WEB to cweb

WEB to cweb

Converting TEX from WEB to cweb

Für meinen Vater

MARTIN RUCKERT *Munich University of Applied Sciences*

Second edition

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

Internet page http://hint.userweb.mwn.de/hint/web2w.html 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
Second edition, August 2021

Revision: 2463, Date: Wed, 04 Aug 2021

Preface

This book describes a project to convert the TEX source code[3] written by Donald E. Knuth as a "WEB"[4] into a "cweb"[6].

- 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 feed 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*

Preface to the Second Edition

- It is November 11, 2020, by now and to my own surprise, most of the "fussy ideas" presented in the previous paragraphs have become a reality. Using `ctex.w`, the output of the `web2w` program, as the basis for the HINT project[7, 9, 10, 11] was a clear success. The success was so complete, that my initial idea of $0.x$ versions and improved $1.y$ versions turned out to be complete nonsense. As a consequence, I will switch to a conventional numbering schema presenting in this book version 1.0 of `web2w`.

  There were a few changes necessary—of course. In 2019, I got a strange compiler error, when I did ask `gcc` to optimize floating point math. I found out that the include file replaced a few math functions by macros, and these macros used the C keyword "*register*". In the TEX sources, you find the following line:

  define *register* = 89 { internal register (`\count`, `\dimen`, etc.) }

  It defines *register* as the number 89. Fortunately, `web2w` already contained code to rename identifiers consistently, and I renamed it to *internal_register*. Later the year, I had to rename TEX's macros *read*, *write* and *close* because they conflict with the file handling functions in the standard C library. Other than that, no changes were necessary until today.

  While working with `ctex.w`, however, some shortcomings became apparent: the constant struggle to avoid name conflicts with TEX's macros, the cumbersome manipulation of the string pool, and the limitation of 16-bit pointers to name just the most important ones. Section 2 gives a detailed account of all the small and big improvements made in version 1.0.

- It is December 31, 2020, the last day of a remarkable year. Yesterday, I finished what I thought then would be version 1.0 of `web2w`. The 32-bit version as well as the 64-bit version passed the triptest and the extended triptest with flying colors. But today, after a good breakfast, a new idea suddenly popped up in my mind. I had never planed to make my C version of TEX anything but the basis of the HINT project. But during that project, I had written a change file that incorporated the ksearchpath library into the file handling code and adopted the command line conventions of TEX Live. Now it occurred to me that the proper place for these extensions would be the `web2w` sub-project not the HINT project proper. It could turn `ctex.w` into `ktex.w` (named in honor of Karl Berry) creating a new TEX engine that could in fact be used not only for experiments but also for daily use.

- On the evening of February 4, 2021, `ktex` compiled and run for the first time. To polish up the last details, I started reading `ctex.dvi` fixing things that did not look nice. On my way, I found that TEX is redefining `\?` as `\relax`, which is OK for Pascal but it hides C's "?" operator. Further there were some ugly placements of the "**case**" keyword in the code. While the first issue was easy to solve (the scanner can replace `\?` by `\@`) the second issue took me two full days.
- On March 10, 2021, I thought I finished work on version 1.0 of `web2w` and turned my attention back to the HINT project, only to discover, that I would need another feature in web2w: the generation of a suitable header file exporting macros, constants, types, and selected variables and functions. So I added the `-h` and `-e` command line options described in section 7.3. This kept me busy for another week. During that time I decided to reorganize the conversion of Pascal's subrange types, preferring **int** wherever possible, and then made a final attack on the static initialization of the string pool, to arrive at the version of `web2w` that is presented in the following.

Wolfgang, March 2021 *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. [1] see [3]). 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`[12] 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:

```
define inf_bad = 10000  { infinitely bad value }
        ⋮
if r > 1290 then badness ← inf_bad  { 1290^3 < 2^31 < 1291^3 }
else badness ← (r * r * r + ´400000) div ´1000000;
```

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. Here it would lead to a semicolon inserted after 10000 in the *inf_bad* macro. So the right place to insert a semicolon in one instance can be the wrong place in another instance. `web2w` converts Pascal's assignment to *badness*, the function name, into a return statement and places the semicolon correctly behind the macro invocation like this:

```
#define inf_bad 10000      /* infinitely bad value */
        ⋮
  if (r > 1290) return inf_bad;      /* 1290^3 < 2^31 < 1291^3 */
  else return (r * r * r + °400000)/°1000000;
```

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 8). 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 9). 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 10.

2 Changes to web2w in Version 1.0

First of all, we now have a version number:

⟨ web2w version 1 ⟩ ≡ (1)

```
"1.0"
```

The other changes can be roughly divided into three categories: semantic changes, cosmetic changes, and structural changes. I start with the semantic changes: the changes that affect the semantics of the program produced by web2w. Then I explain the cosmetic changes that cause web2w to produce nicer looking output, and finally I introduce the structural changes that are motivated mainly by the goal to integrate the produced program into the TEX Live distribution.

2.1 Signed/unsigned comparison

In Pascal, comparing of two integer expressions will always return a correct value, because Pascal maps both operands onto a common ordinal type, "large enough" for both operands, before comparing them. This is different in C, where comparing a signed and an unsigned quantity might not produce the expected outcome. With the right warnings enabled, a C compiler will emit complaints about **signed/unsigned** comparisons. Even more complaints are caused by quite a few of TEX's functions that use **integer** as the type of its parameters, while the actual argument is a **pointer** or a **str_number**, leading to unnecessary comparisons between signed and unsigned data types.

Version 0.4 used to replace the PCOLON token preceding the type in a variable declaration either by a CIGNORE token or by a CTSUBRANGE token to mark subrange types. Version 1.0 introduces two new token types: CTINT replaces CTSUBRANGE for global and local variables of ordinal type but not for record fields, arrays, type definitions, and named types; and CTLOCAL replaces CIGNORE for local variables of a named ordinal type. Marked variable types are systematically replaced by **int**. This converts most unstructured variables that belong in Pascal to an enumeration or subrange type to **int** variables. This is in line with Pascal's concept of the "host type" of such a type.

2.2 String pool initialization

Another spot that needed further attention is TEX's string pool. 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. Version 0.4 of web2w replaces literal strings by section names which then expand to an index in the *str_start* array.

Together with a suitable static initialization of the *str_start* and *str_pool* array, this had the desired effect and it was quite readable. For a subset of these literal strings, web2w took extra action to keep them in the code as literal C strings. This had the advantage that these strings were easier to change when modifying the generated .w or .c files.

When implementing the HINT viewer, it turned out that TEX's entire string pool was still necessary to compile it; the code inherited from TEX still contained a few references to the string pool. This seems unnecessary because the HINT viewer does not deal with TEX's control sequences or string handling functions. When implementing version 1.0 of web2w, I started to reduce the amount of strings that enter the string pool further, until only the names of control sequences remained in the string pool. With these names also the first argument of the function *primitive* remained a string number.

Defining new TEX primitives in change files is, however, common for typical extensions of TEX. With ctex.w this has proved to be a bit cumbersome. You can not just write *primitive*(⟨ "newname" ⟩, ...); but you need to define the section name such that it expands to a *str_start_...* macro which you need to define depending on the initialization of the *str_start* array which in turn depends on the initialization of the *str_pool* array. A cumbersome and error prone process.

So in the end, I decided to eliminate the static initialization of the string pool entirely. In version 1.0 the string pool is initialized at runtime with the first 256 single character strings and the empty string. Most other strings are added in the INI version by the *primitive* function. The advantage is simplicity and readability; the disadvantage is the overhead in time and space because names of control sequences will now exist twice: the static string that is the argument of the *primitive* function and its copy in the string pool.

Here is some data on the incurred overhead to justify the decision: Version 0.4 already reduced the initial number of strings in the string pool from 1044 to 730 and the initial size of the string pool from 22742 byte to 4700 byte. Further reductions in version 1.0 left 611 strings with a total of 3701 byte in the string pool but still without simplifying the addition of new primitive control sequences. The next logical step was abandoning the static initialization of the string pool altogether. Two alternatives came to my mind: removing the string pool completely or switching to a dynamic initialization of the string pool. I decided for the second alternative for the following reasons:

- While dynamic initialization adds an overhead in space and time because the strings are present as C string literals and are copied at runtime to the string pool, the space overhead is small (about 1% of the executable's size) and the time overhead is incurred only in the INITEX version of TEX.
- It makes addition of new string literals as simple as possible.
- It simplifies the implementation of web2w.
- It avoids unnecessary changes to Donald Knuth's data structures and algorithms.

As a side effect of this transformation, the *TEX_area* and *TEX_font_area* macros of TEX now were defined as ordinary C strings and the time was ripe to change "TeXinputs:" to "TeXinputs/" and "TeXfonts:" to "TeXfonts/".

Keeping the WEB strings as C string literals in the code and keeping the string pool as well implied changing variable types from **str_number** to **char** $*$ in numerous but not all places. I used an heuristic approach to decide whether to convert or not. I count assignments of string literals and string numbers to variables and function parameters and converted them if the literal strings had the majority. For the most common string functions *print* and *print_esc*, I converted the calls to *printn* and *printn_esc* in case the argument was still a string number. All the remaining inconsistencies are resolved in the final `ctex.patch` file.

2.3 64 bit TEX

The biggest problem with the version 0.4 `ctex.w` was the limitation to a 16-bit **pointer** type which allows access to at most 2^{16} of TEX's memory words. To run a typical LATEX job, loading only a few of the most common packages, this is usually not sufficient. And for most people, TEX is just a synonym for LATEX. Yes, quite a few of them do not even know that LATEX is not the only way of using TEX. Hence, an implementation of TEX that is restricted to 16-bit pointers is a nice research project but it is not suitable for processing typical LATEX workloads. The solution is to investigate the move to 32-bit **pointer** types, pointing to 64-bit memory words. I expected this should be feasible without creating too many complications, because I remember having seen in the 70's a TEX implementation using 48-bit words. So the Pascal code with all its enumeration types should adjust gracefully to a larger range for the **pointer** type.

I started by finding out how big TEX's data structures are in common implementations of TEX by looking at the code of pdfTEX in the TEX Live 2019 distribution that is currently installed on my computer. There are default values for TEX's constants that govern these sizes in `texini.c` and the dynamic values, determined by configuration files, can be found by calling `kpsewhich -var-value` *variable-name*.

Here are my findings:

Constant	Default	Dynamic	Original
mem_bot	0		0
main_memory	250 000	5 000 000	30 000
pool_size	200 000	6 250 000	32 000
pool_free	5 000	47 500	
string_vacancies	75 000	90 000	8 000
max_strings	15 000	500 000	3 000
strings_free	100		
font_mem_size	100 000	8 000 000	20 000
font_max	500	9 000	75
trie_size	20 000	1 000 000	8 000
trie_op_size	35 111		500
hyph_size	659	8 191	307
buf_size	3 000	200 000	500
nest_size	50	500	40
max_in_open	15	15	6
param_size	60	10 000	60

save_size	4 000	100 000	600
stack_size	300	5 000	200
dvi_buf_size	16 384	16 384	800
error_line	79	79	72
half_error_line	50	50	42
max_print_line	79	79	79
hash_extra	24 526	600 000	
hash_size	15 000		2 100
hash_prime	8 501		1 777
max_halfword	#FFF FFFF		#FFFF
max_quarterword	#FF		#FF

TEX's *mem* array extends from index *mem_min* to *mem_max* and TEX Live sets $mem\_top = mem\_bot + main\_memory - 1$, $mem\_min = mem\_bot$, and $mem\_max = mem\_top$. *hash_extra* becomes the size of the hash table between *hash_offset* and *hash_top*; so I might need to increase the hash size by this additional amount. *pool_free* and *strings_free* specify the amount of free space required in the string pool and the string start array after undumping a format file.

I choose $file\_name\_size \equiv 1024$.

Another important choice is the value for *max_halfword*. TEX tests **if** $(2 * max\_halfword < mem\_top - mem\_min)$ *bad* $= 41$; so $2 * max\_halfword$ must not produce an overflow. Choosing $max\_halfword \equiv$ #3FFF FFFF satisfies this condition.

Allocating bigger arrays is not the problem. The problem is the storage space needed for the array indices. So, for example, the main memory is accessed using variables of type **pointer**. Pointers are stored in a *halfword* which in version 0.4 is a **uint16_t** and two halfwords must fit in a *memory_word*. These data structures must be redefined. It was unclear how the packing and unpacking of memory words would be affected by the changes in the structure of these data types.

Well—after changing the constants mentioned above, `web2w` did run without complaints, the patch file needed some changes to adapt to the new data structures, replacing a **uint8_t** by a **uint16_t** at one place and a **uint16_t** by a **uint32_t** at another place, until at the end, to my own surprise, the GNU C compiler would again compile `ctex.c` without further errors or warnings.

2.4 Macro names

Avoiding name conflicts with the long list of macros defined by TEX was a constant concern when working on the HINT project where TEX code was used as part of a larger software project. In the C language, there is no separate name space for macros. Because the preprocessor expands macros before scanner and parser get to see the source file, macro names defined inside a source file have something like a "super-file-scope". Their visibility is limited to the file, but even names of local variables or field names inside a structure are not protected against expansion. Macro names defined inside a header file must be considered "super-global"; they cause—often unexpected—expansions everywhere. Since TEX code relies heavily on macro expansion, it is not possible to use any part of it without including a

header file with many—if not all—macro definitions of TEX. While this created numerous problems, it was possible but not convenient to work around them.

When using the C language, it is common practice to avoid conflicts between macros and variable, function, or field names by making macros use all upper case names where as ordinary names use at least some lower case letters. So the obvious idea is to implement a command line option for `web2w` that makes macro names use upper case letters. Since TEX targets a Pascal compiler and Pascal programs are not case sensitive, it is possible to do this without creating new name conflicts. Changing all macro names to upper case will, however, impact the visual appearance of the TEX program considerably. Therefore this replacement is optional. You might even create the documentation using lower case macros and the running program using upper case macros.

2.5 Macro parameters

The `WEB` system allows the definition of parametrized macros using a single # sign to mark the insertion point(s) for the parameter text. This does not prevent you from passing multiple parameters because commas are allowed in the parameter text. Already in version 0.4 this had to be taken into account because the C preprocessor interprets commas in the parameter text as a separator between multiple parameters and insists that macro definition and macro use agree on the number of parameters. The solution was counting the arguments when a macro was used and inserting the necessary number of macro parameters when writing out the macro definition. In a few cases where the number of parameters would vary, variadic macros were defined using the `ctex.patch` file. The implementation of `WEB` macros however implies one restriction: at every insertion point the complete parameter text—with all its commas—is inserted. It is not possible to insert only a single comma separated component. To overcome this restriction, TEX resorts to “tail calls” in it macro definitions. As an example consider the definition of *char_info* as shown in section 6.9. Version 1.0 now implements the unrolling of the tail calls and produces truly C style macro definitions.

2.6 “case” keyword placement

One of the most difficult problems when converting Pascal to C is the correct placement of the **case** keyword when converting Pascal’s **case** ... **of** statement to C’s **switch** statement. Most problems were solved in version 0.4 by passing TEX’s numeric macros directly to the parser without expanding them. Unfortunately, ϵ-TEX introduced some ordinary macros that expanded to plain integers and used them as case labels—with ugly `cweb` code as a result. To remedy the situation, version 1.0 now includes a postprocessing step to convert ordinary macros to numeric macros if possible.

It was even more difficult to get the placement of the **case** keyword “correct”, when the label was a macro parameter. The solution is described in section 6.16 and requires checking the sequence number and the nesting level of expansions.

While solving the problem, I added links from closing braces to opening braces allowing for a faster scan backwards over Pascal comments. A technique which probably can be useful in other places as well.

2.7 Trailing spaces

Version 0.4 of web2w occasionally generated lines that ended with a space or tab character. Igor Liferenko pointed me to this problem which is a nuisance when creating change files. To eliminate these trailing spaces, a new counter was added to count spaces instead of writing them to the output. Using this counter, the primitive *wput* operation will postpone the output of spaces until a non-space character needs to be written. Further, it led me to using a space instead of a tab character between macro name and macro definition. This is also consistent with tex.web where no attempt is made to align the right hand sides of macro definitions. In the process of implementing these changes, the basic output routines in section 6.1 were reorganized.

2.8 Constants in the outer block

I never liked the way version 0.4 handled the ⟨ Constants in the outer block ⟩: creating a separate enumeration type for each constant. The new version will now generate a single enumeration type for all ⟨ Constants in the outer block ⟩. The biggest problem, which led me to the old implementation, is the final constant in this list: the *pool_name*. In the WEB implementation it's an integer constant (a string number) in the cweb implementation it became a literal string which can not be part of a C enumeration type. The first attempt at improving the situation was inserting the "**enum** {" before the first constant declaration and to place the closing "} ;" of the enumeration after the last such declaration. It needed the help of the parser to identify the first and the last declaration, and scanning backward from the last declaration (the constant string), skipping all non-Pascal tokens until arriving at the proper place to insert the closing "} ;". It worked quite well, but I realized later that *pool_name* is no longer used because there is no string pool file any more. This led to the much nicer implementation: using Pascal's **const** keyword as a hook to insert the "**enum** { ... } ;", and eliminating *pool_name* using the final patch file.

2.9 Miscellaneous changes

When I thought I would be done, I started to read cweb.dvi from start to finish and discovered a whole series of necessary or at least desirable improvements.

For example, I eliminated the definition of *done6*, which was never used, by marking it as obsolete. Then I generalized the concept introducing two new counters *scan_count* and *use_count* counting all the occurrences of an identifier (except for macro and format definitions) and all usages of an identifier. Based on this, I eliminated all definitions of macros that were never used—except for a few of them, like *top_skip* or *toks*, keeping them for documentation purposes. While at it, I ended the (mis)use of the *value* field in the symbol table for counting the **goto**'s by introducing a new field named *goto_count*.

At other places, I started to rename variables for greater consistency. Now I consistently use the ww_ prefix for variables and functions in the scanner, and pp_ for the parser; string numbers get the suffix _no; and underscores are used to structure variable names. Then I stumbled over a code sequence which seemed to make no sense—until I discovered that the question mark operator of C was

missing. It turned out that TEX was redefining `\?` as `\relax` (which is OK for Pascal code). A small change in the scanner now replaces `\?` by `\@`. Another change replaces `\AT!` (defined in `webmac.tex`) by `\AT` (defined in `cwebmac.tex`). On other occasions, I simply used the patch file to fix TEX code that just didn't look right.

I also made an attempt at rewriting TEX's help system having only a single help macro for any number of lines. Based on the fact that a C compiler concatenates adjacent strings, all lines were rewritten to include a trailing newline and the help macro would just assign the string to the *help_ptr*. The code grew complex and it required quite a few patches to cope with all the nasty things that TEX and ε-TEX do with the *help_line* array. Since it neither improved readability, nor performance, and neither maintainability, but just adds complexity, I removed it again after working on it for two full days. I had more luck with the removal of the tail calls in parametrized macros. This job was accomplished without too much headache in less than a day.

2.10 TEX Live integration

The new TEX engine that is generated using `web2w` aspires to become a member of the TEX Live family of programs. To reach this goal, four major accomplishments are necessary:

The new TEX engine must

- fit in with the TEX Live build process,
- respect the TEX Live conventions for command line parameters,
- find its input files using the `kpathsearch` library, and
- implement TEX primitives to support LATEX.

To simplify the build process, I introduced further changes to `web2w`. Name conflicts when linking TEX with external libraries are avoided by declaring all functions (except *main*) and global variables as **static**. I also needed a convenient method to include additional header files. Header files need to go before defining TEX's macros, because very often macro names (e.g. *link*, *name*, *time*, ...) conflict with identifiers used in standard header files. Therefore, I renamed ⟨ Compiler directives ⟩ to ⟨ Header files and function declarations ⟩ and made it the first section of the program. After that, I included the macro definitions using `@h`, followed by constants, types, global variables, and functions.

To arm the new TEX engine with the necessary extended functionality for LATEX, it is based on ε-TEX; and to supply it with sufficient resources to cope with large LATEX packages, it is based on the 64-bit version of `ctex`. But `ectex64.w`, the extended 64-bit version of TEX, is still not enough. A final change file `ktex.ch` is necessary to produce `ktex.w`. Figure 6 illustrates the complete build process.

To read the complete documentation of these changes run "`cweave ectex64.w ktex.ch ktex.tex`", then run "`ktex ktex.tex`" and read `ktex.dvi`.

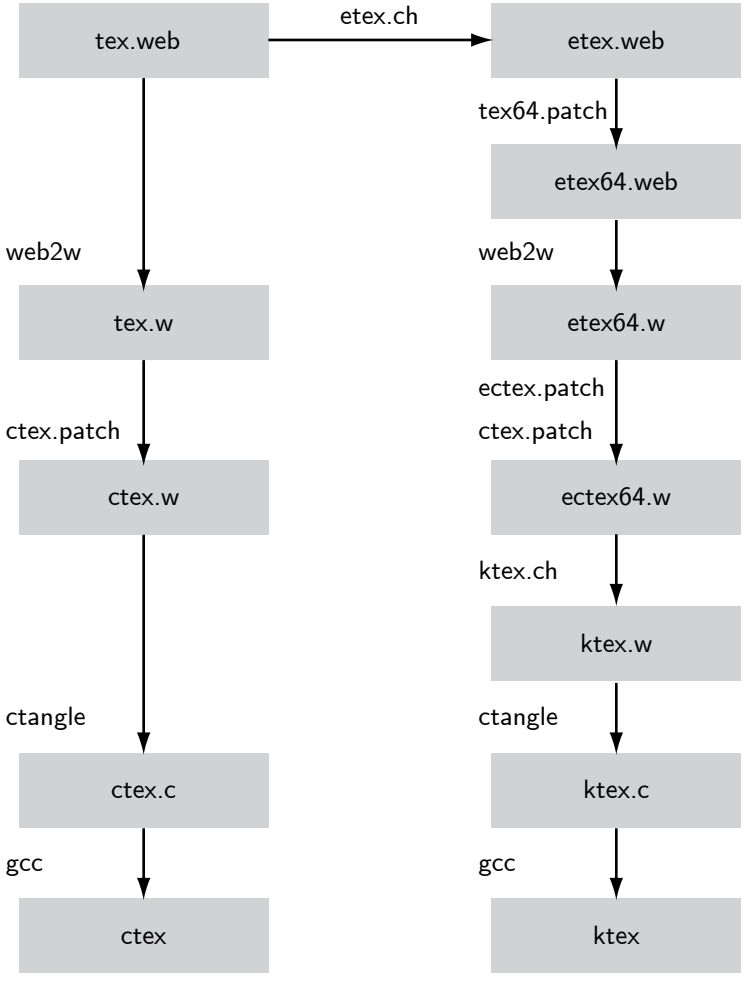

Fig. 6: Making the programs `ctex` *and* `ktex`.

3 Converting WEB to cweb

web2w is implemented by a C code file:

```
#include <stdlib.h>                                                    (2)
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdbool.h>
#include <stdint.h>
#include <limits.h>
#include <math.h>
#include "web2w.h"
#include "pascal.tab.h"
  ⟨ internal declarations 11 ⟩
  ⟨ global variables 12 ⟩
  ⟨ auxiliary functions 68 ⟩
  ⟨ functions 14 ⟩
  int main(int argc, char *argv[])
  { ⟨ process the command line 220 ⟩
    ⟨ read the WEB 5 ⟩
    ⟨ parse Pascal 99 ⟩
    ⟨ generate cweb output 114 ⟩
    ⟨ show summary 13 ⟩
    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 scanner and parser.

```
⟨ web2w.h 3 ⟩ ≡                                                        (3)
  ⟨ external declarations 4 ⟩
```

4 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 *ww_lex* from the file web.l.

⟨ external declarations 4 ⟩ ≡ (4)
```
extern int ww_lex(void);                          /* the scanner */
extern FILE *ww_in;                               /* the scanner's input file */
extern FILE *ww_out;                              /* the scanner needs an output file */
```
Used in 3.

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

⟨ read the WEB 5 ⟩ ≡ (5)
```
⟨ initialize token list 23 ⟩
ww_lex( );
⟨ finalize token list 70 ⟩
⟨ postprocess NMACRO definitions 66 ⟩
⟨ postprocess OMACRO definitions 67 ⟩
```
Used in 2.

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 8. The functions and macros used in the action parts inside the file, however, are described below.

4.1 Scanning the WEB

The scanner is written following the WEB User Manual[2].

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

⟨ external declarations 4 ⟩ +≡ (6)
```
#define TEX INITIAL
```

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, NAME, or CONTROL 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; the NAME mode is used to scan module names; and the CONTROL mode is used to scan a variety of WEB control codes.

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 4.6) 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.

4.2 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 of the parse tree upwards to internal nodes and further upwards until possibly reaching the root of the parse tree.

```
⟨ external declarations 4 ⟩ +≡                                    (7)
  typedef struct token {
    int tag;
    struct token *next, *previous, *link, *up;
    union { ⟨ leaf node 8 ⟩; ⟨ internal node 100 ⟩; };
  } token;
```

Tokens that are leaf nodes 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 8 ⟩ ≡                                                    (8)
  struct {
    int sequence_no;
    int line_no;
    ⟨ token specific info 9 ⟩
  }                                                                  Used in 7.
```

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 9 ⟩ ≡                                          (9)
  char *text;                                                        Used in 8.
```

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 4 ⟩ +≡                                       (10)
  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 11 ⟩ ≡                                       (11)
#define MAX_TOKEN_MEM 250000                                         Used in 2.
```

```
⟨ global variables 12 ⟩ ≡                                            (12)
  static token token_mem[MAX_TOKEN_MEM] = {{0}};
  static int free_tokens = MAX_TOKEN_MEM;                            Used in 2.
```

```
⟨ show summary 13 ⟩ ≡                                                (13)
  DBG(dbgbasic, "free␣tokens␣=␣%d\n", free_tokens);                  Used in 2.
```

```
⟨ functions 14 ⟩ ≡                                                   (14)
  static token *new_token(int tag)
  { token *n;
    if (free_tokens > 0) n = &token_mem[--free_tokens];
    else ERROR("token␣mem␣overflow");
    n→line_no = ww_lineno; n→sequence_no = sequence_no; n→tag = tag;
    return n;
  }                                                                  Used in 2.
```

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

```
⟨ external declarations 4 ⟩ +≡                                       (15)
  extern int ww_lineno;
```

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

⟨ global variables 12 ⟩ +≡ (16)
```
int sequence_no = 0;
```

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

⟨ external declarations 4 ⟩ +≡ (17)
```
extern int sequence_no;
#define SEQ (sequence_no++)
```

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

⟨ external declarations 4 ⟩ +≡ (18)
```
void seq(token *t, token *s);
```

⟨ functions 14 ⟩ +≡ (19)
```
void seq(token *t, token *s)
{ CHECK(t→sequence_no ≡ s→sequence_no, "tokens␣in␣line␣%d␣"
  "and␣%d␣belong␣to␣different␣code␣sequences", t→line_no, s→line_no);
}
```

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

⟨ external declarations 4 ⟩ +≡ (20)
```
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 12 ⟩ +≡ (21)
```
static token *first_token;
token *last_token;
```

⟨ external declarations 4 ⟩ +≡ (22)
```
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 23 ⟩ ≡ (23)
```
first_token = last_token = new_token(HEAD);
first_token→text = "";
```
Used in 5.

⟨ functions 14 ⟩ +≡ (24)
```
token *add_token(int tag)
{ token *n = new_token(tag);
  last_token→next = n; n→previous = last_token; last_token = n; return n;
}
```

4.3 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 4 ⟩ +≡ (25)
#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 *ww_text* 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 4 ⟩ +≡ (26)
#define COPY *copy_string*(*ww_text*)

Slightly more complex is the handling of `WEB`'s control codes to produce index entries or to insert verbatim text. Inside the control text the control sequence `\AT!` defined in `webmac.tex` must be replaced by the control sequence `\AT` defined in `cwebmac.tex`. To handle this translation, the scanner switches to `CONTROL` mode when processing the control text. At the end of the control text, the scanner switches back to its previous mode. If the control code was encountered in `TEX` mode no new token needs to be generated. The control text is just added to the current TEX text. In `PASCAL` or `MIDDLE` mode, however, the control text must be packaged as a new control token. This is achieved using the following macros:

⟨ external declarations 4 ⟩ +≡ (27)
#define CTL *pre_ctl_mode* = YY_START; BEGIN(CONTROL)
#define END_CTL
  if (*pre_ctl_mode* ≠ TEX) TOK(*end_string*( ), ATCTL);
  BEGIN(*pre_ctl_mode*)

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 4 ⟩ +≡ (28)
#define BOS *new_string*( )
#define ADD *add_string*(*ww_text*)
#define EOS (*string_length*( ) > 0 ? TOK(*end_string*( ), TEXT) : 0)

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

4.4 Strings

In this section, I define the following functions:

```
⟨ external declarations 4 ⟩ +≡                                    (29)
  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. The scanner can decide when to start a new string by calling *new_string*; when the scanner has identified a string, it can add it to the current string using *add_string*; and when the string is ready for permanent storage, it calls *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 11 ⟩ +≡                                   (30)
#define MAX_STRING_MEM 800000
```

```
⟨ global variables 12 ⟩ +≡                                        (31)
  static char string_mem[MAX_STRING_MEM];
  static int free_strings = MAX_STRING_MEM;
  static int current_string = 0;
```

```
⟨ show summary 13 ⟩ +≡                                            (32)
  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 14 ⟩ +≡                                               (33)
  char *new_string(void)
  { current_string = MAX_STRING_MEM − free_strings;
    return string_mem + current_string;
  }
  static void add_char(char c)
  { if (free_strings > 0) string_mem[MAX_STRING_MEM − free_strings−−] = c;
    else ERROR("String␣memory␣overflow");
  }
  void add_string(char *str)
  { while (*str ≠ 0) add_char(*str++);
  }
```

```
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␣memory␣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; }
void flush_string(void)
{ free_strings = MAX_STRING_MEM - current_string;
  if (free_strings > MAX_STRING_MEM) ERROR("String␣memory␣underflow");
}
```

4.5 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 function *sym_no* is used to access the table using the *name* of the identifier as a key. The table stores pointers to structures called symbols.

⟨ external declarations 4 ⟩ +≡ (34)

```
typedef struct symbol {
  char *name;
  int tag;
  struct symbol *link;
  int use_count, scan_count, goto_count;
  int arity, arg_count;
  int is_string, is_int, is_label, is_global, is_extern, is_zero_based;
  long int value;
  token *type;
  token *eq;
} symbol;
extern int sym_no(char *name);
extern symbol *symbol_table[];
```

⟨ internal declarations 11 ⟩ +≡ (35)

```
#define MAX_SYMBOL_TABLE 6007                          /* a prime */
#define MAX_SYMBOLS 5200          /* about 85% of MAX_SYMBOL_TABLE */
```

⟨ global variables 12 ⟩ +≡ (36)

```
symbol *symbol_table[MAX_SYMBOL_TABLE] = {NULL};
static symbol symbols[MAX_SYMBOLS] = {{0}};
static int free_symbols = MAX_SYMBOLS;
```

⟨ show summary 13 ⟩ +≡ (37)
```
DBG(dbgbasic, "free␣symbols␣=␣%d\n", free_symbols);
```

I organize the symbol table as a hash table using double hashing as described in [5], Chapter 6.4.

⟨ functions 14 ⟩ +≡ (38)
```
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 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 = copy_string(name); symbol_table[i]→tag = ID;
  return i;
}
```

The reference to the symbol can be stored inside the token in two ways: as an index into the *symbol_table* or as a pointer to the **symbol** structure. While scanning the WEB, I will assign the symbol number (*sym_no*), and while parsing Pascal, I will add 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 9 ⟩ +≡ (39)
```
int sym_no;
struct symbol *sym_ptr;
```

This leads to the following macros:

⟨ external declarations 4 ⟩ +≡ (40)
```
#define SYM_PTR (name) symbol_table[sym_no(name)]
#define SYMBOL
  { int s = sym_no(ww_text); 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 41 ⟩ ≡ (41)
```
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 113.

In TEX, like in most programs, there are two kinds of symbols: global and local symbols. While scanning, every symbol is 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. In general, macros are always global and their properties, the *use_count* for example, must be accumulated even for local uses. But when numeric macros are used in **goto** statements to give names to the labels, these labels are of course local symbols.

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

⟨ external declarations 4 ⟩ +≡ (42)
```
  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 4 ⟩ +≡ (43)
```
  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 12 ⟩ +≡ (44)
```
#define MAX_LOCALS 50
  static int locals[MAX_LOCALS];
  static symbol *globals[MAX_LOCALS];
  static int free_locals = MAX_LOCALS;
```

⟨ functions 14 ⟩ +≡ (45)
```
  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++) {
    globals[i]→use_count = symbol_table[locals[i]]→use_count;
    globals[i]→scan_count = symbol_table[locals[i]]→scan_count;
    globals[i]→is_label = symbol_table[locals[i]]→is_label;
    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 14 ⟩ +≡                                                    (46)
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; l→use_count = g→use_count;
  l→scan_count = g→scan_count; l→is_label = g→is_label;
  symbol_table[sym_no] = l; CHECK(free_locals > 0,
      "Overflow␣of␣local␣symbols␣in␣line␣%d", t→line_no);
  free_locals−−; locals[free_locals] = sym_no; globals[free_locals] = g;
  t→sym_ptr = l;
}
```

4.6 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 latter token. This information is also useful at later stages, for example when I expand macros. Table 1 gives a list of related tokens.

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

```
⟨ global variables 12 ⟩ +≡                                             (47)
#define MAX_WW_STACK 200
static token *ww_stack[MAX_WW_STACK] = {0};
static int ww_sp = 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_top_is* can be used to test the *tag* of the token on top of the stack.

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

⟨ external declarations 4 ⟩ +≡ (48)

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

⟨ functions 14 ⟩ +≡ (49)

```
void ww_push(token *left)
{ CHECK(ww_sp < MAX_WW_STACK, "WW␣stack␣overflow");
  DBG(dbglink, "Pushing[%d]:", ww_sp);
  if (left ≠ NULL) DBG(dbglink, THE_TOKEN(left));
  ww_stack[ww_sp++] = left;
}
token *ww_pop(token *right)
{ token *left;
  CHECK(ww_sp > 0, "Mode␣stack␣underflow"); left = ww_stack[--ww_sp];
  if (left ≠ NULL) left→link = right;
  DBG(dbglink, "Popping[%d]:", ww_sp);
  if (left ≠ NULL) DBG(dbglink, THE_TOKEN(left));
  return left;
}
```

```
int ww_top_is(int tag)
{ return ww_sp > 0 ∧ ww_stack[ww_sp − 1] ≠ NULL ∧
        ww_stack[ww_sp − 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. While the *link* field usually points in "forward" direction, the *link* field of the "}" token points back to the "{" token. This is useful for inserting symbols before a possible comment instead of after it. For an example see section 6.9.

```
⟨ external declarations 4 ⟩ +≡                                    (50)
#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), last_token→link = POP)
#define POP_PLEFT
  (EOS, TOK("}", RIGHT), BEGIN(PASCAL), last_token→link = POP)
#define POP_LEFT (ww_top_is(MLEFT) ? POP_MLEFT : (ww_top_is(PLEFT) ?
      POP_PLEFT : POP_NULL))
```

4.7 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 after token n to the name after token m. It returns a negative value for "before"; zero for "equal"; and a positive value for "after" in alphabetic order. After m there is always a full module name; the name after n might end abruptly with an ellipsis.

```
⟨ functions 14 ⟩ +≡                                                (51)
  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 11 ⟩ +≡                                    (52)
#define MAX_MODULE_TABLE 1000

⟨ global variables 12 ⟩ +≡                                         (53)
  static module module_table[MAX_MODULE_TABLE] = {{0}};
  static int free_modules = MAX_MODULE_TABLE;
  static module *module_root = NULL;

⟨ external declarations 4 ⟩ +≡                                     (54)
  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 13 ⟩ +≡                                             (55)
  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 precedes the module name. The function will allocate a new module if needed.

```
⟨ functions 14 ⟩ +≡                                                (56)
  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 14 ⟩ +≡ (57)
  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 by calling *find_module* to cover the case that this is the first and only complete occurrence of the module name.

```
⟨ external declarations 4 ⟩ +≡ (58)
#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 4 ⟩ +≡ (59)
  extern token *program;
#define PROGRAM
  (program→link = last_token, program = last_token), TOK("", EQ)
```

I use the first token as list head.

```
⟨ global variables 12 ⟩ +≡ (60)
  token *program;
```

```
⟨ initialize token list 23 ⟩ +≡ (61)
  program = first_token;
```

4.8 Definitions

In a WEB file, the token "@d" introduces the definition of a numeric constant or a macro with or without parameter. When the scanner encounters such a token, it enters the DEFINITION mode. Similar, the token "@f" introduces a format specification switching the scanner to FORMAT mode. In FORMAT mode, it scans tokens until the first newline character brings the scanner back to MIDDLE mode.

In DEFINITION mode, the first token 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 4 ⟩ +≡                                    (62)
#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))
#define CHGSNO (t, x) ((t)→sym_no = (x)→sym_no)
```

After scanning an "==" token, I know that I have either an ordinary macro or a parametrized macro. A PARAM token tells the difference. I keep track of all macro definitions in three lists:

```
⟨ global variables 12 ⟩ +≡                                        (63)
  static symbol *omacros = NULL, **omacro_tail = &omacros,
      *pmacros = NULL, **pmacro_tail = &pmacros, *nmacros = NULL,
      **nmacro_tail = &nmacros;
```

After scanning, the variables *nmacros*, *omacros*, and *pmacros* are pointing to the lists of all numeric, ordinary, respectively parametric macros, linked together by the *link* field in the symbol table. The lists are used to postprocess the definitions.

```
⟨ functions 14 ⟩ +≡                                               (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;
    if (tag ≡ NMACRO)
    { *nmacro_tail = SYM(id); nmacro_tail = &(SYM(id)→link); }
    else if (tag ≡ OMACRO)
    { *omacro_tail = SYM(id); omacro_tail = &(SYM(id)→link); }
    else if (tag ≡ PMACRO)
```

```
    { *pmacro_tail = SYM(id); pmacro_tail = &(SYM(id)→link); }
    DBG(dbgexpand, "Defining␣%s:␣%s\n", tagname(tag), SYM(id)→name);
  }
```

⟨ external declarations 4 ⟩ +≡ (65)

```
  extern void def_macro(token *eq, int tag);
#define DEF_MACRO (tag) def_macro(last_token, tag)
```

After all macro definitions have been recorded, we apply some postprocessing.

WEB will evaluate numeric macros and insert the computed value in the generated code. When converting such a macro to a C macro it might be necessary to put parentheses around the replacement text to ensure the correct evaluation. Just think of defining two numeric macros x as $2 + 2$ and y as $-x$; then you want y to have the value -4 not 0.

⟨ postprocess NMACRO definitions 66 ⟩ ≡ (66)

```
  { symbol *s;
    for (s = nmacros; s ≠ NULL; s = s→link) { token *t, *start, *end;
      t = s→eq→next;
      while (¬is_pascal(t)) t = t→next;
      start = t;
      if (t→tag ≡ PINTEGER ∨ t→tag ≡ OCTAL ∨ t→tag ≡ HEX ∨ t→tag ≡
          PCHAR ∨ t→tag ≡ PSTRING ∨ t→tag ≡ CHAR ∨ t→tag ≡
          STRING ∨ t→tag ≡ NMACRO) t = t→next;
      end = NULL;
      while (t ≠ s→eq→link) {
        if (t→tag ≡ MLEFT) t = t→link;
        else if (is_pascal(t)) end = t;
        t = t→next;
      }
      if (end ≠ NULL) { winsert_after(start→previous, POPEN, "(");
        winsert_after(end, PCLOSE, ")");
        DBG(dbgmacro,
            "Adding␣parentheses␣for␣numeric␣macro␣%s␣in␣line␣%d\n",
            s→name, s→eq→line_no);
      }
    }
  }
```

Used in 5.

Some of TEX's and ε-TEX's ordinary macros should be converted to numeric macros mainly because of the special way `web2w` treats case labels (see section 6.16). Because WEB uses for ordinary macros the same text replacement mechanism that C uses for macros, no parentheses are needed around the converted macros.

```
⟨ postprocess OMACRO definitions 67 ⟩ ≡                                    (67)
  { symbol *s;
    for (s = omacros; s ≠ NULL; s = s→link) { bool convertible = false;
      token *t;
      t = s→eq→next;
      if (t→tag ≡ PPLUS ∨ t→tag ≡ PMINUS) t = t→next;
      if (t→tag ≡ PINTEGER ∨ t→tag ≡ OCTAL ∨ t→tag ≡ HEX ∨ t→tag ≡
            NMACRO ∨ (t→tag ≡ OMACRO ∧ SYM(t)→tag ≡ NMACRO)) {
        convertible = true;
        for (t = t→next; convertible ∧ t ≠ s→eq→link; t = t→next)
          if (¬is_pascal(t)) continue;
          else if (t→tag ≡ MLEFT) t = t→link;
          else convertible = false;
      }
      if (convertible)
        { t = s→eq→previous; t→tag = NMACRO; s→tag = NMACRO;
        DBG(dbgmacro, "Converting␣%s␣from␣ordinary␣to␣numeric␣\
            macro␣in␣line␣%d\n", s→name, t→line_no);
      }
    }
  }                                                                  Used in 5.
```

Also parametrized macros need postprocessing to identify tail calls and adjust use counts and scan counts. In a macro definition, the link of the equal sign points to the token that ends the definition. To find a tail call, we start with the last token that belongs to the macro and scan backward to the last Pascal token. Instead of checking for all possible Pascal tokens, I check for those non Pascal tokens that I want to skip. If the last Pascal token is a PMACRO token with $arg\_count > 0$, I have found a tail call. In this case, the function *tail_call* returns a pointer to the final PMACRO; otherwise it returns a pointer to the token after the last Pascal token. To skip a Pascal comment, we use the link of its right brace to its left brace.

```
⟨ auxiliary functions 68 ⟩ ≡                                               (68)
  int is_pascal(token *t)
  { switch (t→tag) {
      case NL: case INDENT: case ATPLUS: case ATSLASH: case ATHASH:
      case ATBAR: case ATCTL: case METACOMMENT: case CIGNORE: case
          WGUBED: case WTINI: case WTATS: case ATCOMMA:
      case ATBACKSLASH: case ATSEMICOLON: return 0;
    default: return 1;
    }
  }
  static token *wback(token *t)
  { while (true) { t = t→previous;
      if (t→tag ≡ RIGHT) t = t→link;
      else if (is_pascal(t)) return t;
```

```
    }
  }
  token *tail_call(token *eq)
  { token *p;
    p = wback(eq→link);
    if (p→tag ≡ PMACRO) return p;
    else return p→next;
  }
```
Used in 2.

Eliminating a macro tail call possibly renders a PMACRO unused and ready for elimination itself. So after parsing, we traverse the PMACROs, determine their tail calls, and reduce if necessary the use- and scan counts. For later use, we store the pointer to the tail call in the *type* field of the symbol table.

```
⟨ postprocess PMACRO definitions 69 ⟩ ≡
  { symbol *s;
    for (s = pmacros; s ≠ NULL; s = s→link)
      if (s→arity < s→arg_count) { token *t = tail_call(s→eq);
        s→type = t;
        if (t→tag ≡ PMACRO)
          { DBG(dbgmacro, "Tail call of %s(%d,%d) found in macro \
              %s(%d) in line %d\n", SYM(t)→name, SYM(t)→use_count,
              SYM(t)→scan_count, s→name, s→use_count, t→line_no);
            SYM(t)→use_count -= s→use_count; SYM(t)→scan_count--;
          }
      }
  }
```
(69) Used in 99.

4.9 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. Further, i mark the main program as **extern** and take the opportunity to change the directory separator for the *TEX_area* and the *TEX_font_area*.

```
⟨ finalize token list 70 ⟩ ≡
  TOK("", ATP); PROGRAM; PUSH;
  TOK("", PEOF); TOK("", WEBEOF); POP; SYM_PTR("main")→is_extern = 1;
  SYM_PTR("\"TeXinputs:\"")→name = "\"TeXinputs/\"";
  SYM_PTR("\"TeXfonts:\"")→name = "\"TeXfonts/\"";
```
(70) Used in 5.

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

```
⟨ finalize token list 70 ⟩ +≡
  if (debugflags & dbgtoken) { token *t = first_token;
```
(71)

```
    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␣(%s)\n", i, symbols[i].name,
          tagname(symbols[i].tag));
  }
```

5 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 simple. 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 6, 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 *pp_lex*. In addition, the parser expects a function *pp_error* to produce error messages.

```
⟨ external declarations 4 ⟩ +≡                                    (72)
  extern int pp_lex(void);
  extern void pp_error(const char *message);
```

The function *pp_error* is very simple:

```
⟨ functions 14 ⟩ +≡                                                (73)
  void pp_error(const char *message)
  { ERROR("Parse␣error␣in␣line␣%d:␣%s", pp_lval→line_no, message); }
```

5.1 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. The stack is in the array *pp_stack* and is accessed using *pp_sp* as a stack pointer. It grows from $pp\_sp \equiv$ `MAX_PPSTACK`, the empty stack, down to $pp\_sp \equiv 0$, a full stack. I add an extra entry to the *pp_stack* array to make $pp\_sp \equiv$ `MAX_PPSTACK` a valid index into the array. This avoids the test for an empty stack in the macro `DBGTOKS`.

```
⟨ global variables 12 ⟩ +≡                                         (74)
#define MAX_PPSTACK 40
  static struct {
    token *next, *end, *link;
    int environment;
```

```
  token *parameter;
} pp_stack[MAX_PPSTACK + 1] = {{0}};
static int pp_sp = MAX_PPSTACK;
```

In each stack entry, *next* points to the next token and *end* 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 14 ⟩ +≡ (75)

```
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␣pp_lex[%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␣pp_lex[%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 *pp_lex* 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, *pp_lval*, the semantic value of the token, is the token pointer itself.

```
⟨ functions 14 ⟩ +≡ (76)
  int pp_lex(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 95 ⟩
        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 78 ⟩
        default: ERROR("Unexpected␣token␣in␣pp_lex:"THE_TOKEN(t));
        }
      }
    }
  found:
    DBG(dbgpascal, "pp_lex:␣%s->\t", tagname(tag));
    DBG(dbgpascal, THE_TOKEN(t));
    if (pp_out ≠ NULL) fprintf(pp_out, "%s␣", token2string(t));
    t→pp_sp = pp_sp; pp_lval = t; return tag;
  }
```

In the above procedure, we record the nesting level of the expansion stack in the token before we assign it to *pp_lval* this will help us to avoid some unfortunate placements of case labels in section 6.16.

```
⟨ token specific info 9 ⟩ +≡ (77)
  int pp_sp;
```

5.2 Simple cases for the parser

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

```
⟨ special treatment for WEB tokens 78 ⟩ ≡ (78)
case NL: case INDENT:
  if (pp_out ≠ NULL) fprintf(pp_out, "%s", token2string(t));
case METACOMMENT: case ATCTL: case ATEX: case ATQM: case ATPLUS:
  case ATSLASH: case ATBACKSLASH: case ATBAR: case ATHASH: case
      ATCOMMA: case ATAND: case ATSEMICOLON: case ATLEFT: case
      ATRIGHT:
  t = t→next; continue;
```

Used in 76.

Comments can be skipped in a single step:

⟨ special treatment for WEB tokens 78 ⟩ +≡ (79)
```
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 78 ⟩ +≡ (80)
```
case PEOF: pp_stack[pp_sp].next = t→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 78 ⟩ +≡ (81)
```
case ATDOLLAR: case OCTAL: case HEX: case CHAR:
  pp_stack[pp_sp].next = t→next; tag = PINTEGER; goto found;
```

The last simple case is an identifier. 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* ≡ ID the default tag PID and link tokens to the actual symbol structure, which might be local or global.

⟨ special treatment for WEB tokens 78 ⟩ +≡ (82)
```
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;
  }
```

The parser will increment the *use_count* of the symbol, depending on the usage of it. ID's that are defined but never used will have a *use_count* of zero and can be eliminated.

⟨ external declarations 4 ⟩ +≡ (83)
```
#define USE (T) (T)→sym_ptr→use_count++;
#define USE_NMACRO (T) USE(T); propagate_use((T)→sym_ptr);
```

When a numeric macro is used, also its defining expression gets used. So we have to propagate the use count.

⟨ functions 14 ⟩ +≡ (84)
```
void propagate_use(symbol *s)
{ token *t;
  if (s→use_count > 1) return;
  for (t = s→eq→next; t ≠ s→eq→link; t = t→next)
    if (t→tag ≡ NMACRO) { SYM(t)→use_count++; propagate_use(SYM(t));
    }
}
```

⟨ external declarations 4 ⟩ +≡ (85)
extern void *propagate_use*(**symbol** ∗*s*);

A few macros are actually never used nor are they mentioned somewhere in the explanatory text. Still, I want to keep their definition as part of the documentation. So I mark them as used.

⟨ finalize token list 70 ⟩ +≡ (86)
SYM_PTR("below_display_skip")→*use_count* = 1;
SYM_PTR("below_display_short_skip")→*use_count* = 1;
SYM_PTR("top_skip")→*use_count* = 1;
SYM_PTR("tab_skip")→*use_count* = 1;
SYM_PTR("thin_mu_skip")→*use_count* = 1;
SYM_PTR("med_mu_skip")→*use_count* = 1;
SYM_PTR("thick_mu_skip")→*use_count* = 1;
SYM_PTR("toks")→*use_count* = 1;
SYM_PTR("output_penalty")→*use_count* = 1;
SYM_PTR("TeX_banner")→*use_count* = 1; /∗ ϵ-TEX ∗/

5.3 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 78 ⟩ +≡ (87)
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 88 ⟩ ≡                                    (88)
case WDEBUG:
  if (t→previous→tag ≡ BAR) wputs(t→text);
  else { wprint_pre("#ifdef␣@!DEBUG");
  }
  t = t→next; break;
case WINIT:
  if (t→previous→tag ≡ BAR) wputs(t→text);
  else { wprint_pre("#ifdef␣@!INIT");
  }
  t = t→next; break;
case WSTAT:
  if (t→previous→tag ≡ BAR) wputs(t→text);
  else { wprint_pre("#ifdef␣@!STAT");
  }
  t = t→next; break;
case WGUBED: case WTINI: case WTATS:
  if (t→previous→tag ≡ BAR) wputs(t→text);
  else { wprint_pre("#endif");
  }
  t = t→next;
  if (t→tag ≡ ATPLUS ∨ t→tag ≡ ATSLASH) t = t→next;
  if (t→tag ≡ NL) t = t→next;
  break;                                                  Used in 115.
```

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 88 ⟩ +≡                                   (89)
case ATPLUS: t = t→next;
  if (¬following_directive(t)) wputs("@+");
  else DBG(dbgcweb, "Eliminating␣@+␣in␣line␣%d\n", t→line_no); break;
```

5.4 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, the parser will increment the *use_count* 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 78 ⟩ +≡                             (90)
case NMACRO: t→sym_ptr = SYM(t);
  if (t→sym_ptr→eq→next→tag ≡ STRING) {
    token *s = t→sym_ptr→eq→next;
```

```
    if (s→sym_ptr ≠ NULL) { s→sym_ptr = SYM(s);
      DBG(dbgstring, "Using␣numeric␣macro␣%s␣␣in␣line␣%d\n",
          s→sym_ptr→name, t→line_no);
    }
  }
  pp_stack[pp_sp].next = t→next; goto found;
case STRING:
  if (t→sym_no ≠ 0) { t→sym_ptr = SYM(t);
  }
  DBG(dbgstring, "Using␣string␣%s␣in␣line␣%d\n", t→text, t→line_no);
  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 4 ⟩ +≡ (91)

```
extern long int getval(token *t);
```

⟨ functions 14 ⟩ +≡ (92)

```
long int getval(token *t)
{ long int n = 0;
  switch (t→tag) {
  case ATDOLLAR: n = 0; break;
  case PINTEGER: n = strtol(t→text, NULL, 10); break;
  case OMACRO:
    if (SYM(t)→tag ≠ NMACRO)
      ERROR("Unable␣to␣get␣value␣for␣OMACRO␣in␣line␣%d", t→line_no);
  case NMACRO: t = SYM(t)→eq; CHECK(t→tag ≡ EQEQ,
      "=␣expected␣in␣numeric␣macro␣in␣line␣%d", t→line_no);
    t = t→next;
    if (t→tag ≡ POPEN) t = t→next;
    if (t→tag ≡ PMINUS) { t = t→next; n = −getval(t);
    }
    else if (t→tag ≡ PPLUS) { 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→line_no);
    }
    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 4 ⟩ +≡ (93)

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

5.5 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 78 ⟩ +≡ (94)

```
case ATLESS:
  { token *eq, *atgreater;
    atgreater = find_module(t)→atgreater;
    CHECK(atgreater ≠ NULL, "Undefined module @<%s...@> in line %d",
        token2string(t→next), t→line_no);
    DBG(dbgexpand, "Expanding module @<%s@> in line %d\n",
        token2string(t→next), t→line_no); 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 95 ⟩ ≡ (95)

```
  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 76.

Slightly simpler are ordinary macros. Before we expand them, however, we check if they have been converted to numeric macros.

⟨ special treatment for WEB tokens 78 ⟩ +≡ (96)

```
case OMACRO:
  if (SYM(t)→tag ≡ NMACRO) { tag = NMACRO; t→sym_ptr = SYM(t);
    pp_stack[pp_sp].next = t→next; goto found;
  }
  else { token *eq;
    eq = SYM(t)→eq; SYM(t)→use_count++; pp_stack[pp_sp].next = t→next;
    DBG(dbgexpand, "Expanding␣ordinary␣macro␣%s␣in␣line␣%d\n",
        token2string(t), t→line_no);
    t = pp_push(NULL, eq, 0, NULL); continue;
  }
```

5.6 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)`, where `font == type` and `type(#)=mem[#].hh.b0`, I call *pp_lex* 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 *pp_lex* will reach the end of the expansion, pop the stack, and then find the "(" token.)

⟨ special treatment for WEB tokens 78 ⟩ +≡ (97)

```
case PMACRO:
  { token *open, *eq;
    int popen;
    DBG(dbgexpand, "Expanding␣parameter␣macro␣%s␣in␣line␣%d\n",
        token2string(t), t→line_no);
    eq = SYM(t)→eq; SYM(t)→use_count++; pp_stack[pp_sp].next = t→next;
    popen = pp_lex( );
    CHECK(popen ≡ POPEN, "expected␣(␣after␣macro␣with␣parameter");
    open = pp_lval; pp_stack[pp_sp].next = open→link→next; ⟨ count macro
        parameters 148 ⟩
    t = pp_push(NULL, eq, pp_sp, open); continue;
  }
```

Whenever a parametrized macro gets expanded, I also count the number of its parameters storing it in the symbol table. It will help me in section 6.9 to convert WEB macros to nice C macros.

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 78 ⟩ +≡                                (98)
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;
  }
```

5.7 The function *pp_parse*

The function *pp_parse* 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 *pp_parse* which I call after initializing the *pp_stack* in preparation for the first call to *pp_lex*.

```
⟨ parse Pascal 99 ⟩ ≡                                                     (99)
  program = first_token->link; pp_push(program->link, program->next, 0, NULL);
  pp_parse(); ⟨ postprocess PMACRO definitions 69 ⟩                  Used in 2.
```

The function *pp_parse* 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 100 ⟩ ≡                                                   (100)
  struct {
    long int value;
  }                                                                  Used in 7.
```

Internal nodes are constructed using the function *join*.

```
⟨ external declarations 4 ⟩ +≡                                            (101)
  token *join(int tag, token *left, token *right, long int value);

⟨ functions 14 ⟩ +≡                                                       (102)
  token *join(int tag, token *left, token *right, long int value)
  { token *n = new_token(tag);
    n->previous = left; n->next = right; n->value = value;
    DBG(dbgjoin, "tree:␣"); DBGTREE(dbgjoin, n); return n;
  }
```

5.8 Pascal's predefined symbols

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

⟨ initialize token list 23 ⟩ +≡ (103)

```
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);
```

⟨ functions 14 ⟩ +≡ (104)

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

6 Writing the cweb

6.1 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 members of the family *wprint_int*, *wprint_str*, and *wprint_pre*. 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 *columns* counts the characters on the current line. The variable *spaces* counts spaces that still need to be written; these spaces are suppressed at the end of a line.

⟨ global variables 12 ⟩ +≡ (105)

```
static int alfanum = 0;
static int comma = 0;
static int columns = 0;
static int spaces = 0;
```

The low-level output function is *wput*. It counts spaces and columns; it writes newlines resetting the space and column count; and it writes any other character after writing the delayed spaces. It sets the indicators for trailing commas or alphanumeric characters. The function *wputs* writes a complete string of characters using *wput*.

⟨ auxiliary functions 68 ⟩ +≡ (106)

```
static void wput(char c)
{ alfanum = comma = false;
  if (c ≡ '␣') spaces++;
  else if (c ≡ '\n') fputc('\n', w), columns = spaces = 0;
  else { ⟨ output spaces 107 ⟩
    fputc(c, w), columns++; alfanum = isalnum(c); comma = c ≡ ',';
  }
}
static void wputs(char *str)
{ while (*str ≠ 0) wput(*str++); }
```

If it is known that the next character is neither a space nor a newline, the delayed spaces are added to the output.

⟨ output spaces 107 ⟩ ≡ (107)
```
if (spaces > 0) { alfanum = comma = false;
  do fputc('␣', w), spaces−−, columns++; while (0 < spaces);
}
```
Used in 106 and 110.

There are three higher level output functions. The most common function is *wprint* which is used to output C tokens and takes care of inserting spaces when necessary to separate tokens and—for a nicer looking output—after commas. Note that *alfanum* or *comma* imply *spaces* ≡ 0 except after ⟨ separate tokens ⟩.

⟨ separate tokens 108 ⟩ ≡ (108)
```
if (alfanum ∨ comma) spaces++;
```
Used in 109, 110, 131, and 155.

⟨ auxiliary functions 68 ⟩ +≡ (109)
```
static void wprint(char *str)
{ if (isalnum(str[0])) ⟨ separate tokens 108 ⟩
  wputs(str);
}
```

The *wprint_int* function uses *fprintf* for convenience. No test is necessary to tell that its output starts and ends with a digit.

⟨ auxiliary functions 68 ⟩ +≡ (110)
```
static void wprint_int(int i)
{ ⟨ separate tokens 108 ⟩
  ⟨ output spaces 107 ⟩
  columns += fprintf(w, "%d", i); alfanum = true; comma = false;
}
```

The *wprint_str* function will escape special characters according to the rules of C.

⟨ auxiliary functions 68 ⟩ +≡ (111)
```
static void wprint_str(char *str)
{ 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] ≡ '"') wputs("\\\"");
    else wput(str[0]);
    str++;
  }
}
```

When we output a preprocessor directive, it should start at the beginning of a line, but we may want to keep the indentation, as given by *spaces*, that the code had previously.

```
⟨ auxiliary functions 68 ⟩ +≡ (112)
  static void wprint_pre(char *str)
  { if (columns ≠ 0) wput('\n');
    while (*str ≠ 0) fputc(*str++, w);
    fputc('\n', w); columns = 0; alfanum = comma = false;
  }
```

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

```
⟨ auxiliary functions 68 ⟩ +≡ (113)
  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 41 ⟩
    }
  }
```

6.2 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 114 ⟩ ≡ (114)
  ⟨ rename reserved words 133 ⟩
  wprint_to(first_token, last_token); ⟨ generate a header section if requested 227
      ⟩
```
Used in 2.

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 14 ⟩ +≡ (115)
  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→line_no);
    switch (t→tag)
```

```
  { ⟨ convert t from WEB to cweb 88 ⟩
    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.

```
⟨ auxiliary functions 68 ⟩ +≡ (116)
  static token *wtoken(token *t);
  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;
  }
```

6.3 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 88 ⟩ +≡ (117)
case CIGNORE: case CTLOCAL: case PLABEL: 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 4 ⟩ +≡ (118)
#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 88 ⟩ +≡ (119)
case PFORWARD:
  if (t→next→tag ≡ PSEMICOLON) wput(';'), t = t→next→next;
  else wprint("forward"), t = t→next;          /* as in |forward| */
  if (t→tag ≡ ATCTL) t = t→next;
  break;
```

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

```
⟨ convert t from WEB to cweb 88 ⟩ +≡ (120)
case METACOMMENT:
  { char *c;
```

```
    wputs("␣/*");
    for (c = t→text + 2; c[0] ≠ '@' ∨ c[1] ≠ '}'; c++) wput(*c);
    wputs("*/"); t = t→next;
  }
  break;
case ATLEFT: wprint_pre("#if␣0"); t = t→next; break;
case ATRIGHT: wprint_pre("#endif"); 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 88 ⟩ +≡                                    (121)
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→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;
```

The PROGRAM statement of Pascal is no longer needed.

```
⟨ convert t from WEB to cweb 88 ⟩ +≡                                    (122)
case PPROGRAM:
  if (t→link ≠ NULL) t = t→link→next;
  else wputs(t→text), 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 field to connect key tokens of certain Pascal constructions. For example, the parser links the PPROGRAM token to the PSEMICOLON that ends

Pascal's program heading. Using these links, I can find the different parts (including the intervening WEB tokens) and rearrange or skip them as needed. The above example also demonstrates that extra care is needed before using the *link* field: When the identifier *program* occurs as part of the documentation, it is not parsed and its *link* will be NULL.

Linking tokens is achieved with the following macro which also checks that the link stays within the same code sequence.

⟨ external declarations 4 ⟩ +≡ (123)
#**define** LNK (*from*, *to*) ((*from*) ? (*seq*((*from*), (*to*)), (*from*)→*link* = (*to*)) : 0)

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 88 ⟩ +≡ (124)
case PBEGIN: *wput*('{'), *t* = *t*→*next*;
  if (¬*following_directive*(*t*)) *wputs*("@+");
  break;

⟨ auxiliary functions 68 ⟩ +≡ (125)
  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 88 ⟩ +≡ (126)
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 " .. ".

6.4 Pascal division

In some cases build-in functions of Pascal can be replaced by suitably defined macros in C 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 "$X/(\textbf{double})(Y)$" 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 12 ⟩ +≡ (127)
```
static int float_constant_no;
```

⟨ initialize token list 23 ⟩ +≡ (128)
```
float_constant_no = predefine("float_constant", ID, 0);
```

⟨ convert t from WEB to cweb 88 ⟩ +≡ (129)
```
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→line_no);
  }
  t = t→next; break;
```

6.5 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 88 ⟩ +≡ (130)
```
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(dbgcweb, "Tag after @! is %s in line %d\n", tagname(t→tag),
        t→line_no);
  }
  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.

```
⟨ auxiliary functions 68 ⟩ +≡ (131)
  static token *wid(token *t)
  { ⟨ separate tokens 108 ⟩
    if (t→previous→tag ≡ ATEX) wputs("@!");
    wputs(SYM(t)→name); return t→next;
  }
```

I use this function like this:

```
⟨ convert t from WEB to cweb 88 ⟩ +≡ (132)
case ID: case PID: case NMACRO: case CINTDEF: case PFUNCID: t = wid(t);
  break;
```

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

```
⟨ rename reserved words 133 ⟩ ≡ (133)
  SYM_PTR("switch")→name = "get_cur_chr";
  SYM_PTR("continue")→name = "resume"; SYM_PTR("int")→name = "i";
  SYM_PTR("register")→name = "internal_register";
  SYM_PTR("exit")→name = "end"; SYM_PTR("free")→name = "is_free";
  SYM_PTR("write")→name = "pascal_write";
  SYM_PTR("read")→name = "pascal_read";
  SYM_PTR("close")→name = "pascal_close";
  SYM_PTR("xclause")→name = "else";
  SYM_PTR("remainder")→name = "rem";
```

Used in 114.

6.6 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 88 ⟩ +≡ (134)
case ATLESS: wputs("@<"); t = t→next; CHECK(t→tag ≡ TEXT,
        "Module␣name␣expected␣instead␣of␣%s␣in␣line␣%d",
        token2string(t), t→line_no);
  { char *str = t→text;
    do if (str[0] ≡ '@' ∧ str[1] ≡ ',') str = str + 2;        /* control codes are
          forbidden in section names */
    else if (columns > 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→line_no);
wputs("@>"); t = t→next;
if (t→tag ≡ ATSLASH) t→tag = ATSEMICOLON;
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.

6.7 Strings

Multiletter Pascal strings are translated to C strings. Note that the parser occasionally converts CHAR tokens to PSTRING tokens using the *pchar2string* function. Single character Pascal or WEB strings are converted to C character constants.

```
⟨ convert t from WEB to cweb 88 ⟩ +≡                                    (135)
case PSTRING: wprint_str(t→text); t = t→next; break;
case CHAR: case PCHAR: wput('\'');
  switch (t→text[1]) {
  case '\'': wputs("\\'"); break;
  case '\\': wputs("\\\\"); break;
  case '@': wputs("@@"); break;
  default: wput(t→text[1]); break;
  }
  wput('\''); t = t→next; break;
```

6.8 Replacing the WEB string pool file

Multiletter WEB strings need more work because I have to replace the WEB string pool file. But let's start with the big simplification in web2w version 1.0: all multiletter WEB strings are now translated to C string literals. The string pool checksum is simply replaced by zero, because it is no longer used.

```
⟨ convert t from WEB to cweb 88 ⟩ +≡                                    (136)
case STRING: wprint_str(t→text); t = t→next; break;
case ATDOLLAR: wputs("0"); t = t→next; break;
```

The real work starts when TEX assigns WEB strings to variables or passes them as arguments to functions because you can not simply assign a string literal to a variable or function parameter of type **str_number** which is simply an integer. So it seems necessary to change the type of these variables from **str_number** to **char** ∗. But changing all of these variables is not a good solution either, because a large part of TEX still uses string numbers. Therefore web2w will convert variables to **char** ∗ type if the majority of operations involve string literals. To obtain statistical data, the Pascal parser tracks two situations: assignments to a variable and comparisons for equality with a variable. In both cases, the parser calls the function *pvar_string*. This function will increment the *is_string* counter of the variable operand if the other operand is a string and decrements it otherwise.

When web2w outputs variable definitions, it will check this counter and convert variables to type **char** ∗ if the counter is positive.

```
⟨ external declarations 4 ⟩ +≡                                    (137)
  extern void pvar_string(token *id, token *val);

⟨ functions 14 ⟩ +≡                                               (138)
  void pvar_string(token *id, token *str)
  { if (id→tag ≡ PID) {
      if (str→tag ≡ STRING ∨ (str→tag ≡ PID ∧ str→sym_ptr→is_string > 0))
        id→sym_ptr→is_string++;
      else if (str→tag ≡ PINTEGER ∧ getval(str) ≡ 0)
        ;                                          /* could go both ways */
      else id→sym_ptr→is_string−−;
      DBG(dbgstring, "Variable␣%s␣string␣(%d)␣marked␣in␣line␣%d\n",
          id→sym_ptr→name, id→sym_ptr→is_string, id→line_no);
    }
  }
```

If the parameter type has changed, the final patch file will adjust functions accordingly. For example, the function *primitive* will then enter the literal string that is now its argument into the string pool and continue with the string number obtained as before.

There are, however, a few exceptions to the general rule. The two procedures *print* and *print_esc* are used very often and sometimes the argument is a string number and sometimes a string literal. In this case, I would like to call the augmented *print* and *print_esc* functions if the argument is a string and the unchanged functions, renamed as *printn* and *printn_esc*, otherwise. This change is accomplished by calling the *pstring2n* procedure before writing a PCALLID to the output (see section 6.19). For every procedure call, the parser has put a representation of the argument list in the *up* field of the procedure name. The function *pstring2n* is then called with the *id* of the procedure and its *up* pointer.

```
⟨ auxiliary functions 68 ⟩ +≡                                     (139)
  static int is_string(token *arg)
  { if (arg→tag ≡ PSTRING ∨ arg→tag ≡ STRING) return 1;
    else if (arg→tag ≡ PID) {
      DBG(dbgstring, "Variable␣%s␣string␣(%d)␣argument␣in␣line␣%d\n",
          arg→sym_ptr→name, arg→sym_ptr→is_string, arg→line_no);
      if (arg→sym_ptr→is_string > 0) return 1;
    }
    return 0;
  }
  static void pstring2n(token *id, token *arg)
  { if (arg ≡ NULL ∨ arg→tag ≡ PCOLON ∨ arg→tag ≡ CREFID) return;
    if (id→sym_no ≡ print_no ∧ ¬is_string(arg)) id→sym_no = printn_no;
```

```
    else if (id→sym_no ≡ print_esc_no ∧ ¬is_string(arg))
      id→sym_no = printn_esc_no;
  }
```

Remember that single character strings are routinely converted to character constants. This conversion is correct for string numbers but not for literal strings. The parser will undo the conversion by calling *pchar2string* if such a single character string is passed to one of the following functions: *print*, *print_esc*, *print_nl*, and *scan_keyword*.

```
⟨ external declarations 4 ⟩ +≡                                    (140)
  void pchar2string(token *id, token *arg);

⟨ functions 14 ⟩ +≡                                               (141)
  void pchar2string(token *id, token *arg)
  { if (arg→tag ≡ CHAR ∧
          (id→sym_no ≡ print_no ∨ id→sym_no ≡ print_esc_no ∨
           id→sym_no ≡ print_nl_no ∨ id→sym_no ≡ scan_keyword_no))
      arg→tag = PSTRING;
    id→up = arg;
  }
```

To check for the respective symbols, `web2w` uses these variables:

```
⟨ global variables 12 ⟩ +≡                                        (142)
  static int print_no, printn_no, print_esc_no, printn_esc_no, print_nl_no,
      scan_keyword_no;
```

The variables are initialized like this:

```
⟨ initialize token list 23 ⟩ +≡                                   (143)
  printn_no = predefine("printn", PPROCID, 0);
  print_no = predefine("print", PPROCID, 0);
  print_esc_no = predefine("print_esc", PPROCID, 0);
  printn_esc_no = predefine("printn_esc", PPROCID, 0);
  print_nl_no = predefine("print_nl", PPROCID, 0);
  scan_keyword_no = predefine("scan_keyword", PFUNCID, 0);
```

6.9 Macro and format declarations

Before I output a macro declaration, I first check if the translation has made it obsolete, either because it just gives a name to a label or because its *use_count* and *scan_count* both are zero. In this case, I skip it. Otherwise, I output the initial part of the macro declaration converting macro names to upper case if requested. From here on, I go different routes for the different types of declarations. Format declarations follow the same schema but are simpler.

```
⟨ convert t from WEB to cweb 88 ⟩ +≡                              (144)
case ATD:
  { token *eq = t→next→next;
```

```
    if (eq→tag ≠ EQEQ) eq = eq→next;                    /* PMACRO */
    DBG(dbgcweb, "Macro␣definition␣in␣line␣%d\n", t→line_no);
    if (SYM(t→next)→is_label) {
      DBG(dbgid, "Eliminating␣label␣definition␣%s␣in␣line␣%d\n",
          SYM(t→next)→name, t→line_no); t = wskip_to(t, eq→link);
    }
    else if (SYM(t→next)→use_count ≡ 0 ∧ SYM(t→next)→scan_count ≡ 0) {
      DBG(dbgid, "Eliminating␣unused␣macro␣%s␣in␣line␣%d\n",
          SYM(t→next)→name, t→line_no); t = wskip_to(t, eq→link);
    }
    else { wputs("@d␣"), t = t→next; ⟨ convert macro names name to upper
        case if requested 222 ⟩
      wprint(SYM(t)→name);
      if (t→tag ≡ NMACRO) ⟨ convert NMACRO from WEB to cweb 145 ⟩
      else if (t→tag ≡ OMACRO) ⟨ convert OMACRO from WEB to cweb 146 ⟩
      else if (t→tag ≡ PMACRO) ⟨ convert PMACRO from WEB to cweb 152 ⟩
      else ERROR("Macro␣name␣expected␣in␣line␣%d", t→line_no);
    }
    DBG(dbgcweb, "End␣Macro␣definition␣in␣line␣%d\n", t→line_no); break;
  }
case ATF:
  { token *eq = t→next→next;
    DBG(dbgcweb, "Format␣definition␣in␣line␣%d\n", t→line_no);
    if (SYM(t→next)→scan_count ≡ 0) { DBG(dbgid,
          "eliminating␣unused␣format␣definition␣%s␣in␣line␣%d\n",
          SYM(t→next)→name, t→line_no); t = wskip_to(t, eq→link);
    }
    else { wputs("@f␣"), t = t→next; wprint(SYM(t)→name);
      t = wprint_to(eq→next, eq→link);
    }
    break;
  }
```

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, the macro postprocessing has taken care of inserting necessary parentheses. So all that needs to be done here is inserting a space.

```
⟨ convert NMACRO from WEB to cweb 145 ⟩ ≡                    (145)
  { wput('␣'); t = eq→next;
  }
```
Used in 144.

Ordinary parameterless macros usually map directly to C style macros. But if they end with a tail call of a macro that has an *arg_count* > 0 they turn into a macro with parameters.

```
⟨ convert OMACRO from WEB to cweb 146 ⟩ ≡                    (146)
  { token *p = tail_call(eq);
```

```
if (p→tag ≡ PMACRO ∧ SYM(p)→arg_count > 0 ∧ SYM(p)→arity > 0) { int
        arg_count = SYM(p)→arg_count;
  SYM(t)→arg_count = arg_count;
  SYM(t)→arity = SYM(p)→arity; t→tag = SYM(t)→tag = PMACRO;
  wput('('), wprint_args(0, arg_count), wputs(")␣");
  t = wprint_to(eq→next, p→next);
  wput('('), wprint_args(0, arg_count), wputs(")");
  }
  else { wput('␣'); t = eq→next;
  }
}
```

Used in 144.

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. If TEX uses a macro with a variable number of arguments, I set the counter to −1 and generate a variadic macro.

⟨ internal declarations 11 ⟩ +≡ (147)

```
#define VARIADIC -1
```

WEB uses a single "#" to indicate the parameter position in the replacement text. To construct macros where multiple parameters are inserted at different places, TEX uses "tail calls". An example is TEX's definition of char_info:

```
char_info_end(#)==#].qqqq
char_info(#)==font_info[char_base[#]+char_info_end
```

which is used as

```
char_info(f)(c)
```

The char_info macro ends with char_info_end, the tail call, without specifying the parameter for it. In C, I would prefer the translation

```
#define char_info(A, B) font_info[char_base[A]+B].qqqq
```

The following code is used when expanding a parametrized macro for the parser. It counts the number of parameters substituted for the # sign when the macro gets expanded—its *arity*—and the total number of parameters including those that are passed to a possible tail call—its *arg_count*.

⟨ count macro parameters 148 ⟩ ≡ (148)

```
{ token *p;
  int count;
  ⟨ determine the arity 150 ⟩
  ⟨ determine the arg_count 151 ⟩
}
```

Used in 97.

Counting the *arity* is accomplished with the *count_arity* function. Its parameter *open* points to the the OPEN token that follows after the macro name. The function determines the number of arguments, returning zero if a hash sign is found and the

number of arguments can not be determined. We assume that there are no Pascal comments inside the argument list.

```
⟨ auxiliary functions 68 ⟩ +≡                                    (149)
  int count_arity(token *open)
  { int count = 1;
    token *p;
    if (open→next→tag ≡ HASH) return 0;
    for (p = open→next; p ≠ open→link; p = p→next)
      if (p→tag ≡ PCOMMA) count++;
      else if (p→tag ≡ POPEN) p = p→link;
    return count;
  }
```

```
⟨ determine the arity 150 ⟩ ≡                                    (150)
  count = count_arity(open);
  if (count > 0) {
    if (SYM(t)→arity ≡ 0) SYM(t)→arity = count;
    else if (SYM(t)→arity ≠ count) SYM(t)→arity = VARIADIC;
  }
  DBG(dbgmacro, "Counting␣%s␣parameters:␣%d␣in␣line␣%d\n",
      SYM(t)→name, count, open→line_no);                         Used in 148.
```

To determine the *arg_count* of a PMACRO we skip its natural argument and then count the number of expressions enclosed in parentheses. Between these parentheses, we might find all kind of "non-Pascal" tokens, which we have to skip.

```
⟨ determine the arg_count 151 ⟩ ≡                                (151)
  p = open→link→next;
  while (p ≠ NULL) {
    if (p→tag ≡ POPEN) {
      if (p→link ≠ NULL ∧ p→link→tag ≡ PCLOSE)
      { int c = count_arity(p);
        if (c ≡ 0) { count = 0; break; }
        else count = count + c;
        p = p→link→next;
      }
      else ERROR(")␣expected␣matching␣(␣in␣line␣%d", p→line_no);
    }
    else if (¬is_pascal(p)) p = p→next;
    else break;
  }
  if (count > 0) {
    if (SYM(t)→arg_count ≡ 0) SYM(t)→arg_count = count;
    else if (SYM(t)→arg_count ≠ count) SYM(t)→arg_count = VARIADIC;
    DBG(dbgmacro, "Counting␣%s␣arguments:␣%d␣in␣line␣%d\n",
        SYM(t)→name, count, open→link→next→line_no);
```

```
    if (SYM(t)→arg_count > 'Z' − 'A' + 1)
      ERROR("Macro␣%s␣with␣%d>%d␣arguments␣found␣in␣line␣%d",
          SYM(t)→name, SYM(t)→arg_count, 'Z' − 'A' + 1,
          open→link→next→line_no);
  }
```

Used in 148.

Now that I know *arity* and *arg_count* of a macro, I can construct the macro definition.

⟨ convert PMACRO from WEB to cweb 152 ⟩ ≡ (152)

```
  { int arity, arg_count;
    current_macro = SYM(t); current_arg = 0;
    arity = current_macro→arity, arg_count = current_macro→arg_count;
    if (arity ≡ 0)                                  /* used only with (#) */
    { DBG(dbgmacro, "Defining␣parametrized␣macro␣%s␣without␣\
          arguments␣in␣line␣%d\n", SYM(t)→name, t→line_no);
      wputs("(X)␣"); t = eq→next;
    }
    else if (arity ≡ VARIADIC) {
      DBG(dbgmacro, "Defining␣variadic␣macro␣%s␣in␣line␣%d\n",
          SYM(t)→name, t→line_no); wput('('), wputs("..."), wputs(")␣");
      t = eq→next;
    }
    else if (arity > arg_count)
      ERROR("Macro␣%s␣with␣arity␣%d␣>␣argument␣count␣%d␣in␣line␣%d",
          SYM(t)→name, arity, arg_count, t→line_no);
    else if (arity ≡ arg_count) {
      wput('('), wprint_args(0, arg_count), wputs(")␣");
      t = wprint_to(eq→next, eq→link);
    }
    else { token *tail;
      DBG(dbgmacro, "Defining␣macro␣%s␣with␣tail␣call␣in␣line␣%d\n",
          SYM(t)→name, t→line_no);
      wput('('), wprint_args(0, arg_count), wputs(")␣");
      eq = current_macro→eq; tail = current_macro→type; t = tail→next;
      while (current_arg < arg_count) { wprint_to(eq→next, tail);
        current_arg = current_arg + current_macro→arity;
        if (tail→tag ≠ PMACRO) break;
        current_macro = SYM(tail); eq = current_macro→eq;
        tail = tail_call(eq);
      }
    }
  }
```

Used in 144.

The preceding code simply uses *wprint_to* to generate the replacement text of the macro. To be able to expand a HASH token inside the replacement text by

the right sequence of macro parameters, the necessary information is kept in two global variables:

```
⟨ global variables 12 ⟩ +≡ (153)
  static symbol *current_macro;
  static int current_arg = 0;
```

Using them the HASH token can be expanded.

```
⟨ convert t from WEB to cweb 88 ⟩ +≡ (154)
case HASH:
  if (current_macro→arity ≡ 0) wprint("X");
  else if (current_macro→arity ≡ VARIADIC) wprint("__VA_ARGS__");
  else wprint_args(current_arg, current_arg + current_macro→arity);
  t = t→next; break;
```

I have used an auxiliary function to generate the parameter lists. The following function might produce garbage if more than 26 parameters are used. It is an easy exercise to extend the function to cope with longer parameter lists, but for now we leave it to the C compiler to produce an error message.

```
⟨ auxiliary functions 68 ⟩ +≡ (155)
  void wprint_args(int from, int to)
  { while (true) { ⟨ separate tokens 108 ⟩
      wput('A' + from); from++;
      if (from ≥ to) break;
      else wput(',');
    }
  }
```

6.10 Macro calls

Macro calls may occur everywhere in the program including the replacement text of an other macro. The following code handles all such calls, replacing sequences of parameter lists by one long "flat" parameter list. Inside a parameter list, the top-level PCOMMA tokens separate the parameters. It is important to ignore additional commas that are at lower levels inside of parentheses. Between parameter lists there must be no other Pascal tokens, only non-Pascal tokens are allowed.

```
⟨ convert t from WEB to cweb 88 ⟩ +≡ (156)
case PMACRO: case OMACRO:
  { int count = SYM(t)→arg_count;
    token *macro = t;
    t = wid(t);
    if (count > 0 ∧ t→tag ≡ POPEN) { wput('(');
      while (count > 0 ∧ t→tag ≡ POPEN) { token *p = t→next;
        count--;
        while (p ≠ t→link) {
          if (p→tag ≡ PCOMMA) count--;
```

```
        else if (p→tag ≡ HASH ∧ current_macro→arity > 0)
          count = count − (current_macro→arity − 1);
        if (p→tag ≡ POPEN) p = wprint_to(p, p→link);
        else p = wtoken(p);
      }
      t = t→link;
      if (t→tag ≠ PCLOSE)
        ERROR(")␣expected␣in␣macro␣call␣in␣line␣%d", t→line_no);
      t = t→next;
      if (count > 0) { wput('(');
        while (¬is_pascal(t)) t = wtoken(t);
        if (t→tag ≠ POPEN)
          ERROR("(␣expected␣to␣continue␣macro␣%s(%d)␣par\
              ameters␣in␣line␣%d", SYM(macro)→name,
              SYM(macro)→arg_count, t→line_no);
      }
    }
    wput(')');
  }
}
break;
```

6.11 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.

In most cases the labels in TEX are numeric macros. In this case, the parser will change the tag from NMACRO to CLABEL and set the *is_label* flag. To complete the bookkeeping, it decrements the *scan_count* of the numeric macro. As seen before, the declaration of a numeric macro with *is_label* ≡ 1 will be removed.

In the *goto_count* of the label, the parser maintains a count of the **goto**'s that reference it. If transformations remove all **goto**'s, it is also possible to remove the target label. The whole bookkeeping is achieved by calling the function *clabel* at appropriate places in the parser.

⟨ external declarations 4 ⟩ +≡ (157)

```
extern void clabel(token *t, int reference);
```

⟨ functions 14 ⟩ +≡ (158)

```
void clabel(token *t, int reference)
{ if (t→tag ≡ NMACRO) { t→tag = CLABEL; SYM(t)→is_label = 1; }
  if (t→tag ≡ CLABEL) { SYM(t)→goto_count += reference;
    SYM(t)→scan_count −−;
  }
  else if (t→tag ≡ PRETURN) SYM(t)→goto_count += reference;
  else {
```

```
    if (t→tag ≡ PINTEGER) t→tag = CLABELN;
    return;
  }
  DBG(dbgreturn, "Using␣label␣%s␣(%d)␣in␣line␣%d\n", SYM(t)→name,
      SYM(t)→goto_count, t→line_no);
}
```

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 4 ⟩ +≡                                    (159)
  extern int exit_no;
#define TOK_RETURN add_token (PRETURN)→sym_no = exit_no
```

```
⟨ global variables 12 ⟩ +≡                                        (160)
  int exit_no;
```

```
⟨ initialize token list 23 ⟩ +≡                                   (161)
  exit_no = sym_no("exit");
```

To reflect the local nature of *exit* labels, I replace the symbol number by the symbol pointer before feeding a PRETURN token to the parser.

```
⟨ special treatment for WEB tokens 78 ⟩ +≡                        (162)
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 *goto_count* 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. In the rare case where the label is indeed a plain integer, the tag is CLABELN and I add the prefix "label" to the numeric value to make it a C identifier. The remaining cases are TEX's use of the labels *final_end* and **return**.

```
⟨ convert t from WEB to cweb 88 ⟩ +≡                              (163)
case CLABEL:
  if (t→sym_ptr→goto_count ≤ 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;
```

6.12 Constant declarations

The PCONST symbol occurs in TEX only once: before ⟨ Constants in the outer block ⟩. We use it to make the whole section an enumeration type.

```
⟨ convert t from WEB to cweb 88 ⟩ +≡                                (164)
case PCONST: wprint("enum␣{"); t = wtoken(t→next); wprint("};"); break;
```

In TEX there are only declarations of integer constants the only exception being the definition of the *pool_name* string. The parser changes the *pool_name* token into a CSTRDEF token; all the other identifiers are changed into CINTDEF tokens. I use the opportunity to replace the definition of *pool_name* by the definition of *empty_string* (see section 6.8).

```
⟨ convert t from WEB to cweb 88 ⟩ +≡                                (165)
case CSTRDEF: t = t→link→next;
  while (t→tag ≡ NL) { t = t→next;
    if (t→tag ≡ INDENT) t = t→next;
    if (t→tag ≡ PLEFT) t = t→link→next;
    if (t→tag ≡ ATCTL) t = t→next;
  }
  wputs("@!empty_string=256␣"
  "/*the␣empty␣string␣follows␣after␣256␣characters*/\n\n"); break;
```

To add the empty string to the string pool, the `ctex.patch` file replaces the section ⟨ Read the other strings from the `TEX.POOL` file ... ⟩ by a simple *make_string*( ).

6.13 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 identifiers 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. If the type starts with the PARRAY token (see next section) the variables will be printed as part of the type. If not, I continue after printing the type with whatever is left from this list. Note the special precautions taken to map subrange types to the different C integer types. I deal with this problem in section 6.14 and 6.17.

```
⟨ global variables 12 ⟩ +≡                                          (166)
  static token *varlist = NULL;
```

Using this information I can convert the variable declaration.

⟨ convert t from WEB to cweb 88 ⟩ +≡ (167)

```
case PDEFVARID:
  DBG(dbgcweb, "Converting␣variable␣list␣in␣line␣%d\n", t→line_no);
  varlist = t;
  { ⟨ determine the type and storage class 168 ⟩
    if (¬is_extern ∧ is_global) wprint("static");
    ⟨ print the variable's type 169 ⟩
    DBG(dbgcweb, "Finished␣variable␣type␣in␣line␣%d\n", t→line_no);
    while (varlist→tag ≡ PDEFVARID ∨ varlist→tag ≡ PID) { wid(varlist);
      varlist = wprint_to(varlist→next, varlist→link);
    }
    t = t→link;
  }
  DBG(dbgcweb, "Finishing␣variable␣list␣in␣line␣%d\n", t→line_no);
  break;
```

The following loop checks the list of variables for the requested type and storage class.

⟨ determine the type and storage class 168 ⟩ ≡ (168)

```
symbol *s;
int is_extern = 0, string_type = 0, is_int = 0, is_global = 0;
do { s = t→sym_ptr; is_global |= s→is_global; is_extern |= s→is_extern;
  is_int |= s→is_int;
  if (s→is_string > 0) {
    DBG(dbgstring, "Variable␣%s␣string␣(%d)␣defined␣in␣line␣%d\n",
        s→name, s→is_string, t→line_no); string_type = 1;
  }
  t = t→link;
} while (t→tag ≡ PID);
```

Used in 167 and 230.

After the previous loop has traversed the list, t points to the colon preceding the type. If the type is not a string or an integer, it is printed based on t.

⟨ print the variable's type 169 ⟩ ≡ (169)

```
if (string_type) wprint("char␣*");
else if (is_int) wprint("int");
else wprint_to(t, t→link);
```

Used in 167 and 230.

6.14 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 88 ⟩ +≡ (170)

```
case PDEFTYPEID:
  { token *type_name = t;
    token *type = type_name→link;
    DBG(dbgcweb, "Defining␣type␣%s␣in␣line␣%d\n", token2string(t),
        t→line_no); 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`. Since mixing signed and unsigned types in expression can create problems in C, I prefer the signed types over the unsigned type if possible.

⟨ convert t from WEB to cweb 88 ⟩ +≡ (171)

```
case CTSUBRANGE:
  { long int lo = t→up→previous→value;
    long int hi = t→up→next→value;
    DBG(dbgcweb, "Defining␣subrange␣type␣%ld..%ld␣in␣line␣%d\n", lo, hi,
        t→line_no);
    if (INT8_MIN ≤ lo ∧ hi ≤ INT8_MAX) wprint("int8_t");
    else if (0 ≤ lo ∧ hi ≤ UINT8_MAX) wprint("uint8_t");
    else if (INT16_MIN ≤ lo ∧ hi ≤ INT16_MAX) wprint("int16_t");
    else if (0 ≤ lo ∧ hi ≤ UINT16_MAX) wprint("uint16_t");
    else if (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␣%ld..%ld␣in␣line␣%d\n",
          lo, hi, t→line_no);
    t = t→link; break;
  }
case CTINT: wprint("int"); t = t→link; break;
```

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

⟨ external declarations 4 ⟩ +≡ (172)

```
#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 88 ⟩ +≡ (173)

```
case PRECORD:
  { DBG(dbgcweb, "Converting␣record␣type␣in␣line␣%d\n", t→line_no);
    wprint("struct␣{␣"); t = wprint_to(t→next, t→link);
```

```
    DBG(dbgcweb, "Finished␣record␣type␣in␣line␣%d\n", t→line_no);
    wput('}');
    if (t→tag ≠ NL) wput('␣');
    break;
  }
case CUNION:
  { DBG(dbgcweb, "Converting␣union␣type␣in␣line␣%d\n", t→line_no);
    wprint("union␣{␣"); t = wprint_to(t→next, t→link); wprint("};");
    DBG(dbgcweb, "Finished␣union␣type␣in␣line␣%d\n", t→line_no); 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 88 ⟩ +≡ (174)

```
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;
    long int lo, hi;
    int 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[%ld..%ld]␣type␣in␣line␣%d\n",
        lo, hi, t→line_no);
    t = wprint_to(element_type, element_type→link);
    while (true) { CHECK(varlist ≠ NULL,
          "Nonempty␣variable␣list␣expected␣in␣line␣%d",
```

```
        varlist→line_no);
      DBG(dbgarray, "Generating␣array␣variable␣%s␣in␣line␣%d\n",
        varlist→sym_ptr→name, varlist→line_no);
      wid(varlist); varlist→sym_ptr→is_zero_based = zero_based;
      if (¬zero_based) wput('0'); /* add a zero to the array name */
      wput('['); ⟨ generate array size 175 ⟩ wput(']');
      if (¬zero_based) /* now I need the array with the appropriate offset */
      { DBG(dbgarray,
          "Generating␣array␣pointer␣%s[%s=%ld..␣]␣in␣line␣%d\n",
          varlist→sym_ptr→name, token2string(from→next), lo,
          varlist→line_no); wputs(",␣*const␣"); wid(varlist);
        wputs("␣=␣"); wid(varlist), wput('0'); ⟨ generate array offset 176 ⟩;
      }
      varlist = varlist→link;
      if (varlist→tag ≡ PDEFVARID ∨ varlist→tag ≡ PID) wput(',');
      else break;
    }
    DBG(dbgarray, "Finished␣array␣type␣in␣line␣%d\n", t→line_no);
  }
  break;
```

⟨ generate array size 175 ⟩ ≡ (175)

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

Used in 174.

⟨ generate array offset 176 ⟩ ≡ (176)

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

Used in 174.

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.

```
⟨ auxiliary functions 68 ⟩ +≡                                    (177)
  static long int generate_constant(token *t, char sign, long 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→previous ≠ 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→line_no);
  }
```

6.15 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 88 ⟩ +≡                             (178)
case PFILE:
  { DBG(dbgcweb, "Converting␣file␣type␣in␣line␣%d\n", t→line_no);
    wprint("struct␣{@+FILE␣*f;@+"); t = wprint_to(t→next, t→link);
    wprint("@,d;@+}␣");
    DBG(dbgcweb, "Finished␣file␣type␣in␣line␣%d\n", t→line_no); break;
  }
```

As I will show in section 6.19, 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` 88 ⟩ +≡ (179)
case PUP: *wputs*("`.d`"); $t = t\rightarrow next$; **break**;

6.16 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` 88 ⟩ +≡ (180)
case PWHILE: *wprint*("`while␣`");
if ($t\rightarrow link \neq$ `NULL`) { *wput*('`(`'); $t = wprint\_to(t\rightarrow next, t\rightarrow link)$; *wputs*("`)␣`"); }
$t = t\rightarrow next$; **break**;

Other structured statements need more work.

Let's start with the Pascal **case** statement. Adding parentheses around the controlling expression follows the same schema that was just used for the **while** statement: the PCASE token is linked to the corresponding POF token while parsing. The body of the switch statement is always a compound statement so an opening brace is added.

⟨ convert t from `WEB` to `cweb` 88 ⟩ +≡ (181)
case PCASE:
if ($t\rightarrow link \equiv$ `NULL`) { *wprint*($t\rightarrow text$); $t = t\rightarrow next$; }
else
{ *wprint*("`switch␣(`"); $t = wprint\_to(t\rightarrow next, t\rightarrow link)$; *wputs*("`)␣{`"); }
break;

The complications of the case statement start with the case labels. Pascal requires a comma separated list of labels followed by a colon, a statement, and a semicolon; C needs the keyword "**case**" preceding every single label, followed by a colon, and a statement list which usually ends with "**break**;".

When faced with this problem, I tried a new strategy: inserting new tokens while parsing. I insert a CCASE token before each Pascal case label and replace the PCOMMAs between labels by CCOLONs. While it worked quite well, I still wished, I could have solved the problem without modifying the token list.

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

⟨ external declarations 4 ⟩ +≡ (182)
extern token $*$*winsert_case*(**token** $*t$, **token** $*s$);

There are a few spots where inserting the CCASE token just before the first token of the label produces rather unsightly results. For example when TEX processes a ligature or kern command, TEX uses case labels of the form $qi(1), qi(5)$: ... where the macro $qi(A)$ is defined as $A + min\_quarterword$. The *pp_lex* function will expand the *qi* macro feeding $1 + min\_quarterword, 5 + min\_quarterword$: ... to the parser. Inserting the **case** just before 1 and 5, as it was done in `web2w` version 0.4, produces $qi(\textbf{case}\ 1)$: $qi(\textbf{case}\ 5)$: ... which, by the way, is correct C code. A situation like this occurs only if the label starts in the same code sequence as the comma and the colon is nested inside a macro as a macro parameter. To solve the problem, both tokens, the start of the label and the comma, respectively colon, is passed to the function *winsert_case*. A simple loop now moves the insertion point backward until the enclosing macro is found.

```
⟨ functions 14 ⟩ +≡                                                    (183)
  static token *winsert_after(token *t, int tag, char *text)
  { token *n;
    DBG(dbgstmt, "Inserting␣token␣%s␣after␣%s␣in␣line␣%d\n",
        tagname(tag), TAG(t), t→line_no); n = new_token(tag);
    n→next = t→next; n→next→previous = n; n→previous = t; t→next = n;
    n→sequence_no = t→sequence_no; n→line_no = t→line_no; n→text = text;
    return n;
  }
  token *winsert_case(token *t, token *s)
  { if (t ≡ NULL) return NULL;                    /* no need to insert anything */
    if (t→sequence_no ≡ s→sequence_no ∧ t→pp_sp < s→pp_sp) {
      DBG(dbgstmt, "Moving␣case␣in␣line␣%d␣(%d/%d!=%d)\n", t→line_no,
          t→sequence_no, t→pp_sp, s→pp_sp);
      while (t→previous→sequence_no ≡ s→sequence_no ∧ t→tag ≠ PMACRO)
        t = t→previous;
    }
    return winsert_after(t→previous, CCASE, "case␣");
  }
```

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

```
⟨ convert t from WEB to cweb 88 ⟩ +≡                                   (184)
case CBREAK:
  if (t→previous→tag ≠ PSEMICOLON ∧ t→previous→tag ≠
      CSEMICOLON ∧ t→previous→tag ≠ PEND) wputs("@;");
  if (¬dead_end(t→up, t→line_no)) wprint("@+break;");
  t = t→next; break;
```

The semicolon that might be necessary before the "**break**" is inserted using a general procedure described in section 6.18.

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

```
⟨ auxiliary functions 68 ⟩ +≡                                        (185)
  static int dead_end(token *t, int line_no)
  { DBG(dbgbreak, "Searching␣for␣dead␣end␣in␣line␣%d:\n", line_no);
    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 88 ⟩ +≡                                 (186)
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, an **else**, or a new section.

```
⟨ convert t from WEB to cweb 88 ⟩ +≡                                 (187)
case PREPEAT: wprint("@/do@+{"); t = t→next; break;
case PUNTIL:
  { int sequence_no, line_no;
    token *end;
    wputs("}@+␣while␣(!("); sequence_no = t→sequence_no;
    line_no = t→line_no; end = t→next;
    while (true) {
      if (end→tag ≡ PSEMICOLON ∨ end→tag ≡ CSEMICOLON ∨ end→tag ≡
          PELSE) break;
      else if (end→tag ≡ ATSPACE ∨ end→tag ≡ ATSTAR) {
        end = wback(end)→next; break;
      }
      end = end→next;
    }
```

```
    CHECK(sequence_no ≡ end→sequence_no,
        "until:␣end␣of␣expression␣not␣found␣in␣line␣%d", line_no);
    t = wprint_to(t→next, end); wputs("))"); break;
  }
```

6.17 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 88 ⟩ +≡                                    (188)
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→line_no);
    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 TEX's 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. Here, I call the function *for_loop_variable* with three parameters: *id*, the loop control variable; *line_no*, 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: $+1$ for an upward loop, -1 for a downward loop, and 0 if the loop's limit is a variable.

While `web2w` version 0.4 went a long way to decide whether the loop variables subrange type needs to be promoted to an **int**, version 1.0 now simply converts all such variables to an **int**.

```
⟨ external declarations 4 ⟩ +≡                                          (189)
  extern void for_loop_variable(token *id, int line_no, int to, int direction);
```

⟨ functions 14 ⟩ +≡ (190)

```
void for_loop_variable(token *id, int line_no, int to, int direction)
{ SYM(id)→is_int = 1; DBG(dbgstmt, "\tLoop␣control␣variable␣%s:␣"
      "limit␣%d,␣direction␣%d␣in␣line␣%d\n",
      SYM(id)→name, to, direction, line_no);
}
```

6.18 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 a semicolon in certain cases, for example preceding an "**else**" or a "`}`", 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` 88 ⟩ +≡ (191)

```
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 14 ⟩ +≡ (192)
  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;
  }
  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→line_no);
        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→line_no);
        winsert_after(t, ATSEMICOLON, "@;");
      }
    }
  }
```

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

The function *wend* is called by the parser after a procedure body is parsed. It passes a pointer p to the body's parse tree and a pointer t to the terminating PEND token.

```
⟨ functions 14 ⟩ +≡ (193)
  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→line_no); label→tag = CIGNORE;
    SYM(label)→goto_count = −1000; CHECK(label→next→tag ≡ PCOLON,
        "Expected␣colon␣after␣label␣in␣line␣%d\n", label→line_no);
    label→next→tag = CIGNORE; p→next→tag = CIGNORE;
  }
  else DBG(dbgreturn, "No␣trailing␣exit:␣found␣preceding␣line␣%d\n",
        t→line_no);
}
```

⟨ external declarations 4 ⟩ +≡ (194)

```
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 41 ⟩ +≡ (195)

```
case CSEMICOLON: return ";";
```

⟨ special treatment for WEB tokens 78 ⟩ +≡ (196)

```
case CSEMICOLON: t = t→next; continue;
```

6.19 Procedure definitions

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. After these preparations, this is sufficient to get started with the procedure heading:

⟨ convert t from WEB to cweb 88 ⟩ +≡ (197)

```
case PPROCEDURE:
  DBG(dbgcweb, "Converting␣procedure␣heading␣in␣line␣%d\n", t→line_no);
  if (SYM(t) ≡ NULL ∨ SYM(t)→is_extern ≡ 0) wprint("static");
  ⟨ print the procedure heading 198 ⟩
  break;
```

⟨ print the procedure heading 198 ⟩ ≡ (198)

```
{ wprint("void"); t = wprint_to(t→next, t→link);
  if (t→tag ≠ POPEN) wputs("(void)");
  else t = wprint_to(t, t→link→next);
}
```

Used in 197 and 230.

If the procedure heading features a parameter list, the parser has converted the parameter identifiers to either PDEFPARAMID or PDEFREFID tokens, 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. It is handled similar to a variable declaration. The type identifier, however, needs to be repeated for each parameter in the list.

```
⟨ convert t from WEB to cweb 88 ⟩ +≡                                    (199)
case PDEFPARAMID: case PDEFREFID:
  { token *varlist = t, *type = t→link;
    DBG(dbgcweb, "Converting␣parameter␣list␣in␣line␣%d\n", t→line_no);
    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→line_no);
    break;
  }
```

Inside the procedure body, one thing that needs special attention are variables passed by reference. 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 88 ⟩ +≡                                    (200)
case CREFID: wputs("(*"), t = wid(t), wput(')'); break;
```

In a procedure, the parser changes TEX's **return** macro to a CPROCRETURN token and now I expand the macro to a **goto** *end* if the label *end* still exists because it was not in a tail position. Otherwise I generate a C **return** statement.

```
⟨ convert t from WEB to cweb 88 ⟩ +≡                                    (201)
case CPROCRETURN:
  if (t→sym_ptr→goto_count > 0) wprint("goto␣end");
  else wprint("return");
  t = t→next; break;
```

6.20 Procedure calls

The most complex part of a procedure call 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 identifier's entry in the symbol table. A value of 1 means "empty 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. The parser uses these definitions:

⟨ external declarations 4 ⟩ +≡ (202)
```
  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 12 ⟩ +≡ (203)
```
  unsigned int param_mask, param_bit;
```

When a procedure identifier is used in a procedure call, the parser changes the token's tag to PCALLID. This triggers the execution of the following code. It prints the procedure name after checking for a possible change of the name caused by the elimination of the string pool, and adds `"()"` for parameterless procedures. In TEX the procedure identifier 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.

⟨ convert t from WEB to cweb 88 ⟩ +≡ (204)
```
case PCALLID: DBG(dbgcweb, "Converting␣call␣to␣%s␣in␣line␣%d\n",
      SYM(t)→name, t→line_no);
  pstring2n(t, t→up); wid(t);
  if (SYM(t)→value ≡ 1) wputs("()");
  t = t→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 88 ⟩ +≡ (205)
```
case POPEN: wput('(');
  if (t→up ≡ NULL ∨ SYM(t→up)→value ≡ 0) t = t→next;
  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 if (t→tag ≡ POPEN) t = wprint_to(t, t→link); /* skip nested calls */
      else t = wtoken(t);
    }
  } break;
```

6.21 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 88 ⟩ +≡                                    (206)
case PFUNCTION:
  DBG(dbgcweb, "Converting␣function␣heading␣in␣line␣%d\n", t→line_no);
  if (SYM(t) ≡ NULL ∨ SYM(t)→is_extern ≡ 0) wprint("static");
  ⟨ print the function heading 207 ⟩
  break;
```

```
⟨ print the function heading 207 ⟩ ≡                                    (207)
  { token *param = t→link;
    token *type;
    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;
  }
```

Used in 206 and 230.

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 a tail position of the function. While parsing, I build a parse tree of the function body. 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 14 ⟩ +≡                                                     (208)
  static bool wtail(token *t)
  { CHECK(t ≠ NULL, "Unexpected␣NULL␣token␣while␣searching␣f\
        or␣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 4 ⟩ +≡ (209)

```
  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 *goto_count*. If at the end it is zero, I can omit the label *end* marking the end of the function body.

⟨ functions 14 ⟩ +≡ (210)

```
  void wreturn(token *t, int tail, token *link)
  { CHECK(t ≠ NULL,
    "Unexpected␣NULL␣token␣while␣searching␣for␣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→line_no); t→tag = CRETURN; IGN(t→next);
        if (link ≠ NULL) { link→sym_ptr→goto_count −−;
          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→goto_count, t→line_no);
      }
    }
    return;
  case CRETURN: /* this happened when the return; is inside a macro */
    if (t→sym_ptr ≠ NULL) { t→sym_ptr→goto_count−−;
      DBG(dbgreturn, "Eliminating␣label␣%s␣(%d)␣in␣line␣%d\n",
          t→sym_ptr→name, t→sym_ptr→goto_count, t→line_no);
    }
    return;
  default:
    ERROR("Unexpected␣tag␣%s␣in␣line␣%d""␣while␣searching␣fo\
        r␣return␣statements", TAG(t), t→line_no);
  }
}
```

The CRETURN tokens created by *wreturn* are finally translated into a C return statement.

```
⟨ convert t from WEB to cweb 88 ⟩ +≡                                    (211)
case CRETURN:
  DBG(dbgcweb, "Converted␣function␣return␣%s␣in␣line␣%d\n",
      SYM(t)→name, t→line_no); wprint("return"); t = t→next; break;
```

After these transformations, 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.

```
⟨ global variables 12 ⟩ +≡                                              (212)
  static int x_over_n_no, xn_over_d_no;

⟨ initialize token list 23 ⟩ +≡                                         (213)
  x_over_n_no = sym_no("x_over_n"); xn_over_d_no = sym_no("xn_over_d");
```

The parser calls *wlocal_value* to 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.

```
⟨ external declarations 4 ⟩ +≡                                          (214)
  extern int wlocal_value(token *t, token *begin, token *end);

⟨ functions 14 ⟩ +≡                                                     (215)
  int wlocal_value(token *t, token *begin, token *end)
  { int f_no = t→sym_no;
    if (f_no ≠ x_over_n_no ∧ f_no ≠ xn_over_d_no) return 0;
    DBG(dbgcweb, "Discovered␣function␣%s;␣in␣line␣%d\n",
        SYM(t)→name, t→line_no);
```

```
    CHGTAG(begin, PFBEGIN); begin→sym_no = f_no;
    CHGTAG(end, PFEND); end→sym_no = f_no;
    return 1;
  }
```

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 88 ⟩ +≡ (216)

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

6.22 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 88 ⟩ +≡ (217)

```
case CMAIN: if (SYM_PTR("main")→is_extern) SYM_PTR("main")→type = t;
  else wprint("static");
  wprint("int␣main(void)␣{"); t = t→next; break;
case CMAINEND: wprint("return␣0;␣}"); t = t→next; break;
```

7 Running `web2w`

7.1 The command line

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

```
⟨ functions 14 ⟩ +≡                                                    (218)
  void usage(void)
  { fprintf(stderr, "Usage: web2w [parameters] filename.web\n"
    "Parameters:\n"
    "\t -p\t\t\t generate a Pascal output file\n"
    "\t -o file\t\t specify an output file name\n"
    "\t -l\t\t\t redirect stderr to a log file\n"
    "\t -y\t\t\t generate a trace while parsing Pascal\n"
    "\t -u\t\t\t generate macro names using upper case letters\n"
    "\t -h\t\t\t generate a header file section\n"
    "\t -e file\t\t file with extern identifiers\n"
    ⟨ show debugging options 235 ⟩
    ); exit(1);
  }
```

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

```
⟨ global variables 12 ⟩ +≡                                             (219)
#define MAX_NAME 256
  static char basename[MAX_NAME];
```

```
⟨ process the command line 220 ⟩ ≡                                     (220)
  { int mk_logfile = 0, mk_pascal = 0, baselength = 0;
    char *w_file_name = NULL;
    ww__flex_debug = 0; pp_debug = 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++;
        if (*argv ≡ NULL ∨ *argv[0] ≡ '-') usage();
        w_file_name = *argv; break;
      case 'l': mk_logfile = 1; break;
      case 'y': pp_debug = 1; break;
      case 'u': uppercase = 1; break;
      case 'h': header_section = 1; break;
      case 'e': argv++;
        if (*argv ≡ NULL ∨ *argv[0] ≡ '-') usage();
        exports_name = *argv; break;
      case 'd':
        argv++;
        if (*argv ≡ NULL ∨ *argv[0] ≡ '-') usage();
        debugflags = strtol(*argv, NULL, 16);
        if (debugflags & dbgflex) ww__flex_debug = 1;
        if (debugflags & dbgbison) pp_debug = 1;
        break;
      }
    }
    else { strncpy(basename, *argv, MAX_NAME − 1);
      baselength = strlen(basename);
      if (baselength > 4 ∧ strncmp(basename + baselength − 4, ".web", 4) ≡ 0)
        baselength = baselength − 4;
      basename[baselength] = 0;
      if (*(argv + 1) ≠ NULL) usage();
    }
    argv++;
  }
  ⟨ open the files 225 ⟩
}
```

Used in 2.

The `-u` option causes `web2w` to convert all macro names to upper case letters.

⟨ global variables 12 ⟩ +≡ (221)

```
int uppercase = 0;
```

⟨ convert macro names name to upper case if requested 222 ⟩ ≡ (222)

```
if (uppercase) { char *s = SYM(t)→name;
  while (*s ≠ 0) {
    if ('a' ≤ *s ∧ *s ≤ 'z') *s = *s − 'a' + 'A';
    s++;
  }
}
```

Used in 144.

7.2 Opening files

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 *pp_out*, if the output of the Pascal code is requested. For technical reasons, the scanner generated by `flex` needs an output file *ww_out*. The log file is opened first because this is the place where error messages should go while the other files are opened.

```
⟨ global variables 12 ⟩ +≡                                              (223)
  static FILE *w = NULL;
  static FILE *pp_out = NULL;
  FILE *logfile = NULL;
  static char *exports_name = NULL;

⟨ external declarations 4 ⟩ +≡                                          (224)
  extern FILE *logfile;

⟨ open the files 225 ⟩ ≡                                                (225)
  if (mk_logfile) { strcat(basename, ".log");
    logfile = freopen(basename, "w", stderr);
    if (logfile ≡ NULL) {
      fprintf(stderr, "Unable to open logfile %s", basename);
      logfile = stderr;
    }
    ww_out = logfile; basename[baselength] = 0;
  }
  else { logfile = stderr; ww_out = stderr;
  }
  strcat(basename, ".web"); ww_in = fopen(basename, "r");
  if (ww_in ≡ NULL) ERROR("Unable to open input file %s", basename);
  basename[baselength] = 0;
  if (w_file_name ≡ NULL) { strcat(basename, ".w"); w_file_name = basename;
  }
  w = fopen(w_file_name, "w");
  if (w ≡ NULL) ERROR("Unable to open output file %s", w_file_name);
  basename[baselength] = 0;
  if (mk_pascal) { strcat(basename, ".pas"); pp_out = fopen(basename, "w");
    if (pp_out ≡ NULL) ERROR("Unable to open pp_out file %s", basename);
    basename[baselength] = 0;
  }                                                                     Used in 220.
```

7.3 Generating a header file

The `-h` option causes `web2w` to generate a final section that produces a header file. The header file contains all macros defined in the `WEB` file and all the **extern** procedures, functions, and global variables.

```
⟨ global variables 12 ⟩ +≡                                              (226)
  int header_section = 0;
```

```
⟨ generate a header section if requested 227 ⟩ ≡                    (227)
  if (header_section) { wputs("\n@␣Generating␣the␣header␣file.\n");
    wputs("@("); wputs(basename); wputs(".h@>=\n"
    "@h\n"
    "enum␣{@+@<Constants␣in␣the␣outer␣block@>@+};\n"
    "@<Types␣in␣the␣outer␣block@>@;\n");
    ⟨ generate extern declarations 230 ⟩
  }                                                  Used in 114.
```

The `-e` option can be used to specify a file with a list of identifiers that should be exported. It is followed by the name of a text file that contains one identifier per line. If a global variable, function, or procedure identifier matches one of these lines, it is marked as **extern** and a definition is included in the header file. All other global identifiers are defined as **static**.

When we read the text file, we ignore `'\r'`, but otherwise only alphanumeric characters, underscores, and newlines are allowed. The *is_extern* field in the symbol table is used to link extern symbols into a list.

```
⟨ global variables 12 ⟩ +≡                                          (228)
  int extern_symbols = −1;

⟨ finalize token list 70 ⟩ +≡                                       (229)
  if (exports_name ≠ NULL) { int c, n = 0;
    char s[MAX_NAME];
    int k;                                          /* index into s */
    FILE *e = fopen(exports_name, "r");
    if (e ≡ NULL)
      ERROR("Unable␣to␣open␣extern␣name␣file␣%s", exports_name);
    SYM_PTR("main")→is_extern = 0; c = getc(e);
    while (c ≠ EOF) { n++; k = 0;
      while (c ≠ EOF) {
        if (c ≡ '\n') { c = getc(e); break; }
        else if (c ≡ '\r') c = getc(e);
        else if (¬isalnum(c) ∧ c ≠ '_')
          ERROR("Unexpected␣character␣%c(%x)␣in␣%s,␣line␣%d\n",
              c, c, exports_name, n);
        else { s[k++] = c;
          if (k ≥ MAX_NAME)
            ERROR("extern␣name␣too␣long␣in␣%s,␣line␣%d\n",
                exports_name, n);
          c = getc(e);
        }
      }
      if (k > 0) { int i;
        s[k] = 0; i = sym_no(s); symbol_table[i]→is_extern = extern_symbols;
        extern_symbols = i;
        DBG(dbgid, "Extern␣symbol␣%s\n", symbol_table[i]→name);
```

```
    }
  }
  fclose(e);
}
```

Next we generate the **extern** declarations. The parser has set the *type* field of an extern symbol to point to the PPROCEDURE token of the procedure's definition, and the *sym_no* of the PPROCEDURE token to the symbol number of the procedure's name. The former is used now to generate the declaration, the latter is used above to decide whether to add the **static** keyword. The same linkage is also done for PFUNCTION tokens. Simpler is the linkage for global variables, where the PID or PDEFVARID is linked to itself. However the generation of extern definitions for variables is more complex due to type conversions and the special handling of Pascal arrays. For arrays, only a constant pointer is exported, pointing to the actual or virtual element with index zero.

⟨ generate **extern** declarations 230 ⟩ ≡ (230)

```
while (extern_symbols > 0) { symbol *x = symbol_table[extern_symbols];
  token *t = x→type;
  if (t ≡ NULL) ERROR("Symbol␣%s␣in␣file␣%s␣was␣never␣defined",
      x→name, exports_name);
  else { wprint("extern");
    if (t→tag ≡ CMAIN) wprint("int␣main(void)");
    if (t→tag ≡ PPROCEDURE) ⟨ print the procedure heading 198 ⟩
    else if (t→tag ≡ PFUNCTION) ⟨ print the function heading 207 ⟩
    else if (t→tag ≡ PDEFVARID ∨ t→tag ≡ PID) { varlist = t;
      token *array_type = NULL;
      ⟨ determine the type and storage class 168 ⟩
      if (¬is_extern)
        ERROR("Static␣variables␣become␣non␣static␣in␣line␣%d\n",
            t→line_no);
      if (t→next→tag ≡ PARRAY) array_type = t→next;
      else if (t→next→tag ≡ PPACKED ∧ t→next→next→tag ≡ PARRAY)
        array_type = t→next→next;
      if (array_type ≠ NULL) {
        token *element_type = array_type→link→link→link→link;
        wprint_to(element_type, element_type→link);
        if (x→is_zero_based) { wprint(x→name); wputs("[]"); }
        else { wputs("␣*const␣"); wprint(x→name); }
      }
      else { ⟨ print the variable's type 169 ⟩ wprint(x→name); }
    }
    wputs(";\n");
  }
  extern_symbols = x→is_extern;
}
```

Used in 227.

As an example, suppose you want to use just a few of TEX's functions in one of your projects, and for the sake of simplicity, assume that you just need the simple *x_over_n* function. Then you could create a file, let's call it `myexterns.txt` containing only a single line with the name of the function you need: `x_over_n`. Now you run "`web2w -h -e myexterns.txt tex.web`" to get `tex.w` and patch it with `ctex.patch` to get `ctex.w` (see also section 10.4). When you now run `ctangle` on `ctex.w` you will get `ctex.c` as usual, and in addition, you get a header file `tex.h` that contains TEX's macros, constants, types and at the bottom a single function declaration: "`extern scaled x_over_n(scaled x,int n);`". In the `ctex.c` file, all global variables and functions, even the *main* function, are declared **static**—except of course the function *x_over_n*. When you compile `ctex.c`, ask your compiler to optimize your code, for example by adding the option `-O3`, and it will remove all the static functions and variables that are defined but never used. In our example, the compiler will produce an object file containing in the text segment just the function *x_over_n* (88 byte) and in the bss section two global variables **bool** *arith_error* and **scaled** *rem* (5 byte). The two variables are in the object file because they are referenced by *x_over_n* but they are not visible. To make them visible, simply add them to your `myexterns.txt` file.

7.4 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 4 ⟩ +≡ (231)

```
#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 4 ⟩ +≡ (232)

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

The amount of debugging depends on the debugging flags.

⟨ global variables 12 ⟩ +≡ (233)

```
debugmode debugflags = dbgnone;
```

The different debug values are taken from an enumeration type.

⟨ external declarations 4 ⟩ +≡ (234)

```
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,
    dbgstmt = #1000, dbgslash = #2000, dbgmacro = #4000,
```

```
        dbgarray = #8000, dbgreturn = #10000, dbgsemicolon = #20000,
        dbgbreak = #40000, dbgtypes = #80000
  } debugmode;
  extern debugmode debugflags;
  extern int ww_flex_debug;
```

Here is an explanation of these flags as shown in the help text.

⟨ show debugging options 235 ⟩ ≡ (235)

```
  "\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␣Pascal␣statement␣debugging\n"
  "\t␣\t␣␣␣␣\t␣XX=2000␣real␣division␣debugging\n"
  "\t␣\t␣␣␣␣\t␣XX=4000␣macro␣debugging\n"
  "\t␣\t␣␣␣␣\t␣XX=8000␣array␣debugging\n"
  "\t␣\t␣␣␣␣\t␣XX=10000␣return␣debugging\n"
  "\t␣\t␣␣␣␣\t␣XX=20000␣semicolon␣debugging\n"
  "\t␣\t␣␣␣␣\t␣XX=40000␣break␣debugging\n"
  "\t␣\t␣␣␣␣\t␣XX=80000␣type␣debugging\n"
```

Used in 218.

The following macros are used to produce debug output.

⟨ external declarations 4 ⟩ +≡ (236)

```
#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␣|␣%ld\n", TAG(t),
      TAG(t→previous), TAG(t→next), t→value)
```

8 The scanner

```
%{
#include "web2w.h"
#include "pascal.tab.h"
static int  pre_ctl_mode=0;
%}

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

%x PASCAL MIDDLE DEFINITION FORMAT NAME CONTROL

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);SEQ;BEGIN(DEFINITION);
@[fF]                   EOS;TOK("@f ",ATF);SEQ;BEGIN(FORMAT);
@[pP]                   EOS;TOK("@p ",ATP);PROGRAM;PUSH;SEQ;BEGIN(PASCAL);
@\<{SP}                 EOS;TOK("@<",ATLESS);PUSH;BOS;SEQ;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;
@\^                     ADD;CTL;
@\.                     ADD;CTL;
@\:                     ADD;CTL;
@!                      EOS;TOK("@!",ATEX); BOS;
@\?                     EOS;TOK("@?",ATQM); BOS;
```

```
@@                      EOS;TOK("@@",ATAT);BOS;

\n                      ADD;
\\\?                    add_string("\\@@"); /* |\?| is used in cwebmac.tex */
([^\\%@|{}.\n])*        ADD; /* we do not analyze TEX parts any further */
\\[\\%@|{}]             ADD;
\\                      ADD;
\%.*                    ADD;
.                       ADD;

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

<CONTROL>{
@\>                     ADD; END_CTL;
\\AT!                   add_string("\\AT"); /* webmac uses \AT! */
\\_                     add_string("\\_");
_                       add_string("\\_"); /* etex uses '_' without '\' */
\n                      ERROR("Unexpected \\n in control text");
@                       ERROR("Unexpected @ in control text");
.                       ADD;
}

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

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

\n                      TOK("\n",NL); /* non Pascal tokens */
@\^                     BOS; ADD; CTL;
@\.                     BOS; ADD; CTL;
@\:                     BOS; ADD; CTL;
@\$                     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);

=                   TOK("=",PEQ); /* Pascal tokens */
\+                  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);

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

"mod"               TOK("mod",PMOD); /* pascal keywords */
"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; SYM(last_token)->scan_count++;
\"([^"\n]|\"\")\"       TOK(COPY,CHAR); /* single character string */
\"([^"\n]|\"\")*\"      TOK(COPY,STRING); /* multiple character string */
\'([^'\n]|\'\'|@@)\'    TOK(COPY,PCHAR);
\'([^'\n]|\'\')*\'      TOK(COPY,PSTRING);
[0-9]+                  TOK(COPY,PINTEGER);
{REAL}                  TOK(COPY,PREAL);

^[[:blank:]]+           TOK(COPY,INDENT);
[[:blank:]]             ; /* 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);}

%%
```

9 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; nonterminal symbols use a slanted font.

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

static int function=0;

%}

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

extern int pp_parse(void);
extern int pp_debug;
}

%token-table
%defines
%error_verbose
%debug
%name-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 ATCTL
%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"
%token CTLOCAL "local :  id"
%token CTINT "int type"

%%
program : programheading globals
          PBEGIN statements PEND PDOT
          { CHGTAG($3,CMAIN); CHGTAG($5,CMAINEND); IGN($6);
            wsemicolon($4,$5);
          }
        ;

programheading : PPROGRAM PID PSEMICOLON { LNK($1,$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)->scan_count--;}
          | 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($2,$4);
                    SETVAL($1,getval($3)); CHGID($1,PCONSTID);
                    CHGTAG($1,CINTDEF); CHGTAG($2,PASSIGN); CHGTAG($4,PCOMMA); }
                ;

conststringdefinition : PID PEQ PSTRING PSEMICOLON
                      { seq($1,$4); LNK($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); }
         ;

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)); USE_NMACRO($1); }
              | PCONSTID { $$=join(PCONSTID,$1,NULL,getval($1)); }
              ;

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

packed : PPACKED
       |
       ;

builtin_type : PTYPEINT
             | PTYPEREAL
             | PTYPEBOOL
             | PTYPECHAR
             ;

typename : builtin_type { $$=NULL; }
         | PID { $$=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,CTINT); UP($2,$3); LNK($2,$4); }
               ;

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

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

```
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)->scan_count--; localize($1); }
               | PINTEGER { IGN($1); }
               | labeldecl PPLUS PINTEGER { IGN($2); IGN($3); }
               ;

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

localvardeclarations : localvardeclaration
                     | localvardeclarations localvardeclaration
                     ;

localvardeclaration : localvarids PCOLON PID PSEMICOLON
                      { LNK($1,$2); CHGTAG($2,CTLOCAL); LNK($2,$4); }
                    | localvarids PCOLON builtin_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,CTINT);
                        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
           { function=0;
             if (!wlocal_value($1,$3,$6))
             { 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); CHGSNO($1,$2); CHGTYPE($2,$1); CHGID($2,PPROCID);
             CHGVALUE($2,1); IGN($3); }
         | PPROCEDURE pid POPEN formals PCLOSE PSEMICOLON
           { LNK($1,$3); CHGSNO($1,$2); CHGTYPE($2,$1); CHGID($2,PPROCID);
             CHGVALUE($2,param_mask); LNK($4,$5); IGN($6); }
         ;

fheading : PFUNCTION pid PCOLON typename PSEMICOLON
           { $$=$2; LNK($1,$3); CHGSNO($1,$2); CHGTYPE($2,$1);
             CHGID($2,PFUNCID); CHGVALUE($2,1);IGN($3); LNK($3,$5); IGN($5); }
         | PFUNCTION pid POPEN formals PCLOSE
           PCOLON typename PSEMICOLON { $$=$2; LNK($1,$3); CHGSNO($1,$2);
             CHGTYPE($2,$1); 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 : localentire_var { NEXT_PARAM; CHGTAG($1,PDEFPARAMID); $$=$1; }
      | PVAR localentire_var { REF_PARAM; NEXT_PARAM; IGN($1);
          CHGTAG($2,PDEFREFID);CHGID($2,CREFID); $$=$2; }
      ;

proc_stmt : PPROCID POPEN args PCLOSE { CHGTAG($1,PCALLID); $$=$1;
              UP($2,$1); pchar2string($1,$3); }
          | PCALLID POPEN args PCLOSE
            { $$=$1; UP($2,$1); pchar2string($1,$3); }
          | PPROCID { CHGTAG($1,PCALLID); $$=$1; }
          | PCALLID { $$=$1; }
          ;

function_call : PFUNCID POPEN args PCLOSE
                { CHGTAG($1,PCALLID); $$=$1; UP($2,$1); pchar2string($1,$3); }
              | PCALLID POPEN args PCLOSE
                { $$=$4; UP($2,$1); pchar2string($1,$3); }
              | 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 { $$=$2; }
          | PRETURN { if (function) clabel($1,1);
              else { CHGTAG($1,CPROCRETURN);$1->sym_ptr->goto_count++; }
              $$=$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 { $$=$2; pvar_string($1,$3);}
            | variable PASSIGN STRING { $$=$2; pvar_string($1,$3); }
            | variable PASSIGN POPEN STRING PCLOSE
              { $$=$2; pvar_string($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 statement { $$=$2; }
             | POTHERS statement { $$=$2; }
             ;

case_labels : case_label PCOLON { winsert_case($1, $2); }
            | case_label PCOMMA { winsert_case($1,$2);
                CHGTAG($2,CCOLON); CHGTEXT($2,": "); } case_labels
            | case_label CCOLON { winsert_case($1,$2); } case_labels
            ;
```

```
case_label : PINTEGER { $$=$1; }
           | NMACRO { USE_NMACRO($1); $$=$1; }
           | PINTEGER PPLUS NMACRO { USE_NMACRO($3); $$=$1; }
           | NMACRO PPLUS NMACRO { USE_NMACRO($1); USE_NMACRO($3); $$=$1; }
           | NMACRO PPLUS PINTEGER { USE_NMACRO($1); $$=$1; }
           | NMACRO PMINUS NMACRO PPLUS NMACRO
             { USE_NMACRO($1); USE_NMACRO($3); USE_NMACRO($5);$$=$1; /* eTeX */ }
           | CCASE NMACRO { USE_NMACRO($2); $$=NULL; }
           | CCASE PINTEGER { $$=NULL; }
           | CCASE NMACRO PPLUS NMACRO
             { USE_NMACRO($2); USE_NMACRO($4); $$=NULL; }
           ;

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
           { for_loop_variable($2,$1->line_no,0,0);
             DBG(dbgstmt,"for variable %s, limit variable in line %d\n",
             SYM($2)->name,$2->line_no);
             $$=join(PFOR,$8,NULL,0);LNK($1,$5);LNK($5,$7); }
         | PFOR PID PASSIGN expression PTO iconst PDO statement
           { for_loop_variable($2,$1->line_no,$6->value,1);
             DBG(dbgstmt,"for variable %s, limit up in line %d\n",
             SYM($2)->name,$2->line_no);
             $$=join(PFOR,$8,NULL,0);LNK($1,$5);LNK($5,$7); }
         | PFOR PID PASSIGN expression PDOWNTO iconst PDO statement
           { for_loop_variable($2,$1->line_no,$6->value,-1);
             DBG(dbgstmt,"for variable %s, limit down in line %d\n",
             SYM($2)->name,$2->line_no);
             $$=join(PFOR,$8,NULL,0);LNK($1,$5);LNK($5,$7); }
         ;

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

```
simple_variable : PID { USE($1); }
                | CREFID { USE($1); }
                ;

compound_variable : indexed_var
                  | field_var
                  | file_var
                  ;

variable : simple_variable
         | compound_variable
         ;

indexed_var : variable PSQOPEN expression PSQCLOSE
              { $$=join(PSQOPEN,$1,$3,0); }
            | variable PSQOPEN STRING PSQCLOSE
              { $$=join(PSQOPEN,$1,$3,0); }
            | PARRAYFILEID PSQOPEN expression PSQCLOSE
              { $$=join(PSQOPEN,$1,$3,0); }
            ;

field_var : variable PDOT PID { $$=join(PDOT,$1,$3,0); }
          ;

file_var : variable PUP { $$=join(PUP,$1,NULL,0); }
         ;

expression : simple_expr { $$=$1; }
           | simple_expr relop simple_expr
             { pvar_string($1,$3); $$=join($2->tag,$1,$3,0); }
           | simple_expr PEQ STRING
             { pvar_string($1,$3); $$=join(PEQ,$1,$3,0); }
           ;

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

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

```
sign : PPLUS
     | PMINUS
     ;

addop : PPLUS
      | PMINUS
      | POR
      ;

term : factor { $$=$1; }
     | term mulop factor { $$=$2; }
     ;

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

factor : variable { $$=$1; }
       | unsigned_const { $$=$1; }
       | POPEN expression PCLOSE { $$=$2; }
       | function_call
       | PNOT factor { $$=$1; }
       ;

unsigned_const : real
               | PINTEGER
               | NMACRO { USE_NMACRO($1); }
               | PSTRING
               | PCHAR
               | PCONSTID
               ;

real : PREAL
     | PINTEGER PDOT PINTEGER { $$=$2; /* used in line 2361 */}
     ;

%%

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

10 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.

10.1 Generating TEX

1. The `web2w` package can be downloaded from `www.ctan.org`. Alternatively you can download it from the HINT project web site at `hint.userweb.mwn.de` where you might also find more recent development versions. 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 might not be able to use the standard configuration 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 you 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`.

10.2 Running TEX

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

There are some differences between the virgin 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`

10.3 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.

10.4 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 you 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 you are ready to apply `web2w`. Run the command
    ```
    ./web2w -o tex.w tex/tex.web
    ```
    This command will produce `tex.w`, but you are not yet finished yet. you 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] 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.

[2] Donald E. Knuth. *The WEB system of structured documentation.* Stanford University, Computer Science Dept., Stanford, CA, 1983. STAN-CS-83-980. `https://ctan.org/pkg/cweb`.

[3] Donald E. Knuth. *TEX: The Program.* Computers & Typesetting, Volume B. Addison-Wesley, 1986.

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

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

[6] Donald E. Knuth and Silvio Levy. *The CWEB System of Structured Documentation.* Addison Wesley, 1994. `https://ctan.org/pkg/cweb`.

[7] Martin Ruckert. Converting TEX from `WEB` to cweb. *TUGboat*, 38(3):353–358, 2017.

[8] Martin Ruckert. *WEB to cweb.* KDP, 2017. ISBN 1-548-58234-4. `https://amazon.com/dp/1548582344`.

[9] Martin Ruckert. HINT: Reflowing TEX output. *TUGboat*, 39(3):217–223, 2018.

[10] Martin Ruckert. The design of the HINT file format. *TUGboat*, 40(2):143–146, 2019.

[11] Martin Ruckert. *HINT: The File Format.* KDP, 2019. ISBN 1-079-48159-1.

[12] TEX User Group, `https://tug.org/web2c`. *Web2c: A TEX implementation.*

Index

Symbols

A

B

C

D

E

F

G

H

N

O

P

Q

R

S