.w`: First, because you seem to have that file already if you are reading this, and second, because it is a much more complicated process. In addition, if you like reading on paper and prefer a nicely bound book over a mess of photocopies, you can buy this document also as a book titled "WEB to cweb"[8].

20. From `web2w.c`, `web2w.h`, `web.l`, and `pascal.y`, you get `web2w` by running
    ```
    flex -o web.lex.c web.l
    bison -d -v pascal.y
    cc -o web2w web2w.c web.lex.c pascal.tab.c
    ```
    The first command produces the scanner `web.lex.c`; the second command produces the parser in two files `pascal.tab.c` and `pascal.tab.h`. If your version of `bison` does not support an api prefix, you can use the option `-p pp` instead. The last command invokes the C compiler to create `web2w`.

21. Next I want to run `tex.web` through `web2w`. To obtain `tex.web` download the package `tex.zip` from `www.ctan.org` which contains the files of the original TEX distribution by Donald E. Knuth in directory `systems/knuth/dist/tex` and extract the files into the `tex` subdirectory of `web2w` (see also step 17).
22. Now I am ready to apply `web2w`. Run
    ```
    ./web2w -o tex.w tex/tex.web
    ```
    This command will produce `tex.w`, but I am not yet finished. I have to apply the patch file `ctex.patch` to get the finished `ctex.w` like this:
    ```
    patch --verbose -o ctex.w tex.w ctex.patch
    ```
    And `ctex.w` has been created.

References

[1] *Web2c: A TEX implementation.* `https://tug.org/texinfohtml/web2c.html`.

[2] Silvio Levy Donald E. Knuth. *The CWEB System of Structured Documentation.* Addison Wesley, 1994.

[3] C. O. Grosse-Lindemann and H. H. Nagel. Postlude to a pascal-compiler bootstrap on a decsystem-10. *Software: Practice and Experience*, 6(1):29–42, 1976.

[4] Donald E. Knuth. *The WEB system of structured documentation.* Calif. Univ. Stanford. Comput. Sci. Dept., Stanford, CA, 1983.

[5] Donald E. Knuth. *TEX: the Program.* Computers & Typesetting B. Addison-Wesley, 1986.

[6] Donald E. Knuth. *Literate Programming.* CSLI Lecture Notes Number 27. Center for the Study of Language and Information, Stanford, CA, 1992.

[7] Donald E. Knuth. *The Art of Computer Programming.* Addison Wesley, 1998.

[8] Martin Ruckert. *WEB to cweb.* 2017.

Index

B

C

D

E

F

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z

Crossreference of Sections

⟨ Character k cannot be printed ⟩ Defined in section 128. Used in section 127.
⟨ convert some strings to macro names ⟩ Defined in section 125 and 136.
Used in section 124.
⟨ convert token t to a string ⟩ Defined in section 39 and 189. Used in section 99.
⟨ convert `NMACRO` from `WEB` to `cweb` ⟩ Defined in section 143. Used in section 141.
⟨ convert `OMACRO` from `WEB` to `cweb` ⟩ Defined in section 142. Used in section 141.
⟨ convert `PMACRO` from `WEB` to `cweb` ⟩ Defined in section 145. Used in section 141.
⟨ convert t from `WEB` to `cweb` ⟩ Defined in section 79, 80, 104, 106, 107, 108, 109, 112, 115, 116, 119, 121, 122, 123, 124, 141, 147, 154, 155, 159, 160, 161, 163, 164, 169, 170, 171, 172, 175, 178, 179, 180, 185, 191, 192, 193, 196, 197, 198, 205, 206, and 207.
Used in section 101.
⟨ count macro parameters ⟩ Defined in section 144. Used in section 90.
⟨ decide whether to replace a subrange type for loop control variables ⟩
Defined in section 184. Used in section 159.
⟨ external declarations ⟩ Defined in section 5, 6, 9, 14, 16, 17, 19, 21, 24, 25, 26, 27, 32, 38, 40, 41, 46, 48, 49, 55, 59, 60, 63, 65, 68, 83, 85, 94, 105, 130, 148, 150, 156, 162, 173, 176, 181, 183, 188, 194, 200, 202, 210, 211, 215, 216, and 217. Used in section 2.
⟨ finalize token list ⟩ Defined in section 66, 67, 81, 89, and 120. Used in section 4.
⟨ functions ⟩
Defined in section 13, 18, 23, 31, 36, 43, 44, 47, 50, 52, 57, 58, 64, 69, 71, 72, 84, 95, 97, 99, 101, 103, 111, 118, 131, 132, 134, 149, 167, 174, 177, 182, 186, 187, 199, 201, and 209.
Used in section 1.
⟨ generate `cweb` output ⟩ Defined in section 100 and 126. Used in section 1.
⟨ generate array offset ⟩ Defined in section 166. Used in section 164.
⟨ generate array size ⟩ Defined in section 165. Used in section 164.
⟨ generate definition for string k ⟩ Defined in section 129. Used in section 126.
⟨ generate definitions for the first 256 strings ⟩ Defined in section 127.
Used in section 126.
⟨ generate macros for some strings ⟩ Defined in section 137. Used in section 129.
⟨ generate string pool initializations ⟩ Defined in section 140. Used in section 126.
⟨ global variables ⟩ Defined in section 11, 15, 20, 29, 34, 42, 45, 51, 54, 61, 70, 96, 113, 133, 138, 146, 151, 158, 195, 203, and 212. Used in section 1.
⟨ initialize token list ⟩ Defined in section 22, 62, 114, 135, 139, 152, 204, and 208.
Used in section 4.
⟨ internal declarations ⟩
Defined in section 3, 10, 28, 33, 53, 98, 102, 110, 117, 157, and 168. Used in section 1.

⟨ internal node ⟩ Defined in section 93. Used in section 6.
⟨ leaf node ⟩ Defined in section 7. Used in section 6.
⟨ open the files ⟩ Defined in section 214. Used in section 213.
⟨ parse Pascal ⟩ Defined in section 92. Used in section 1.
⟨ process the command line ⟩ Defined in section 213. Used in section 1.
⟨ process the end of a code segment ⟩ Defined in section 87. Used in section 72.
⟨ read the `WEB` ⟩ Defined in section 4. Used in section 1.
⟨ show summary ⟩ Defined in section 12, 30, 35, and 56. Used in section 1.
⟨ special treatment for `WEB` tokens ⟩ Defined in section 73, 74, 75, 76, 77, 78, 82, 86, 88, 90, 91, 153, and 190. Used in section 72.
⟨ token specific info ⟩ Defined in section 8 and 37. Used in section 7.
⟨ `web2w.h` ⟩ Defined in section 2.