HINT:
The File Format

HINT: The File Format

Version 1.4

Reflowable
Output
for TEX

Für meine Mutter

MARTIN RUCKERT *Munich University of Applied Sciences*

Second edition

Ruckert, Martin.
HINT: The File Format
Includes index.
ISBN 979-854992684-4

Internet page http://hint.userweb.mwn.de/hint/format.html may contain current information about this book, downloadable software, and news.

ruckert@cs.hm.edu

ISBN-13: 979-854992684-4

First printing: August 2019
Second edition: August 2021

Last commit: Tue Feb 4 19:26:26 2025

Preface

Late in summer 2017, with my new C based `cweb` implementation of TEX[9] in hand[20][17][18], I started to write the first prototype of the HINT viewer. I basically made two copies of TEX: In the first copy, I replaced the *build_page* procedure by an output routine which used more or less the printing routines already available in TEX. This was the beginning of the HINT file format. In the second copy, I replaced TEX's main loop by an input routine that would feed the HINT file more or less directly to TEX's *build_page* procedure. And after replacing TEX's *ship_out* procedure by a modified rendering routine of a dvi viewer that I had written earlier for my experiments with TEX's Computer Modern fonts[16], I had my first running HINT viewer. My sabbatical during the following Fall term gave me time for "rapid prototyping" various features that I considered necessary for reflowable TEX output[19].

The textual output format derived from the original TEX debugging routines proved to be insufficient when I implemented a "page up" button because it did not support reading the page content "backwards". As a consequence, I developed a compact binary file format that could be parsed easily in both directions. The HINT short file format was born. I stopped an initial attempt at eliminating the old textual format because it was so much nicer when debugging. Instead, I converted the long textual format into the short binary format as a preliminary step in the viewer. This was not a long term solution. When opening a big file, as produced from a 1000 pages TEX file, the parsing took several seconds before the first page would appear on screen. This delay, observed on a fast desktop PC, is barley tolerable, and the delay one would expect on a low-cost, low-power, mobile device seemed prohibitive. The consequence is simple: The viewer will need an input file in the short format; and to support debugging (or editing), separate programs are needed to translate the short format into the long format and back again. But for the moment, I did not bother to implement any of this but continued with unrestricted experimentation.

With the beginning of the Spring term 2018, I stopped further experiments with the HINT viewer and decided that I had to write down a clean design of the HINT file format. Or of both file formats? Professors are supposed to do research, and hence I tried an experiment: Instead of writing down a traditional language specification, I decided to stick with the "literate programming" paradigm[10] and write the present book. It describes and implements the `stretch` and `shrink` programs translating one file format into the other. As a side effect, it contains the underlying language specification. Whether this experiment is a success as a language specification remains to be seen, and you should see for yourself. But the only important measure for the value of a scientific experiment is how much you can learn form it—and I learned a lot.

The whole project turned out to be much more difficult than I had expected. Early on, I decided that I would use a recursive descent parser for the short format and an LR(k) parser for the long format. Of course, I would use `lex/flex` and `yacc/bison` to generate the LR(k) parser, and so I had to extend the `cweb` tools[11] to support the corresponding source files.

About in mid May, after writing down about 100 pages, the first problems emerged that could not be resolved with my current approach. I had started to describe font definitions containing definitions of the interword glue and the default hyphen, and the declarative style of my exposition started to conflict with the sequential demands of writing an output file. So it was time for a first complete redesign. Two more passes over the whole book were necessary to find the concepts and the structure that would allow me to go forward and complete the book as you see it now.

While rewriting was on its way, many "nice ideas" were pruned from the book. For example, the initial idea of optimizing the HINT file while translating it was first reduced to just gathering statistics and then disappeared completely. The added code and complexity was just too distracting.

What you see before you is still a snapshot of the HINT file format because its development is still under way. We will know what features are needed for a reflowable TEX file format only after many people have started using the format. To use the format, the end-user will need implementations, and the implementer will need a language specification. The present book is the first step in an attempt to solve this "chicken or egg" dilemma.

München
August 20, 2019 *Martin Ruckert*

Contents

1 Introduction

This book defines a file format for reflowable text. Actually it describes two file formats: a long format that optimizes readability for human beings, and a short format that optimizes readability for machines and the use of storage space. Both formats use the concept of nodes and lists of nodes to describe the file content. Programs that process these nodes will likely want to convert the compressed binary representation of a node—the short format—or the lengthy textual representation of a node—the long format—into a convenient internal representation. So most of what follows is just a description of these nodes: their short format, their long format and sometimes their internal representation. Where as the description of the long and short external format is part of the file specification, the description of the internal representation is just informational. Different internal representations can be chosen based on the individual needs of the program.

While defining the format, I illustrate the processing of long and short format files by implementing two utilities: `shrink` and `stretch`. `shrink` converts the long format into the short format and `stretch` goes the other way.

There is also a prototype viewer for this file format and a special version of TEX[8] to produce output in this format. Both are not described here; a survey describing them can be found in [19].

1.1 Glyphs

Let's start with a simple and very common kind of node: a node describing a character. Because we describe a format that is used to display text, we are not so much interested in the character itself but we are interested in the specific glyph. In typography, a glyph is a unique mark to be placed on the page representing a character. For example the glyph representing the character 'a' can have many forms among them '*a*', '**a**', or '`a`'. Such glyphs come in collections, called fonts, representing every character of the alphabet in a consistent way.

The long format of a node describing the glyph 'a' might look like this:" `<glyph 97 *1>`". Here "`97`" is the character code which happens to be the ASCII code of the letter 'a' and "`*1`" is a font reference that stands for "Computer Modern Roman 10pt". Reference numbers, as you can see, start with an asterisk reminiscent of references in the C programming language. The Astrix enables us to distinguish between ordinary numbers like "`1`" and references like "`*1`".

To make this node more readable, we will see in section 2.3 that it is also possible to write " `<glyph 'a' (cmr10) *1>`". The latter form uses a comment "`(cmr10)`", enclosed in parentheses, to give an indication of what kind of font happens to be font 1, and it uses "`'a'`", the character enclosed in single quotes to denote the

ASCII code of 'a'. But let's keep things simple for now and stick with the decimal notation of the character code.

The rest is common for all nodes: a keyword, here "`glyph`", and a pair of pointed brackets "`<`...`>`".

Internally, we represent a glyph by the font number and the character number or character code. To store the internal representation of a glyph node, we define an appropriate structure type, named after the node with an uppercase first letter.

⟨ hint types 1 ⟩ ≡ (1)

typedef struct { **uint32_t** *c*; **uint8_t** *f*; } **Glyph**;

Used in 548, 550, and 557.

Let us now look at the program `shrink` and see how it will convert the long format description to the internal representation of the glyph and finally to a short format description.

1.2 Scanning the Long Format

First, `shrink` reads the input file and extracts a sequence of tokens. This is called "scanning". We generate the procedure to do the scanning using the program `flex`[12] which is the GNU version of the common UNIX tool `lex`[13].

The input to `flex` is a list of pattern/action rules where the pattern is a regular expression and the action is a piece of C code. Most of the time, the C code is very simple: it just returns the right token number to the parser which we consider shortly.

The code that defines the tokens will be marked with a line ending in "$---\Longrightarrow$". This symbol stands for "*Reading the long format*". These code sequences define the syntactical elements of the long format and at the same time implement the reading process. All sections where that happens are preceded by a similar heading and for reference they are conveniently listed together starting on page 237.

Reading the Long Format: $---\Longrightarrow$

⟨ symbols 2 ⟩ ≡ (2)

%**token** START `"<"`
%**token** END `">"`
%**token** GLYPH `"glyph"`
%**token** $< u >$ UNSIGNED
%**token** $< u >$ REFERENCE

Used in 553.

You might notice that a small caps font is used for START, END or GLYPH. These are "terminal symbols" or "tokens". Next are the scanning rules which define the connection between tokens and their textual representation.

⟨ scanning rules 3 ⟩ ≡ (3)

```
"<"          SCAN_START; return START;
">"          SCAN_END; return END;
glyph        return GLYPH;
```

```
0|[1-9][0-9]*   SCAN_UDEC(yytext); return UNSIGNED;
\*(0|[1-9][0-9]*)   SCAN_UDEC(yytext + 1); return REFERENCE;
[[:space:]]   ;
\([^()\n]*[)\n]   ;
```

Used in 552.

As we will see later, the macros starting with SCAN_... are scanning macros. Here SCAN_UDEC is a macro that converts the decimal representation that did match the given pattern to an unsigned integer value; it is explained in section 2.1. The macros SCAN_START and SCAN_END are explained in section 4.2.

The action “;” is a “do nothing” action; here it causes spaces or comments to be ignored. Comments start with an opening parenthesis and are terminated by a closing parenthesis or the end of line character. The pattern “[^()\n]” is a negated character class that matches all characters except parentheses and the newline character. These are not allowed inside comments. For detailed information about the patterns used in a `flex` program, see the `flex` user manual[12].

1.3 Parsing the Long Format

Next, the tokens produced by the scanner are assembled into larger entities. This is called “parsing”. We generate the procedure to do the parsing using the program `bison`[12] which is the GNU version of the common UNIX tool `yacc`[13].

The input to `bison` is a list of parsing rules, called a “grammar”. The rules describe how to build larger entities from smaller entities. For a simple glyph node like “ `<glyph 97 *1>`”, we need just these rules:

Reading the Long Format: − − − ⟹

⟨ symbols 2 ⟩ +≡ (4)
%**type** < *u* > *start*
%**type** < *c* > *glyph*

⟨ parsing rules 5 ⟩ ≡ (5)

glyph: UNSIGNED REFERENCE
  { $$.*c* = $1; REF(*font_kind*, $2); $$.*f* = $2; };
content_node: *start* GLYPH *glyph* END { *hput_tags*($1, *hput_glyph*(&($3))); };
start: START { HPUTNODE; $$ = (**uint32_t**)(*hpos*++ − *hstart*); }

Used in 553.

You might notice that a slanted font is used for *glyph*, *content_node*, or *start*. These are “nonterminal symbols’ and occur on the left hand side of a rule. On the right hand side of a rule you find nonterminal symbols, as well as terminal symbols and C code enclosed in braces.

Within the C code, the expressions $1 and $2 refer to the variables on the parse stack that are associated with the first and second symbol on the right hand side of the rule. In the case of our glyph node, these will be the values 97 and 1, respectively, as produced by the macro SCAN_UDEC. $$ refers to the variable associated with the left hand side of the rule. These variables contain the internal representation of the object in question. The type of the variable is specified by a mandatory **token** or optional **type** clause when we define the symbol. In the above **type** clause for *start* and *glyph* , the identifiers u and c refer to the **union** declaration of the parser (see page 231) where we find **uint32_t** u and **Glyph** c. The macro `REF` tests a reference number for its valid range.

Reading a node is usually split into the following sequence of steps:

- Reading the node specification, here a *glyph* consisting of an UNSIGNED value and a REFERENCE value.
- Creating the internal representation in the variable `$$` based on the values of `$1`, `$2`, ... Here the character code field c is initialized using the UNSIGNED value stored in `$1` and the font field f is initialized using `$2` after checking the reference number for the proper range.
- A *content_node* rule explaining that *start* is followed by GLYPH, the keyword that directs the parser to *glyph*, the node specification, and a final END.
- Parsing *start*, which is defined as the token START will assign to the corresponding variable p on the parse stack the current position *hpos* in the output and increments that position to make room for the start byte, which we will discuss shortly.
- At the end of the *content_node* rule, the `shrink` program calls a *hput_*... function, here *hput_glyph*, to write the short format of the node as given by its internal representation to the output and return the correct tag value.
- Finally the *hput_tags* function will add the tag as a start byte and end byte to the output stream.

Now let's see how writing the short format works in detail.

1.4 Writing the Short Format

A content node in short form begins with a start byte. It tells us what kind of node it is. To describe the content of a short HINT file, 32 different kinds of nodes are defined. Hence the kind of a node can be stored in 5 bits and the remaining bits of the start byte can be used to contain a 3 bit "info" value.

We define an enumeration type to give symbolic names to the kind-values. The exact numerical values are of no specific importance; we will see in section 4.2, however, that the assignment chosen below, has certain advantages.

Because the usage of kind-values in content nodes is slightly different from the usage in definition nodes, we define alternative names for some kind-values. To display readable names instead of numerical values when debugging, we define two arrays of strings as well. Keeping the definitions consistent is achieved by creating all definitions from the same list of identifiers using different definitions of the macro `DEF_KIND`.

⟨ hint basic types 6 ⟩ ≡ (6)

\#**define** `DEF_KIND` (C, D, N) $C\#\#\_kind = N$
  typedef enum { ⟨ kinds 9 ⟩ , ⟨ alternative kind names 10 ⟩ } **Kind**;
\#**undef** `DEF_KIND`

Used in 545.

⟨ define *content_name* and *definition_name* 7 ⟩ ≡ (7)

```
#define DEF_KIND (C, D, N) #C
  const char *content_name[32] = { ⟨ kinds 9 ⟩ } ;
#undef DEF_KIND
#define DEF_KIND (C, D, N) #D
  const char *definition_name[#20] = { ⟨ kinds 9 ⟩ } ;
#undef DEF_KIND
```

Used in 546.

⟨ print *content_name* and *definition_name* 8 ⟩ ≡ (8)

```
  printf("const␣char␣*content_name[32]={");
  for (k = 0; k ≤ 31; k++) { printf("\"%s\"", content_name[k]);
    if (k < 31) printf(",␣");
  }
  printf("};\n\n"); printf("const␣char␣*definition_name[32]={");
  for (k = 0; k ≤ 31; k++) { printf("\"%s\"", definition_name[k]);
    if (k < 31) printf(",␣");
  }
  printf("};\n\n");
```

Used in 546.

⟨ kinds 9 ⟩ ≡ (9)

```
  DEF_KIND(list, list, 0),
  DEF_KIND(param, param, 1),
  DEF_KIND(range, range, 2),
  DEF_KIND(xdimen, xdimen, 3),
  DEF_KIND(adjust, adjust, 4),
  DEF_KIND(glyph, font, 5),
  DEF_KIND(kern, dimen, 6),
  DEF_KIND(glue, glue, 7),
  DEF_KIND(ligature, ligature, 8),
  DEF_KIND(disc, disc, 9),
  DEF_KIND(language, language, 10),
  DEF_KIND(rule, rule, 11),
  DEF_KIND(image, image, 12),
  DEF_KIND(leaders, leaders, 13),
  DEF_KIND(baseline, baseline, 14),
  DEF_KIND(hbox, hbox, 15),
  DEF_KIND(vbox, vbox, 16),
  DEF_KIND(par, par, 17),
  DEF_KIND(math, math, 18),
  DEF_KIND(table, table, 19),
  DEF_KIND(item, item, 20),
  DEF_KIND(hset, hset, 21),
  DEF_KIND(vset, vset, 22),
  DEF_KIND(hpack, hpack, 23),
```

```
DEF_KIND(vpack, vpack, 24),
DEF_KIND(stream, stream, 25),
DEF_KIND(page, page, 26),
DEF_KIND(link, label, 27),
DEF_KIND(color, color, 28),
DEF_KIND(undefined1, undefined1, 29),
DEF_KIND(undefined2, undefined2, 30),
DEF_KIND(penalty, int, 31)
```

Used in 6 and 7.

For a few kind-values we have alternative names; we will use them to express different intentions when using them.

⟨ alternative kind names 10 ⟩ ≡ (10)

font_kind = *glyph_kind*, *int_kind* = *penalty_kind*, *unknown_kind* = *penalty_kind*, *dimen_kind* = *kern_kind*, *label_kind* = *link_kind*, *outline_kind* = *link_kind*

Used in 6.

The info values can be used to represent numbers in the range 0 to 7; for an example see the *hput_glyph* function later in this section. Mostly, however, the individual bits are used as flags indicating the presence or absence of immediate parameter values. If the info bit is set, it means the corresponding parameter is present as an immediate value; if it is zero, it means that there is no immediate parameter value present, and the node specification will reveal what value to use instead. In some cases there is a common default value that can be used, in other cases a one byte reference number is used to select a predefined value.

To make the binary representation of the info bits more readable, we define an enumeration type.

⟨ hint basic types 6 ⟩ +≡ (11)

typedef enum { *b000* = 0, *b001* = 1, *b010* = 2, *b011* = 3, *b100* = 4, *b101* = 5, *b110* = 6, *b111* = 7 } **Info**;

After the start byte follows the node content and it is the purpose of the start byte to reveal the exact syntax and semantics of the node content. Because we want to be able to read the short form of a HINT file in forward direction and in backward direction, the start byte is duplicated after the content as an end byte.

We store a kind and an info value in one byte and call this a tag.

⟨ hint basic types 6 ⟩ +≡ (12)

typedef uint8_t Tag;

The following macros are used to assemble and disassemble tags:

⟨ hint macros 13 ⟩ ≡ (13)

```
#define KIND(T) (((T) >> 3) & #1F)
#define NAME(T) content_name[KIND(T)]
#define INFO(T) ((T) & #7)
#define TAG(K,I) (((K) << 3) | (I))
```

Used in 545 and 550.

Writing a short format HINT file is implemented by a collection of *hput_*... functions; they follow most of the time the same schema:

- First, we define a variable for *info*.
- Then follows the main part of the function body, where we decide on the output format, do the actual output and set the *info* value accordingly.
- We combine the info value with the kind-value and return the correct tag.
- The tag value will be passed to *hput_tags* which generates debugging information, if requested, and stores the tag before and after the node content.

After these preparations, we turn our attention again to the *hput_glyph* function. The font number in a glyph node is between 0 and 255 and fits nicely in one byte, but the character code is more difficult: we want to store the most common character codes as a single byte and less frequent codes with two, three, or even four byte. Naturally, we use the *info* bits to store the number of bytes needed for the character code.

Writing the Short Format: ⟹ ...

```
⟨ put functions 14 ⟩ ≡                                                    (14)
  static uint8_t hput_n(uint32_t n)
  { if (n ≤ #FF) { HPUT8(n); return 1; }
    else if (n ≤ #FFFF) { HPUT16(n); return 2; }
    else if (n ≤ #FFFFFF) { HPUT24(n); return 3; }
    else { HPUT32(n); return 4; }
  }
  Tag hput_glyph(Glyph *g)
  { Info info;
    info = hput_n(g→c); HPUT8(g→f);
    return TAG(glyph_kind, info);
  }
```

Used in 551 and 554.

The *hput_tags* function is called after the node content has been written to the stream. It gets a the position of the start byte and the tag. With this information it writes the start byte at the given position and the end byte at the current stream position.

```
⟨ put functions 14 ⟩ +≡                                                   (15)
  void hput_tags(uint32_t pos, Tag tag)
  { DBGTAG(tag, hstart + pos); DBGTAG(tag, hpos); HPUTX(1);
    *(hstart + pos) = *(hpos++) = tag; }
```

The variables *hpos* and *hstart*, the macros HPUT8, HPUT16, HPUT24, HPUT32, and HPUTX are all defined in section 8.3; they put 8, 16, 24, or 32 bits into the output stream and check for sufficient space in the output buffer. The macro DBGTAG writes debugging output; its definition is found in section 14.

Now that we have seen the general outline of the `shrink` program, starting with a long format file and ending with a short format file, we will look at the program `stretch` that reverses this transformation.

1.5 Parsing the Short Format

The inverse of writing the short format with a *hput_...* function is reading the short format with a *hget_...* function.

The schema of *hget_...* functions reverse the schema of *hput_...* functions. Here is the code for the initial and final part of a get function:

⟨ read the start byte a 16 ⟩ ≡ (16)

```
Tag a, z;                                    /* the start and the end byte */
uint32_t node_pos = hpos − hstart;
if (hpos ≥ hend)
  QUIT("Attempt␣to␣read␣a␣start␣byte␣at␣the␣end␣of␣the␣section");
HGETTAG(a);
```

Used in 18, 94, 121, 139, 146, 158, 167, 202, 313, 369, 388, 396, and 410.

⟨ read and check the end byte z 17 ⟩ ≡ (17)

```
HGETTAG(z); if (a ≠ z)
  QUIT("Tag␣mismatch␣[%s,%d]!=[%s,%d]␣at␣0x%x␣to␣"SIZE_F"\n",
      NAME(a), INFO(a), NAME(z), INFO(z), node_pos, hpos − hstart − 1);
```

Used in 18, 94, 121, 139, 146, 158, 167, 202, 313, 369, 388, 396, and 410.

The central routine to parse the content section of a short format file is the function *hget_content_node* which calls *hget_content* to do most of the processing.

hget_content_node will read a content node in short format and write it out in long format: It reads the start byte a, writes the START token using the function *hwrite_start*, and based on KIND(a), it writes the node's keyword found in the *content_name* array. Then it calls *hget_content* to read the node's content and write it out. Finally it reads the end byte, checks it against the start byte, and finishes up the content node by writing the END token using the *hwrite_end* function. The function returns the tag byte so that the calling function might check that the content node meets its requirements.

hget_content uses the start byte a, passed as a parameter, to branch directly to the reading routine for the given combination of kind and info value. The reading routine will read the data and store its internal representation in a variable. All that the `stretch` program needs to do with this internal representation is writing it in the long format. As we will see, the call to the proper *hwrite_...* function is included as final part of the the reading routine (avoiding another switch statement).

Reading the Short Format: $\cdots \Longrightarrow$

⟨ get functions 18 ⟩ ≡ (18)

```
static void hget_content(Tag a);

Tag hget_content_node(void)
{ ⟨ read the start byte a 16 ⟩ hwrite_start();
  if (content_known[KIND(a)] & (1 ≪ INFO(a)))
    hwritef("%s", content_name[KIND(a)]);
  hget_content(a);
  ⟨ read and check the end byte z 17 ⟩
  hwrite_end(); return a;
}

static void hget_content(Tag a)
{ switch (a)
  { ⟨ cases to get content 20 ⟩
    default:
      if (¬hget_unknown(a)) TAGERR(a);
      break;
  }
}
```

Used in 555 and 557.

We implement the code to read a glyph node in two stages. First we define a general reading macro HGET_GLYPH(I, G) that reads a glyph node with info value I into a **Glyph** variable G; then we insert this macro in the above switch statement for all cases where it applies. Knowing the function *hput_glyph*, the macro HGET_GLYPH should not be a surprise. It reverses *hput_glyph*, storing the glyph node in its internal representation. After that, the `stretch` program calls *hwrite_glyph* to produce the glyph node in long format.

Reading the Short Format: $\cdots \Longrightarrow$

⟨ get macros 19 ⟩ ≡ (19)

```
#define HGET_N(I, X)
  if ((I) ≡ 1) (X) = HGET8;
  else if ((I) ≡ 2) HGET16(X);
  else if ((I) ≡ 3) HGET24(X);
  else if ((I) ≡ 4) HGET32(X);
#define HGET_GLYPH(I, G) HGET_N (I, (G).c); (G).f = HGET8;
  REF_RNG(font_kind, (G).f);
  hwrite_glyph(&(G));
```

Used in 555 and 557.

Note that we allow a glyph to reference a font even before that font is defined. This is necessary because fonts usually contain definitions—for example the fonts hyphen character—that reference this or other fonts.

⟨ cases to get content 20 ⟩ ≡ (20)

case TAG(*glyph_kind*, 1): { **Glyph** *g*; HGET_GLYPH(1, *g*); } **break**;
case TAG(*glyph_kind*, 2): { **Glyph** *g*; HGET_GLYPH(2, *g*); } **break**;
case TAG(*glyph_kind*, 3): { **Glyph** *g*; HGET_GLYPH(3, *g*); } **break**;
case TAG(*glyph_kind*, 4): { **Glyph** *g*; HGET_GLYPH(4, *g*); } **break**;

Used in 18.

If this two stage method seems strange to you, consider what the C compiler will do with it. It will expand the HGET_GLYPH macro four times inside the switch statement. The macro is, however, expanded with a constant I value, so the expansion of the **if** statement in HGET_GLYPH$(1, g)$, for example, will become "**if** $(1 \equiv 1) \ldots$ **else if** $(1 \equiv 2) \ldots$" and the compiler will have no difficulties eliminating the constant tests and the dead branches altogether. This is the most effective use of the switch statement: a single jump takes you to a specialized code to handle just the given combination of kind and info value.

Last not least, we implement the function *hwrite_glyph* to write a glyph node in long form—that is: in a form that is as readable as possible.

1.6 Writing the Long Format

The *hwrite_glyph* function inverts the scanning and parsing process we have described at the very beginning of this chapter. To implement the *hwrite_glyph* function, we use the function *hwrite_charcode* to write the character code. Besides writing the character code as a decimal number, this function can handle also other representations of character codes as fully explained in section 2.3. We split off the writing of the opening and the closing pointed bracket, because we will need this function very often and because it will keep track of the *nesting* of nodes and indent them accordingly. The *hwrite_range* and *hwrite_label* functions used in *hwrite_end* are discussed in section 7.4 and 6.2.

Writing the Long Format: $\Longrightarrow$ − − −

⟨ write functions 21 ⟩ ≡ (21)

```
int nesting = 0;

void hwrite_nesting(void)
{ int i;
  hwritec('\n');
  for (i = 0; i < nesting; i++) hwritec('␣');
}

void hwrite_start(void)
{ hwrite_nesting(); hwritec('<'); nesting++;
}

void hwrite_range(void);
void hwrite_label(void);

void hwrite_end(void)
{ nesting--; hwritec('>');
  if (section_no ≡ 2) {
```

```
      if (nesting ≡ 0) hwrite_range( );
      hwrite_label( );
    }
  }
  void hwrite_comment(char *str)
  { char c;
    if (str ≡ NULL) return;
    hwritef(" (");
    while ((c = *str++) ≠ 0)
      if (c ≡ '(' ∨ c ≡ ')') hwritec('_');
      else if (c ≡ '\n') hwritef("\n(");
      else hwritec(c);
    hwritec(')');
  }
  void hwrite_charcode(uint32_t c);
  void hwrite_ref(int n);
  void hwrite_glyph(Glyph *g)
  { char *n = hfont_name[g→f];
    hwrite_charcode(g→c); hwrite_ref(g→f);
    if (n ≠ NULL) hwrite_comment(n);
  }
```

Used in 555 and 557.

The two primitive operations to write the long format file are defined as macros:

⟨ write macros 22 ⟩ ≡ (22)

```
#define hwritec(c) (hout ? putc(c, hout) : 0)
#define hwritef(...) (hout ? fprintf(hout, __VA_ARGS__) : 0)
```

Used in 555 and 557.

Now that we have completed the round trip of shrinking and stretching glyph nodes, we continue the description of the HINT file formats in a more systematic way.

2 Data Types

2.1 Integers

We have already seen the pattern/action rule for unsigned decimal numbers. It remains to define the macro `SCAN_UDEC` which converts a string containing an unsigned decimal number into an unsigned integer. We use the C library function *strtoul*:

Reading the long format: $- - - \Longrightarrow$

⟨scanning macros 23⟩ ≡ (23)

#define `SCAN_UDEC`(S) $yylval.u = strtoul(S, \texttt{NULL}, 10)$

Used in 552.

Unsigned integers can be given in hexadecimal notation as well.

⟨scanning definitions 24⟩ ≡ (24)

```
HEX                [0-9A-F]
```

Used in 552.

⟨scanning rules 3⟩ +≡ (25)

`0x{HEX}+` `SCAN_HEX`$(yytext + 2)$; **return** UNSIGNED;

Note that the pattern above allows only upper case letters in the hexadecimal notation for integers.

⟨scanning macros 23⟩ +≡ (26)

#define `SCAN_HEX`(S) $yylval.u = strtoul(S, \texttt{NULL}, 16)$

Last not least, we add rules for signed integers.

⟨symbols 2⟩ +≡ (27)

%**token** $< i >$ SIGNED

%**type** $< i >$ *integer*

⟨scanning rules 3⟩ +≡ (28)

`[+-](0|[1-9][0-9]*)` `SCAN_DEC`$(yytext)$; **return** SIGNED;

⟨scanning macros 23⟩ +≡ (29)

#define `SCAN_DEC`(S) $yylval.i = strtol(S, \texttt{NULL}, 10)$

⟨ parsing rules 5 ⟩ +≡ (30)

integer: SIGNED | UNSIGNED { RNG("number", $1, 0, #7FFFFFFF); };

To preserve the "signedness" of an integer also for positive signed integers in the long format, we implement the function *hwrite_signed*.

Writing the long format: ⟹ − − −

⟨ write functions 21 ⟩ +≡ (31)

```
void hwrite_signed(int32_t i)
{
  if (i < 0) hwritef("␣-%d", -i);
  else hwritef("␣+%d", +i);
}
```

Reading and writing integers in the short format is done directly with the HPUT and HGET macros.

2.2 Strings

Strings are needed in the definition part of a HINT file to specify names of objects, and in the long file format, we also use them for file names. In the long format, strings are sequences of characters delimited by single quote characters; for example: "'Hello'" or "'cmr10-600dpi.tfm'"; in the short format, strings are byte sequences terminated by a zero byte. Because file names are system dependent, we no not allow arbitrary characters in strings but only printable ASCII codes which we can reasonably expect to be available on most operating systems. If your file names in a long format HINT file are supposed to be portable, you should probably be even more restrictive. For example you should avoid characters like "\" or "/" which are used in different ways for directories.

The internal representation of a string is a simple zero terminated C string. When scanning a string, we copy it to the *str_buffer* keeping track of its length in *str_length*. When done, we make a copy for permanent storage and return the pointer to the parser. To operate on the *str_buffer*, we define a few macros. The constant MAX_STR determines the maximum size of a string (including the zero byte) to be 2^{10} byte. This restriction is part of the HINT file format specification.

⟨ scanning macros 23 ⟩ +≡ (32)

```
#define MAX_STR (1 << 10)    /* 2^10 Byte or 1kByte */
  static char str_buffer[MAX_STR];
  static int str_length;
#define STR_START (str_length = 0)
#define STR_PUT(C) (str_buffer[str_length++] = (C))
#define STR_ADD(C)
  STR_PUT(C); RNG("String␣length", str_length, 0, MAX_STR - 1)
#define STR_END str_buffer[str_length] = 0
#define SCAN_STR yylval.s = str_buffer
```

To scan a string, we switch the scanner to STR mode when we find a quote character, then we scan bytes in the range #20 to #7E, which is the range of printable ASCII characters, until we find the closing single quote. Quote characters inside the string are written as two consecutive single quote characters.

Reading the long format: − − − ⟹

⟨scanning definitions 24⟩ +≡ (33)

```
%x STR
```

⟨symbols 2⟩ +≡ (34)

```
%token <s> STRING
```

⟨scanning rules 3⟩ +≡ (35)

```
'               STR_START; BEGIN(STR);
<STR>{
'               STR_END; SCAN_STR; BEGIN(INITIAL); return STRING;
''              STR_ADD('\'');
[\x20-\x7E]     STR_ADD(yytext[0]);
.               RNG("String character", yytext[0], #20, #7E);
\n              QUIT("Unterminated String in line %d", yylineno);
}
```

The function *hwrite_string* reverses this process; it must take care of the quote symbols.

Writing the long format: ⟹ − − −

⟨write functions 21⟩ +≡ (36)

```
void hwrite_string(char *str)
{ hwritec(' ');
  if (str ≡ NULL) hwritef("''");
  else
  { hwritec('\'');
    while (*str ≠ 0)
    { if (*str ≡ '\'') hwritec('\'');
      hwritec(*str);
      str++;
    }
    hwritec('\'');
  }
}
```

In the short format, a string is just a byte sequence terminated by a zero byte. This makes the function *hput_string*, to write a string, and the macro HGET_STRING, to read a string in short format, very simple. Note that after writing an unbounded string to the output buffer, the macro HPUTNODE will make sure that there is enough space left to write the remainder of the node.

Writing the short format: ⟹ ⋯

```
⟨ put functions 14 ⟩ +≡                                              (37)
  void hput_string(char *str)
  { char *s = str;
    if (s ≠ NULL) { do { HPUTX(1);
          HPUT8(*s);
       } while (*s++ ≠ 0);
       HPUTNODE;
    }
    else HPUT8(0);
  }
```

Reading the short format: ⋯ ⟹

```
⟨ shared get macros 38 ⟩ ≡                                           (38)
#define HGET_STRING(S) S = (char *) hpos;
  while (hpos < hend ∧ *hpos ≠ 0) {
    RNG("String␣character", *hpos, #20, #7E);
    hpos++;
  }
  hpos++;
```

Used in 548 and 557.

2.3 Character Codes

We have already seen in the introduction that character codes can be written as decimal numbers and section 2.1 adds the possibility to use hexadecimal numbers as well.

It is, however, in most cases more readable if we represent character codes directly using the characters themselves. Writing "a" is just so much better than writing "97". To distinguish the character "9" from the number "9", we use the common technique of enclosing characters within single quotes. So "'9'" is the character code and "9" is the number. Therefore we will define CHARCODE tokens and complement the parsing rules of section 1.3 with the following rule:

Reading the long format: − − − ⟹

```
⟨ parsing rules 5 ⟩ +≡                                               (39)
  glyph: CHARCODE REFERENCE
         { $$.c = $1; REF(font_kind, $2); $$.f = $2; };
```

If the character codes are small, we can represent them using ASCII character codes. We do not offer a special notation for very small character codes that map to the non-printable ASCII control codes; for them, the decimal or hexadecimal notation will suffice. For larger character codes, we use the multibyte encoding scheme known from UTF8 as follows. Given a character code c:

- Values in the range #00 to #7f are encoded as a single byte with a leading bit of 0.

⟨ scanning definitions 24 ⟩ +≡ (40)
```
UTF8_1          [\x00-\x7F]
```

⟨ scanning macros 23 ⟩ +≡ (41)
#define `SCAN_UTF8_1`(S) $\mathit{yylval.u} = ((S)[0]\ \&\ {}^{\#}\mathtt{7F})$

- Values in the range $^{\#}$`80` to $^{\#}$`7ff` are encoded in two byte with the first byte having three high bits 110, indicating a two byte sequence, and the lower five bits equal to the five high bits of c. It is followed by a continuation byte having two high bits 10 and the lower six bits equal to the lower six bits of c.

⟨ scanning definitions 24 ⟩ +≡ (42)
```
UTF8_2          [\xC0-\xDF][\x80-\xBF]
```

⟨ scanning macros 23 ⟩ +≡ (43)
#define `SCAN_UTF8_2`(S) $\mathit{yylval.u} = (((S)[0]\ \&\ {}^{\#}\mathtt{1F}) \ll 6) + ((S)[1]\ \&\ {}^{\#}\mathtt{3F})$

- Values in the range $^{\#}$`800` to $^{\#}$`FFFF` are encoded in three byte with the first byte having the high bits 1110 indicating a three byte sequence followed by two continuation bytes.

⟨ scanning definitions 24 ⟩ +≡ (44)
```
UTF8_3          [\xE0-\xEF][\x80-\xBF][\x80-\xBF]
```

⟨ scanning macros 23 ⟩ +≡ (45)
#define `SCAN_UTF8_3`(S)
$\mathit{yylval.u} = (((S)[0]\ \&\ {}^{\#}\mathtt{0F}) \ll 12) + (((S)[1]\ \&\ {}^{\#}\mathtt{3F}) \ll 6) + ((S)[2]\ \&\ {}^{\#}\mathtt{3F})$

- Values in the range $^{\#}$`1000` to $^{\#}$`1FFFFF` are encoded in four byte with the first byte having the high bits 11110 indicating a four byte sequence followed by three continuation bytes.

⟨ scanning definitions 24 ⟩ +≡ (46)
```
UTF8_4          [\xF0-\xF7][\x80-\xBF][\x80-\xBF][\x80-\xBF]
```

⟨ scanning macros 23 ⟩ +≡ (47)
#define `SCAN_UTF8_4`(S)
$\mathit{yylval.u} = (((S)[0]\ \&\ {}^{\#}\mathtt{03}) \ll 18) + (((S)[1]\ \&\ {}^{\#}\mathtt{3F}) \ll 12) +$
$\quad (((S)[2]\ \&\ {}^{\#}\mathtt{3F}) \ll 6) + ((S)[3]\ \&\ {}^{\#}\mathtt{3F})$

In the long format file, we enclose a character code in single quotes, just as we do for strings. This is convenient but it has the downside that we must exercise special care when giving the scanning rules in order not to confuse character codes with strings. Further we must convert character codes back into strings in the rare case where the parser expects a string and gets a character code because the string was only a single character long.

Let's start with the first problem: The scanner might confuse a string and a character code if the first or second character of the string is a quote character which is written as two consecutive quotes. For example `'a''b'` is a string with three characters, "`a`", "`'`", and "`b`". Two character codes would need a space to separate them like this: `'a' 'b'`.

⟨symbols 2⟩ +≡ (48)
```
%token <u> CHARCODE
```

⟨scanning rules 3⟩ +≡ (49)
```
'''           STR_START; STR_PUT('\''); BEGIN(STR);
''''          SCAN_UTF8_1(yytext + 1); return CHARCODE;
'[\x20-\x7E]''  STR_START; STR_PUT(yytext[1]); STR_PUT('\''); BEGIN(STR);
'''''         STR_START; STR_PUT('\''); STR_PUT('\''); BEGIN(STR);
'{UTF8_1}'    SCAN_UTF8_1(yytext + 1); return CHARCODE;
'{UTF8_2}'    SCAN_UTF8_2(yytext + 1); return CHARCODE;
'{UTF8_3}'    SCAN_UTF8_3(yytext + 1); return CHARCODE;
'{UTF8_4}'    SCAN_UTF8_4(yytext + 1); return CHARCODE;
```

If needed, the parser can convert character codes back to single character strings.

⟨symbols 2⟩ +≡ (50)
```
%type <s> string
```

⟨parsing rules 5⟩ +≡ (51)
```
string: STRING | CHARCODE { static char s[2];
    RNG("String element", $1, #20, #7E); s[0] = $1; s[1] = 0; $$ = s; };
```

The function *hwrite_charcode* will write a character code. While ASCII codes are handled directly, larger character codes are passed to the function *hwrite_utf8*. It returns the number of characters written.

Writing the long format: ⟹ − − −

⟨write functions 21⟩ +≡ (52)
```
int hwrite_utf8(uint32_t c)
{ if (c < #80) { hwritec(c); return 1; }
  else if (c < #800)
  { hwritec(#C0 | (c >> 6)); hwritec(#80 | (c & #3F)); return 2; }
  else if (c < #10000)
  { hwritec(#E0 | (c >> 12));
    hwritec(#80 | ((c >> 6) & #3F)); hwritec(#80 | (c & #3F));
    return 3;
  }
  else if (c < #200000)
  { hwritec(#F0 | (c >> 18)); hwritec(#80 | ((c >> 12) & #3F));
    hwritec(#80 | ((c >> 6) & #3F)); hwritec(#80 | (c & #3F));
    return 4;
  }
  else RNG("character code", c, 0, #1FFFFF);
  return 0;
}
```

```
void hwrite_charcode(uint32_t c)
{ if (c < #20) {
    if (option_hex) hwritef(" 0x%02X", c);          /* non printable ASCII */
    else hwritef(" %u", c);
  }
  else if (c ≡ '\'') hwritef(" ''''");
  else if (c ≤ #7E) hwritef(" \'%c\'", c);           /* printable ASCII */
  else if (option_utf8) { hwritef(" \'"); hwrite_utf8(c); hwritec('\''); }
  else if (option_hex) hwritef(" 0x%04X", c);
  else hwritef(" %u", c);
}
```

Reading the short format: ⋯ ⟹

⟨ shared get functions 53 ⟩ ≡ (53)

```
#define HGET_UTF8C(X) (X) = HGET8; if ((X & #C0) ≠ #80)
    QUIT("UTF8 continuation byte expected at "SIZE_F" got 0x%02X\n",
        hpos − hstart − 1, X)
uint32_t hget_utf8(void)
{ uint8_t a;
  a = HGET8;
  if (a < #80) return a;
  else {
    if ((a & #E0) ≡ #C0)
    { uint8_t b; HGET_UTF8C(b);
      return ((a & ∼#E0) ≪ 6) + (b & ∼#C0);
    }
    else if ((a & #F0) ≡ #E0)
    { uint8_t b, c; HGET_UTF8C(b); HGET_UTF8C(c);
      return ((a & ∼#F0) ≪ 12) + ((b & ∼#C0) ≪ 6) + (c & ∼#C0);
    }
    else if ((a & #F8) ≡ #F0)
    { uint8_t b, c, d; HGET_UTF8C(b); HGET_UTF8C(c); HGET_UTF8C(d);
      return ((a & ∼#F8) ≪ 18)
          + ((b & ∼#C0) ≪ 12) + ((c & ∼#C0) ≪ 6) + (d & ∼#C0);
    }
    else QUIT("UTF8 byte sequence expected");
  }
}
```

Used in 549, 555, and 557.

Writing the short format: ⟹ ···

⟨ put functions 14 ⟩ +≡ (54)

```
void hput_utf8(uint32_t c)
{ HPUTX(4);
  if (c < #80) HPUT8(c);
  else if (c < #800) { HPUT8(#C0 | (c >> 6)); HPUT8(#80 | (c & #3F)); }
  else if (c < #10000)
  { HPUT8(#E0 | (c >> 12));
    HPUT8(#80 | ((c >> 6) & #3F)); HPUT8(#80 | (c & #3F));
  }
  else if (c < #200000)
  { HPUT8(#F0 | (c >> 18)); HPUT8(#80 | ((c >> 12) & #3F));
    HPUT8(#80 | ((c >> 6) & #3F)); HPUT8(#80 | (c & #3F));
  }
  else RNG("character code", c, 0, #1FFFFF);
}
```

2.4 Floating Point Numbers

You know a floating point numbers when you see it because it features a radix point. The optional exponent allows you to “float” the point.

Reading the long format: − − − ⟹

⟨ symbols 2 ⟩ +≡ (55)

```
%token < f > FPNUM
%type < f > number
```

⟨ scanning rules 3 ⟩ +≡ (56)

```
[+-]?[0-9]+\.[0-9]+(e[+-]?[0-9])?   SCAN_DECFLOAT; return FPNUM;
```

The layout of floating point variables of type **double** or **float** typically follows the IEEE754 standard[6][7]. We use the following definitions:

⟨ hint basic types 6 ⟩ +≡ (57)

```
#define FLT_M_BITS 23
#define FLT_E_BITS 8
#define FLT_EXCESS 127
#define DBL_M_BITS 52
#define DBL_E_BITS 11
#define DBL_EXCESS 1023
```

⟨ scanning macros 23 ⟩ +≡ (58)

```
#define SCAN_DECFLOAT yylval.f = atof(yytext)
```

When the parser expects a floating point number and gets an integer number, it converts it. So whenever in the long format a floating point number is expected, an integer number will do as well.

⟨ parsing rules 5 ⟩ +≡ (59)

```
number: UNSIGNED { $$ = (float64_t) $1; }
      | SIGNED { $$ = (float64_t) $1; }
      | FPNUM;
```

Unfortunately the decimal representation is not optimal for floating point numbers since even simple numbers in decimal notation like 0.1 do not have an exact representation as a binary floating point number. So if we want a notation that allows an exact representation of binary floating point numbers, we must use a hexadecimal representation. Hexadecimal floating point numbers start with an optional sign, then as usual the two characters "0x", then follows a sequence of hex digits, a radix point, more hex digits, and an optional exponent. The optional exponent starts with the character "x", followed by an optional sign, and some more hex digits. The hexadecimal exponent is given as a base 16 number and it is interpreted as an exponent with the base 16. As an example an exponent of "x10", would multiply the mantissa by 16^{16}. In other words it would shift any mantissa 16 hexadecimal digits to the left. Here are the exact rules:

⟨ scanning rules 3 ⟩ +≡ (60)

```
[+-]?0x{HEX}+\.{HEX}+(x[+-]?{HEX}+)?  SCAN_HEXFLOAT; return FPNUM;
```

⟨ scanning macros 23 ⟩ +≡ (61)

```
#define SCAN_HEXFLOAT yylval.f = xtof(yytext)
```

There is no function in the C library for hexadecimal floating point notation so we have to write our own conversion routine. The function *xtof* converts a string matching the above regular expression to its binary representation. Its outline is very simple:

⟨ scanning functions 62 ⟩ ≡ (62)

```
float64_t xtof(char *x)
{ int sign, digits, exp;
  uint64_t mantissa = 0;
  DBG(DBGFLOAT, "converting␣%s:\n", x);
  ⟨ read the optional sign 63 ⟩
  x = x + 2;                                    /* skip "0x" */
  ⟨ read the mantissa 64 ⟩
  ⟨ normalize the mantissa 65 ⟩
  ⟨ read the optional exponent 66 ⟩
  ⟨ return the binary representation 67 ⟩
}
```

Used in 552.

Now the pieces:

⟨ read the optional sign 63 ⟩ ≡ (63)

```
if (*x ≡ '-') { sign = -1; x++; }
else if (*x ≡ '+') { sign = +1; x++; }
else sign = +1;
```

```
DBG(DBGFLOAT, "\tsign=%d\n", sign);
```

Used in 62.

When we read the mantissa, we use the temporary variable *mantissa*, keep track of the number of digits, and adjust the exponent while reading the fractional part.

⟨ read the mantissa 64 ⟩ ≡ (64)

```
digits = 0;
while (*x ≡ '0') x++;                    /* ignore leading zeros */
while (*x ≠ '.')
{ mantissa = mantissa ≪ 4;
  if (*x < 'A') mantissa = mantissa + *x − '0';
  else mantissa = mantissa + *x − 'A' + 10;
  x++;
  digits++;
}
x++;                                      /* skip "." */
exp = 0;
while (*x ≠ 0 ∧ *x ≠ 'x')
{ mantissa = mantissa ≪ 4;
  exp = exp − 4;
  if (*x < 'A') mantissa = mantissa + *x − '0';
  else mantissa = mantissa + *x − 'A' + 10;
  x++;
  digits++;
}
DBG(DBGFLOAT, "\tdigits=%d␣mantissa=0x%" PRIx64 ",␣exp=%d\n",
    digits, mantissa, exp);
```

Used in 62.

To normalize the mantissa, first we shift it to place exactly one nonzero hexadecimal digit to the left of the radix point. Then we shift it right bit-wise until there is just a single 1 bit to the left of the radix point. To compensate for the shifting, we adjust the exponent accordingly. Finally we remove the most significant bit because it is not stored.

⟨ normalize the mantissa 65 ⟩ ≡ (65)

```
if (mantissa ≡ 0) return 0.0;
{ int s;
  s = digits − DBL_M_BITS/4;
  if (s > 1) mantissa = mantissa ≫ (4 * (s − 1));
  else if (s < 1) mantissa = mantissa ≪ (4 * (1 − s));
  exp = exp + 4 * (digits − 1);
  DBG(DBGFLOAT, "\tdigits=%d␣mantissa=0x%" PRIx64 ",␣exp=%d\n",
      digits, mantissa, exp);
  while ((mantissa ≫ DBL_M_BITS) > 1)
  { mantissa = mantissa ≫ 1; exp++; }
```

```
  DBG(DBGFLOAT, "\tdigits=%d␣mantissa=0x%" PRIx64 ",␣exp=%d\n",
      digits, mantissa, exp);
  mantissa = mantissa & ~((uint64_t) 1 ≪ DBL_M_BITS);
  DBG(DBGFLOAT, "\tdigits=%d␣mantissa=0x%" PRIx64 ",␣exp=%d\n",
      digits, mantissa, exp);
}
```

Used in 62.

In the printed representation, the exponent is an exponent with base 16. For example, an exponent of 2 shifts the hexadecimal mantissa two hexadecimal digits to the left, which corresponds to a multiplication by 16^2.

⟨ read the optional exponent 66 ⟩ ≡ (66)

```
if (*x ≡ 'x')
{ int s;
  x++;                                          /* skip the "x" */
  if (*x ≡ '-') { s = −1; x++; }
  else if (*x ≡ '+') { s = +1; x++; }
  else s = +1;
  DBG(DBGFLOAT, "\texpsign=%d\n", s);
  DBG(DBGFLOAT, "\texp=%d\n", exp);
  while (*x ≠ 0) {
    if (*x < 'A') exp = exp + 4 * s * (*x − '0');
    else exp = exp + 4 * s * (*x − 'A' + 10);
    x++;
    DBG(DBGFLOAT, "\texp=%d\n", exp);
  }
}
RNG("Floating␣point␣exponent",
    exp, −DBL_EXCESS, DBL_EXCESS);
```

Used in 62.

To assemble the binary representation, we use a **union** of a **float64_t** and **uint64_t**.

⟨ return the binary representation 67 ⟩ ≡ (67)

```
{ union { float64_t d; uint64_t bits; } u;
  if (sign < 0) sign = 1; else sign = 0;        /* the sign bit */
  exp = exp + DBL_EXCESS;                      /* the exponent bits */
  u.bits = ((uint64_t) sign ≪ 63)
   | ((uint64_t) exp ≪ DBL_M_BITS) | mantissa;
  DBG(DBGFLOAT, "␣return␣%f\n", u.d);
  return u.d;
}
```

Used in 62.

The inverse function is *hwrite_float64*. It strives to print floating point numbers as readable as possible. So numbers without fractional part are written as integers.

Numbers that can be represented exactly in decimal notation are represented in decimal notation. All other values are written as hexadecimal floating point numbers. We avoid an exponent if it can be avoided by using up to `MAX_HEX_DIGITS`. For the use with extended dimensions, floating point numbers should be printed as a suffix: without a leading space and with a mandatory sign.

Writing the long format: $\Longrightarrow - - -$

⟨ write functions 21 ⟩ +≡ (68)

```
#define MAX_HEX_DIGITS 12
  void hwrite_float64(float64_t d, bool suffix)
  { uint64_t bits, mantissa;
    int exp, digits;
    if (¬suffix) hwritec('␣');
    else if (d ≥ 0) hwritec('+');
    if (floor(d) ≡ d) { hwritef("%d", (int) d); return; }
    if (floor(10000.0 * d) ≡ 10000.0 * d) { hwritef("%g", d); return; }
    DBG(DBGFLOAT, "Writing␣hexadecimal␣float␣%f\n", d);
    if (d < 0) { hwritec('-'); d = −d; }
    hwritef("0x");
    ⟨ extract mantissa and exponent 69 ⟩
    if (exp > MAX_HEX_DIGITS) ⟨ write large numbers 72 ⟩
    else if (exp ≥ 0) ⟨ write medium numbers 73 ⟩
    else ⟨ write small numbers 74 ⟩
  }
```

The extraction just reverses the creation of the binary representation.

⟨ extract mantissa and exponent 69 ⟩ ≡ (69)

```
  { union { float64_t d; uint64_t bits; } u;
    u.d = d; bits = u.bits;
  }
  mantissa = bits & (((uint64_t) 1 ≪ DBL_M_BITS) − 1);
  mantissa = mantissa + ((uint64_t) 1 ≪ DBL_M_BITS);
  exp = ((bits ≫ DBL_M_BITS) & ((1 ≪ DBL_E_BITS) − 1)) − DBL_EXCESS;
  digits = DBL_M_BITS + 1;
  DBG(DBGFLOAT, "\tdigits=%d␣mantissa=0x%" PRIx64 "␣binary␣exp=%d\n",
      digits, mantissa, exp);
```

Used in 68.

After we have obtained the binary exponent, we round it down, and convert it to a hexadecimal exponent.

⟨ extract mantissa and exponent 69 ⟩ +≡ (70)

```
  { int r;
    if (exp ≥ 0) { r = exp % 4;
      if (r > 0) { mantissa = mantissa ≪ r; exp = exp − r; digits = digits + r; }
    }
```

```
    else { r = (-exp) % 4;
      if (r > 0) { mantissa = mantissa >> r; exp = exp + r; digits = digits - r; }
    }
  }
  exp = exp/4;
  DBG(DBGFLOAT, "\tdigits=%d mantissa=0x%" PRIx64 " hex exp=%d\n",
      digits, mantissa, exp);
```

In preparation for writing, we shift the mantissa to the left so that the leftmost hexadecimal digit of it will occupy the 4 leftmost bits of the variable *bits* .

⟨ extract mantissa and exponent 69 ⟩ +≡ (71)

```
  mantissa = mantissa << (64 - DBL_M_BITS - 4);    /* move leading digit to
      leftmost nibble */
```

If the exponent is larger than MAX_HEX_DIGITS, we need to use an exponent even if the mantissa uses only a few digits. When we use an exponent, we always write exactly one digit preceding the radix point.

⟨ write large numbers 72 ⟩ ≡ (72)

```
  { DBG(DBGFLOAT, "writing large number\n");
    hwritef("%X.", (uint8_t)(mantissa >> 60));
    mantissa = mantissa << 4;
    do { hwritef("%X", (uint8_t)(mantissa >> DBL_M_BITS) & #F);
      mantissa = mantissa << 4;
    } while (mantissa ≠ 0);
    hwritef("x+%X", exp);
  }
```

Used in 68.

If the exponent is small and non negative, we can write the number without an exponent by writing the radix point at the appropriate place.

⟨ write medium numbers 73 ⟩ ≡ (73)

```
  { DBG(DBGFLOAT, "writing medium number\n");
    do { hwritef("%X", (uint8_t)(mantissa >> 60));
      mantissa = mantissa << 4;
      if (exp-- ≡ 0) hwritec('.');
    } while (mantissa ≠ 0 ∨ exp ≥ -1);
  }
```

Used in 68.

Last non least, we write numbers that would require additional zeros after the radix point with an exponent, because it keeps the mantissa shorter.

⟨ write small numbers 74 ⟩ ≡ (74)

```
  { DBG(DBGFLOAT, "writing small number\n");
    hwritef("%X.", (uint8_t)(mantissa >> 60));
    mantissa = mantissa << 4;
    do { hwritef("%X", (uint8_t)(mantissa >> 60));
      mantissa = mantissa << 4;
```

```
    } while (mantissa ≠ 0);
    hwritef("x-%X", −exp);
  }
```

Used in 68.

Compared to the complications of long format floating point numbers, the short format is very simple because we just use the binary representation. Since 32 bit floating point numbers offer sufficient precision we use only the **float32_t** type. It is however not possible to just write `HPUT32`(d) for a **float32_t** variable d or `HPUT32`((**uint32_t**) d) because in the C language this would imply rounding the floating point number to the nearest integer. But we have seen how to convert floating point values to bit pattern before.

```
⟨ put functions 14 ⟩ +≡                                                    (75)
  void hput_float32(float32_t d)
  { union { float32_t d; uint32_t bits; } u;
    u.d = d; HPUT32(u.bits);
  }
```

```
⟨ shared get functions 53 ⟩ +≡                                             (76)
  float32_t hget_float32(void)
  { union { float32_t d; uint32_t bits; } u;
    HGET32(u.bits);
    return u.d;
  }
```

2.5 Fixed Point Numbers

TEX internally represents most real numbers as fixed point numbers or "scaled integers". The type **Scaled** is defined as a signed 32 bit integer, but we consider it as a fixed point number with the binary radix point just in the middle with sixteen bits before and sixteen bits after it. To convert an integer into a scaled number, we multiply it by `ONE`; to convert a floating point number into a scaled number, we multiply it by `ONE` and `ROUND` the result to the nearest integer; to convert a scaled number to a floating point number we divide it by (**float64_t**) `ONE`.

```
⟨ hint basic types 6 ⟩ +≡                                                  (77)
  typedef int32_t Scaled;
#define ONE ((Scaled)(1 ≪ 16))
```

```
⟨ hint macros 13 ⟩ +≡                                                      (78)
#define ROUND(X) ((int)((X) ≥ 0.0 ? floor((X) + 0.5) : ceil((X) − 0.5)))
```

Writing the long format: ⟹ − − −

```
⟨ write functions 21 ⟩ +≡                                                  (79)
  void hwrite_scaled(Scaled x)
  { hwrite_float64(x/(float64_t) ONE, false);
  }
```

2.6 Dimensions

In the long format, the dimensions of characters, boxes, and other things can be given in three units: `pt`, `in`, and `mm`.

Reading the long format: --- ⟹

⟨ symbols 2 ⟩ +≡ (80)

```
%token DIMEN "dimen"
%token PT "pt"
%token MM "mm"
%token INCH "in"
%type < d > dimension
```

⟨ scanning rules 3 ⟩ +≡ (81)

```
dimen        return DIMEN;
pt           return PT;
mm           return MM;
in           return INCH;
```

The unit `pt` is a printers point. The unit "`in`" stands for inches and we have 1`in` = 72.27`pt`. The unit "`mm`" stands for millimeter and we have 1`in` = 25.4`mm`.

The definition of a printers point given above follows the definition used in TEX which is slightly larger than the official definition of a printer's point which was defined to equal exactly 0.013837`in` by the American Typefounders Association in 1886[8].

We follow the tradition of TEX and store dimensions as "scaled points" that is a dimension of d points is stored as $d \cdot 2^{16}$ rounded to the nearest integer. The maximum absolute value of a dimension is $(2^{30} - 1)$ scaled points.

⟨ hint basic types 6 ⟩ +≡ (82)

```
  typedef Scaled Dimen;
#define MAX_DIMEN ((Dimen)(#3FFFFFFF))
```

⟨ parsing rules 5 ⟩ +≡ (83)

```
  dimension: number PT
        { $$ = ROUND($1 * ONE);
        RNG("Dimension", $$, -MAX_DIMEN, MAX_DIMEN); }
    | number INCH
        { $$ = ROUND($1 * ONE * 72.27);
        RNG("Dimension", $$, -MAX_DIMEN, MAX_DIMEN); }
    | number MM
        { $$ = ROUND($1 * ONE * (72.27/25.4));
        RNG("Dimension", $$, -MAX_DIMEN, MAX_DIMEN); };
```

When `stretch` is writing dimensions in the long format, for simplicity it always uses the unit "`pt`".

Writing the long format: $\Longrightarrow$ − − −

```
⟨ write functions 21 ⟩ +≡                                              (84)
  void hwrite_dimension(Dimen x)
  { hwrite_scaled(x);
    hwritef("pt");
  }
```

In the short format, dimensions are stored as 32 bit scaled point values without conversion.

Reading the short format: $\cdots \Longrightarrow$

```
⟨ get functions 18 ⟩ +≡                                                (85)
  void hget_dimen(Tag a)
  {
    if (INFO(a) ≡ b000) { uint8_t r;
      r = HGET8;
      REF(dimen_kind, r);
      hwrite_ref(r);
    }
    else { uint32_t d;
      HGET32(d);
      hwrite_dimension(d);
    }
  }
```

Writing the short format: $\Longrightarrow \cdots$

```
⟨ put functions 14 ⟩ +≡                                                (86)
  Tag hput_dimen(Dimen d)
  { HPUT32(d);
    return TAG(dimen_kind, b001);
  }
```

2.7 Extended Dimensions

The dimension that is probably used most frequently in a TEX file is `hsize`: the horizontal size of a line of text. Common are also assignments like `\hsize=0.5\hsize \advance\hsize by -10pt`, for example to get two columns with lines almost half as wide as usual, leaving a small gap between left and right column. Similar considerations apply to `vsize`.

Because we aim at a reflowable format for TEX output, we have to postpone such computations until the values of `hsize` and `vsize` are known in the viewer. Until then, we do symbolic computations on linear functions of `hsize` and `vsize`. We call such a linear function $w + h \cdot \texttt{hsize} + v \cdot \texttt{vsize}$ an extended dimension and represent it by the three numbers w, h, and v.

⟨ hint basic types 6 ⟩ +≡ (87)

```
typedef struct { Dimen w; float32_t h, v; } Xdimen;
```

Since very often a component of an extended dimension is zero, we store in the short format only the nonzero components and use the info bits to mark them: *b100* implies $w \neq 0$, *b010* implies $h \neq 0$, and *b001* implies $v \neq 0$.

Reading the long format: − − − ⟹

⟨ symbols 2 ⟩ +≡ (88)

```
%token XDIMEN "xdimen"
%token H "h"
%token V "v"
%type <xd> xdimen
```

⟨ scanning rules 3 ⟩ +≡ (89)

```
xdimen          return XDIMEN;
h               return H;
v               return V;
```

⟨ parsing rules 5 ⟩ +≡ (90)

```
xdimen: dimension number H number V { $$.w = $1; $$.h = $2; $$.v = $4; }
    | dimension number H { $$.w = $1; $$.h = $2; $$.v = 0.0; }
    | dimension number V { $$.w = $1; $$.h = 0.0; $$.v = $2; }
    | dimension { $$.w = $1; $$.h = 0.0; $$.v = 0.0; };

xdimen_node: start XDIMEN xdimen END {
        hput_tags($1, hput_xdimen(&($3))); };
```

Writing the long format: ⟹ − − −

⟨ write functions 21 ⟩ +≡ (91)

```
void hwrite_xdimen(Xdimen *x)
{ hwrite_dimension(x→w);
  if (x→h ≠ 0.0) { hwrite_float64(x→h, true); hwritec('h'); }
  if (x→v ≠ 0.0) { hwrite_float64(x→v, true); hwritec('v'); }
}

void hwrite_xdimen_node(Xdimen *x)
{ hwrite_start();
  hwritef("xdimen");
  hwrite_xdimen(x);
  hwrite_end();
}
```

Reading the short format: $\cdots \Longrightarrow$

⟨ get macros 19 ⟩ +≡ (92)

```
#define HGET_XDIMEN(I, X)
  if ((I) & b100) HGET32((X).w); else (X).w = 0;
  if ((I) & b010) (X).h = hget_float32(); else (X).h = 0.0;
  if ((I) & b001) (X).v = hget_float32(); else (X).v = 0.0;
```

⟨ get functions 18 ⟩ +≡ (93)

```
void hget_xdimen(Tag a, Xdimen *x)
{
  switch (a) {
  case TAG(xdimen_kind, b001): HGET_XDIMEN(b001, *x); break;
  case TAG(xdimen_kind, b010): HGET_XDIMEN(b010, *x); break;
  case TAG(xdimen_kind, b011): HGET_XDIMEN(b011, *x); break;
  case TAG(xdimen_kind, b100): HGET_XDIMEN(b100, *x); break;
  case TAG(xdimen_kind, b101): HGET_XDIMEN(b101, *x); break;
  case TAG(xdimen_kind, b110): HGET_XDIMEN(b110, *x); break;
  case TAG(xdimen_kind, b111): HGET_XDIMEN(b111, *x); break;
  default: QUIT("Extent expected got [%s,%d]", NAME(a), INFO(a));
  }
}
```

Note that the info value *b000*, usually indicating a reference, is not supported for extended dimensions. Most nodes that need an extended dimension offer the opportunity to give a reference directly without the start and end byte. An exception is the glue node, but glue nodes that need an extended width are rare.

⟨ get functions 18 ⟩ +≡ (94)

```
void hget_xdimen_node(Xdimen *x)
{ ⟨ read the start byte a 16 ⟩
  if (KIND(a) ≡ xdimen_kind) hget_xdimen(a, x);
  else QUIT("Extent expected at 0x%x got %s", node_pos, NAME(a));
  ⟨ read and check the end byte z 17 ⟩
}
```

Writing the short format: $\Longrightarrow \cdots$

⟨ put functions 14 ⟩ +≡ (95)

```
Tag hput_xdimen(Xdimen *x)
{ Info info = b000;
  if (x→w ≡ 0 ∧ x→h ≡ 0.0 ∧ x→v ≡ 0.0) { HPUT32(0); info |= b100; }
  else {
    if (x→w ≠ 0) { HPUT32(x→w); info |= b100; }
    if (x→h ≠ 0.0) { hput_float32(x→h); info |= b010; }
    if (x→v ≠ 0.0) { hput_float32(x→v); info |= b001; }
  }
  return TAG(xdimen_kind, info);
```

```
}
void hput_xdimen_node(Xdimen *x)
{ uint32_t p = hpos++ - hstart;
  hput_tags(p, hput_xdimen(x));
}
```

2.8 Stretch and Shrink

In section 3.5, we will consider glue which is something that can stretch and shrink. The stretchability and shrinkability of the glue can be given in "pt" like a dimension, but there are three more units: fil, fill, and filll. A glue with a stretchability of 1 fil will stretch infinitely more than a glue with a stretchability of 1 pt. So if you stretch both glues together, the first glue will do all the stretching and the latter will not stretch at all. The "fil" glue has simply a higher order of infinity. You might guess that "fill" glue and "filll" glue have even higher orders of infinite stretchability. The order of infinity is 0 for pt, 1 for fil, 2 for fill, and 3 for filll.

The internal representation of a stretch is a variable of type **Stretch**. It stores the floating point value and the order of infinity separate as a **float64_t** and a **uint8_t**.

The short format tries to be space efficient and because it is not necessary to give the stretchability with a precision exceeding about six decimal digits, we use a single 32 bit floating point value. To write a **float32_t** value and an order value as one 32 bit value, we round the two lowest bit of the **float32_t** variable to zero using "round to even" and store the order of infinity in these bits. We define a union type **Stch** to simplify conversion.

⟨ hint basic types 6 ⟩ +≡ (96)

```
typedef enum { normal_o = 0, fil_o = 1, fill_o = 2, filll_o = 3 } Order;
typedef struct { float64_t f; Order o; } Stretch;
typedef union { float32_t f; uint32_t u; } Stch;
```

Writing the short format: ⟹ ···

⟨ put functions 14 ⟩ +≡ (97)

```
void hput_stretch(Stretch *s)
{ uint32_t mantissa, lowbits, sign, exponent;
  Stch st;
  st.f = s->f;
  DBG(DBGFLOAT, "joining %f->%f(0x%X),%d:", s->f, st.f, st.u, s->o);
  mantissa = st.u & (((uint32_t) 1 << FLT_M_BITS) - 1);
  lowbits = mantissa & #7;                                  /* lowest 3 bits */
  exponent = (st.u >> FLT_M_BITS) & (((uint32_t) 1 << FLT_E_BITS) - 1);
  sign = st.u & ((uint32_t) 1 << (FLT_E_BITS + FLT_M_BITS));
  DBG(DBGFLOAT, "s=%d e=0x%x m=0x%x", sign, exponent, mantissa);
  switch (lowbits)                                          /* round to even */
  { case 0: break;                                          /* no change */
```

case 1: $mantissa = mantissa - 1$; **break**; /* round down */
case 2: $mantissa = mantissa - 2$; **break**; /* round down to even */
case 3: $mantissa = mantissa + 1$; **break**; /* round up */
case 4: **break**; /* no change */
case 5: $mantissa = mantissa - 1$; **break**; /* round down */
case 6: $mantissa = mantissa + 1$; /* round up to even, fall through */
case 7: $mantissa = mantissa + 1$; /* round up to even */
  if ($mantissa \geq$ ((**uint32_t**) 1 $\ll$ FLT_M_BITS))
  { $exponent$++; /* adjust exponent */
    RNG("Float32␣exponent", $exponent$, 1, 2 ∗ FLT_EXCESS);
    $mantissa = mantissa \gg 1$;
  }
  break;
}
DBG(DBGFLOAT, "␣round␣s=%d␣e=0x%x␣m=0x%x", $sign$, $exponent$, $mantissa$);
$st.u = sign \mid (exponent \ll$ FLT_M_BITS$) \mid mantissa \mid s \rightarrow o$;
DBG(DBGFLOAT, "float␣%f␣hex␣0x%x\n", $st.f$, $st.u$);
HPUT32($st.u$);
}

Reading the short format: $\cdots \Longrightarrow$

⟨ get macros 19 ⟩ +≡ (98)
#define HGET_STRETCH(S)
 { **Stch** st; HGET32($st.u$); $S.o = st.u$ & 3;
  $st.u$ &= ~3;
  $S.f = st.f$; }

Reading the long format: $---\Longrightarrow$

⟨ symbols 2 ⟩ +≡ (99)
%**token** FIL "fil"
%**token** FILL "fill"
%**token** FILLL "filll"
%**type** < st > *stretch*
%**type** < o > *order*

⟨ scanning rules 3 ⟩ +≡ (100)
fil **return** FIL;
fill **return** FILL;
filll **return** FILLL;

⟨ parsing rules 5 ⟩ +≡ (101)
 order: PT { $$ = *normal_o*; }
  | FIL { $$ = *fil_o*; } | FILL { $$ = *fill_o*; } | FILLL { $$ = *filll_o*; };
 stretch: *number order* { $$.f = $1; $$.o = $2; };

Writing the long format: $\Longrightarrow - - -$

⟨ write functions 21 ⟩ +≡ (102)

```
void hwrite_order(Order o)
{
  switch (o) {
  case normal_o: hwritef("pt"); break;
  case fil_o: hwritef("fil"); break;
  case fill_o: hwritef("fill"); break;
  case filll_o: hwritef("filll"); break;
  default: QUIT("Illegal␣order␣%d", o); break;
  }
}
void hwrite_stretch(Stretch *s)
{ hwrite_float64(s→f, false);
  hwrite_order(s→o);
}
```

3 Simple Nodes

3.1 Penalties

Penalties are very simple nodes. They specify the cost of breaking a line or page at the present position. For the internal representation we use an **int32_t**. The full range of integers is, however, not used. Instead penalties must be between -20000 and +20000. (TEX specifies a range of -10000 to +10000, but plain TEX uses the value -20000 when it defines the supereject control sequence.) The more general node is called an integer node; it shares the same kind-value *int_kind* = *penalty_kind* but allows the full range of values. The info value of a penalty node is 1 or 2 and indicates the number of bytes used to store the integer. The info value 3 can be used for general integers (see section 10.2) that need four byte of storage.

Reading the long format: − − − ⟹

⟨ symbols 2 ⟩ +≡ (103)

%**token** PENALTY "penalty"
%**token** INTEGER "int"
%**type** $< i >$ *penalty*

⟨ scanning rules 3 ⟩ +≡ (104)

```
penalty        return PENALTY;
int            return INTEGER;
```

⟨ parsing rules 5 ⟩ +≡ (105)

penalty: *integer* { RNG("Penalty", \$1, −20000, +20000); \$\$ = \$1; };

content_node: *start* PENALTY *penalty* END { *hput_tags*(\$1, *hput_int*(\$3)); };

Reading the short format: $\cdots \Longrightarrow$

⟨ cases to get content 20 ⟩ +≡ (106)

```
case TAG(penalty_kind, 1): { int32_t p; HGET_PENALTY(1, p); } break;
case TAG(penalty_kind, 2): { int32_t p; HGET_PENALTY(2, p); } break;
case TAG(penalty_kind, 3): { int32_t p; HGET_PENALTY(3, p); } break;
```

⟨ get macros 19 ⟩ +≡ (107)

```
#define HGET_PENALTY(I, P)
  if (I ≡ 1) { int8_t n = HGET8; P = n; }
  else if (I ≡ 2) { int16_t n; HGET16(n); RNG("Penalty", n, −20000, +20000);
    P = n; }
  else if (I ≡ 3) { int32_t n; HGET32(n); RNG("Penalty", n, −20000, +20000);
    P = n; }
  hwrite_signed(P);
```

Writing the short format: $\Longrightarrow \cdots$

⟨ put functions 14 ⟩ +≡ (108)

```
  Tag hput_int(int32_t n)
  { Info info;
    if (n ≥ 0)
    { if (n < #80) { HPUT8(n); info = 1; }
      else if (n < #8000) { HPUT16(n); info = 2; }
      else { HPUT32(n); info = 3; }
    }
    else
    { if (n ≥ −#80) { HPUT8(n); info = 1; }
      else if (n ≥ −#8000) { HPUT16(n); info = 2; }
      else { HPUT32(n); info = 3; }
    }
    return TAG(int_kind, info);
  }
```

3.2 Languages

To render a HINT file on screen, information about the language is not necessary. Knowing the language is, however, very important for language translation and text to speech conversion which makes texts accessible to the visually-impaired. For this reason, HINT offers the opportunity to add this information and encourages authors to supply this information.

Language information by itself is not sufficient to decode text. It must be supplemented by information about the character encoding (see section 10.4).

To represent language information, the world wide web has set universally accepted standards. The Internet Engineering Task Force IETF has defined tags for identifying languages[15]: short strings like “en” for English or “de” for Deutsch, and longer ones like “sl-IT-nedis”, for the specific variant of the Nadiza dialect of

Slovenian that is spoken in Italy. We assume that any HINT file will contain only a small number of different languages and all language nodes can be encoded using a reference to a predefined node from the definition section (see section 10.5). In the definition section, a language node will just contain the language tag as given in [5] (see section 10.2).

Reading the long format: $- - - \Longrightarrow$

⟨ symbols 2 ⟩ +≡ (109)

```
%token LANGUAGE "language"
```

⟨ scanning rules 3 ⟩ +≡ (110)

```
language        return LANGUAGE;
```

When encoding language nodes in the short format, we use the info value *b000* for language nodes in the definition section and for language nodes in the content section that contain just a one-byte reference (see section 10.5). We use the info value 1 to 7 as a shorthand for the references `*0` and `*6` to the predefined language nodes.

Reading the short format: $\cdots \Longrightarrow$
Writing the long format: $\Longrightarrow - - -$

⟨ cases to get content 20 ⟩ +≡ (111)

```
case TAG(language_kind, 1): REF(language_kind, 0); hwrite_ref(0); break;
case TAG(language_kind, 2): REF(language_kind, 1); hwrite_ref(1); break;
case TAG(language_kind, 3): REF(language_kind, 2); hwrite_ref(2); break;
case TAG(language_kind, 4): REF(language_kind, 3); hwrite_ref(3); break;
case TAG(language_kind, 5): REF(language_kind, 4); hwrite_ref(4); break;
case TAG(language_kind, 6): REF(language_kind, 5); hwrite_ref(5); break;
case TAG(language_kind, 7): REF(language_kind, 6); hwrite_ref(6); break;
```

Writing the short format: $\Longrightarrow \cdots$

⟨ put functions 14 ⟩ +≡ (112)

```
Tag hput_language(uint8_t n)
{
  if (n < 7) return TAG(language_kind, n + 1);
  HPUT8(n);
  return TAG(language_kind, 0);
}
```

3.3 Rules

Rules are simply black rectangles having a height, a depth, and a width. All of these dimensions can also be negative but a rule will not be visible unless its width is positive and its height plus depth is positive.

As a specialty, rules can have "running dimensions". If any of the three dimensions is a running dimension, its actual value will be determined by running the rule up to the boundary of the innermost enclosing box. The width is never running in an horizontal list; the height and depth are never running in a vertical list. In the long format, we use a vertical bar "`|`" or a horizontal bar "_" (underscore character) to indicate a running dimension. Of course the vertical bar is meant to indicate a running height or depth while the horizontal bar stands for a running width. The parser, however, makes no distinction between the two and you can use either of them. In the short format, we follow TEX and implement a running dimension by using the special value $-2^{30} = {}^{\#}$`C0000000`.

⟨ hint macros 13 ⟩ +≡ (113)

```
#define RUNNING_DIMEN #C0000000
```

It could have been possible to allow extended dimensions in a rule node, but in most circumstances, the mechanism of running dimensions is sufficient and simpler to use. If a rule is needed that requires an extended dimension as its length, it is always possible to put it inside a suitable box and use a running dimension.

To make the short format encoding more compact, the first info bit *b100* will be zero to indicate a running height, bit *b010* will be zero to indicate a running depth, and bit *b001* will be zero to indicate a running width.

Because leaders (see section 5.3) may contain a rule node, we also provide functions to read and write a complete rule node. While parsing the symbol "*rule*" will just initialize a variable of type **Rule** (the writing is done with a separate routine), parsing a *rule_node* will always include writing it.

⟨ hint types 1 ⟩ +≡ (114)

```
typedef struct { Dimen h, d, w; } Rule;
```

Reading the long format: − − − ⟹

⟨ symbols 2 ⟩ +≡ (115)

```
%token RULE "rule"
%token RUNNING "|"
%type <d> rule_dimension
%type <r> rule
```

⟨ scanning rules 3 ⟩ +≡ (116)

```
rule      return RULE;
"|"       return RUNNING;
"_"       return RUNNING;
```

⟨ parsing rules 5 ⟩ +≡ (117)

```
rule_dimension: dimension | RUNNING { $$ = RUNNING_DIMEN; };
```

```
rule:  rule_dimension rule_dimension rule_dimension
    { $$.h = $1; $$.d = $2; $$.w = $3;
      if ($3 ≡ RUNNING_DIMEN ∧ ($1 ≡ RUNNING_DIMEN ∨ $2 ≡
            RUNNING_DIMEN))
        QUIT("Incompatible␣running␣dimensions␣0x%x␣0x%x␣0x%x",
            $1, $2, $3);
      };

rule_node:  start RULE rule END { hput_tags($1, hput_rule(&($3))); };

content_node:  rule_node;
```

Writing the long format: ⟹ − − −

⟨ write functions 21 ⟩ +≡ (118)

```
static void hwrite_rule_dimension(Dimen d, char c)
{ if (d ≡ RUNNING_DIMEN) hwritef("␣%c", c);
  else hwrite_dimension(d);
}

void hwrite_rule(Rule *r)
{ hwrite_rule_dimension(r→h, '|');
  hwrite_rule_dimension(r→d, '|');
  hwrite_rule_dimension(r→w, '_');
}
```

Reading the short format: ··· ⟹

⟨ cases to get content 20 ⟩ +≡ (119)

```
case TAG(rule_kind, b011):
  { Rule r; HGET_RULE(b011, r); hwrite_rule(&(r)); } break;
case TAG(rule_kind, b101):
  { Rule r; HGET_RULE(b101, r); hwrite_rule(&(r)); } break;
case TAG(rule_kind, b001):
  { Rule r; HGET_RULE(b001, r); hwrite_rule(&(r)); } break;
case TAG(rule_kind, b110):
  { Rule r; HGET_RULE(b110, r); hwrite_rule(&(r)); } break;
case TAG(rule_kind, b111):
  { Rule r; HGET_RULE(b111, r); hwrite_rule(&(r)); } break;
```

⟨ get macros 19 ⟩ +≡ (120)

```
#define HGET_RULE(I, R)
  if ((I) & b100) HGET32((R).h); else (R).h = RUNNING_DIMEN;
  if ((I) & b010) HGET32((R).d); else (R).d = RUNNING_DIMEN;
  if ((I) & b001) HGET32((R).w); else (R).w = RUNNING_DIMEN;
```

```
⟨ get functions 18 ⟩ +≡                                                   (121)
  void hget_rule_node(void)
  { ⟨ read the start byte a 16 ⟩
    if (KIND(a) ≡ rule_kind)
    { Rule r; HGET_RULE(INFO(a), r);
      hwrite_start(); hwritef("rule"); hwrite_rule(&r); hwrite_end();
    }
    else QUIT("Rule␣expected␣at␣0x%x␣got␣%s", node_pos, NAME(a));
    ⟨ read and check the end byte z 17 ⟩
  }
```

Writing the short format: ⟹ · · ·

```
⟨ put functions 14 ⟩ +≡                                                   (122)
  Tag hput_rule(Rule *r)
  { Info info = b000;
    if (r→h ≠ RUNNING_DIMEN) { HPUT32(r→h); info |= b100; }
    if (r→d ≠ RUNNING_DIMEN) { HPUT32(r→d); info |= b010; }
    if (r→w ≠ RUNNING_DIMEN) { HPUT32(r→w); info |= b001; }
    return TAG(rule_kind, info);
  }
```

3.4 Kerns

A kern is a bit of white space with a certain length. If the kern is part of a horizontal list, the length is measured in the horizontal direction, if it is part of a vertical list, it is measured in the vertical direction. The length of a kern is mostly given as a dimension but provisions are made to use extended dimensions as well.

The typical use of a kern is its insertion between two characters to make the natural distance between them a bit wider or smaller. In the latter case, the kern has a negative length. The typographic optimization just described is called "kerning" and has given the kern node its name. Kerns inserted from font information or math mode calculations are normal kerns, while kerns inserted from TEX's `\kern` or `\/` commands are explicit kerns. Kern nodes do not disappear at a line break unless they are explicit.

In the long format, explicit kerns are marked with an "!" sign and in the short format with the *b100* info bit. The two low order info bits are: 0 for a reference to a dimension, 1 for a reference to an extended dimension, 2 for an immediate dimension, and 3 for an immediate extended dimension node. To distinguish in the long format between a reference to a dimension and a reference to an extended dimension, the latter is prefixed with the keyword "`xdimen`" (see section 10.5).

```
⟨ hint types 1 ⟩ +≡                                                       (123)
  typedef struct { bool x; Xdimen d; } Kern;
```

Reading the long format: $---\Longrightarrow$

⟨ symbols 2 ⟩ +≡ (124)
```
%token KERN "kern"
%token EXPLICIT "!"
%type <b> explicit
%type <kt> kern
```

⟨ scanning rules 3 ⟩ +≡ (125)
```
kern        return KERN;
!           return EXPLICIT;
```

⟨ parsing rules 5 ⟩ +≡ (126)
```
explicit: { $$ = false; } | EXPLICIT { $$ = true; };
kern: explicit xdimen { $$.x = $1; $$.d = $2; };
content_node: start KERN kern END { hput_tags($1, hput_kern(&($3))); }
```

Writing the long format: $\Longrightarrow---$

⟨ write functions 21 ⟩ +≡ (127)
```
void hwrite_explicit(bool x)
{ if (x) hwritef(" !"); }
void hwrite_kern(Kern *k)
{ hwrite_explicit(k→x);
  if (k→d.h ≡ 0.0 ∧ k→d.v ≡ 0.0 ∧ k→d.w ≡ 0) hwrite_ref(zero_dimen_no);
  else hwrite_xdimen(&(k→d));
}
```

Reading the short format: $\cdots\Longrightarrow$

⟨ cases to get content 20 ⟩ +≡ (128)
```
case TAG(kern_kind, b010): { Kern k; HGET_KERN(b010, k); } break;
case TAG(kern_kind, b011): { Kern k; HGET_KERN(b011, k); } break;
case TAG(kern_kind, b110): { Kern k; HGET_KERN(b110, k); } break;
case TAG(kern_kind, b111): { Kern k; HGET_KERN(b111, k); } break;
```

⟨ get macros 19 ⟩ +≡ (129)
```
#define HGET_KERN(I, K) K.x = (I) & b100;
  if (((I) & b011) ≡ 2) { HGET32(K.d.w); K.d.h = K.d.v = 0.0; }
  else if (((I) & b011) ≡ 3) hget_xdimen_node(&(K.d));
  hwrite_kern(&k);
```

Writing the short format: $\Longrightarrow \cdots$

$\langle$ put functions 14 $\rangle$ +≡ (130)

$$
\begin{array}{l}
\textbf{Tag}\ \mathit{hput\_kern}(\textbf{Kern}\ {*k})\\
\{\ \textbf{Info}\ \mathit{info};\\
\quad \textbf{if}\ (k\to x)\ \mathit{info} = \mathit{b100};\ \textbf{else}\ \mathit{info} = \mathit{b000};\\
\quad \textbf{if}\ (k\to d.h \equiv 0.0 \wedge k\to d.v \equiv 0.0)\ \{\\
\qquad \textbf{if}\ (k\to d.w \equiv 0)\ \texttt{HPUT8}(\mathit{zero\_dimen\_no});\\
\qquad \textbf{else}\ \{\ \texttt{HPUT32}(k\to d.w);\ \mathit{info} = \mathit{info} \mid 2;\ \}\\
\quad \}\\
\quad \textbf{else}\ \{\ \mathit{hput\_xdimen\_node}(\&(k\to d));\ \mathit{info} = \mathit{info} \mid 3;\ \}\\
\quad \textbf{return}\ \texttt{TAG}(\mathit{kern\_kind}, \mathit{info});\\
\}
\end{array}
$$

3.5 Glue

We have seen in section 2.8 how to deal with stretchability and shrinkability and we will need this now. Glue has a natural width—which in general can be an extended dimension—and in addition it can stretch and shrink. It might have been possible to allow an extended dimension also for the stretchability or shrinkability of a glue, but this seems of little practical relevance and so simplicity won over generality. Even with that restriction, it is an understatement to regard glue nodes as "simple" nodes.

To use the info bits in the short format wisely, I collected some statistical data using the TEXbook as an example. It turns out that about 99% of all the 58937 glue nodes (not counting the interword glues used inside texts) could be covered with only 43 predefined glues. So this is by far the most common case; we reserve the info value *b000* to cover it and postpone the description of such glue nodes until we describe references in section 10.5.

We expect the remaining cases to contribute not too much to the file size, and hence, simplicity is a more important aspect than efficiency when allocating the remaining info values.

Looking at the glues in more detail, we find that the most common cases are those where either one, two, or all three glue components are zero. We use the two lowest bits to indicate the presence of a nonzero stretchability or shrinkability and reserve the info values *b001*, *b010*, and *b011* for those cases where the width of the glue is zero. The zero glue, where all components are zero, is defined as a fixed, predefined glue instead of reserving a special info value for it. The cost of one extra byte when encoding it seems not too high a price to pay. After reserving the info value *b111* for the most general case of a glue, we have only three more info values left: *b100*, *b101*, and *b110*. Keeping things simple implies using the two lowest info bits—as before—to indicate a nonzero stretchability or shrinkability. For the width, three choices remain: using a reference to a dimension, using a reference to an extended dimension, or using an immediate value. Since references to glues are already supported, an immediate width seems best for glues that are not frequently reused, avoiding the overhead of references.

Here is a summary of the info bits and the implied layout of glue nodes in the short format:

- *b000*: reference to a predefined glue
- *b001*: zero width and nonzero shrinkability
- *b010*: zero width and nonzero stretchability
- *b011*: zero width and nonzero stretchability and shrinkability
- *b100*: nonzero width
- *b101*: nonzero width and nonzero shrinkability
- *b110*: nonzero width and nonzero stretchability
- *b111*: extended dimension and nonzero stretchability and shrinkability

⟨ hint basic types 6 ⟩ +≡ (131)

typedef struct { **Xdimen** w; **Stretch** p, m; } **Glue**;

To test for a zero glue, we implement a macro:

⟨ hint macros 13 ⟩ +≡ (132)

#**define** `ZERO_GLUE`(G)
$((G).w.w \equiv 0 \land (G).w.h \equiv 0.0 \land (G).w.v \equiv 0.0 \land (G).p.f \equiv 0.0 \land (G).m.f \equiv 0.0)$

Because other nodes (leaders, baselines, and fonts) contain glue nodes as parameters, we provide functions to read and write a complete glue node in the same way as we did for rule nodes. Further, such an internal *glue_node* has the special property that in the short format a node for the zero glue might be omitted entirely.

Reading the long format: − − − ⟹

⟨ symbols 2 ⟩ +≡ (133)

%**token** GLUE `"glue"`
%**token** PLUS `"plus"`
%**token** MINUS `"minus"`
%**type** $< g >$ *glue*
%**type** $< b >$ *glue_node*
%**type** $< st >$ *plus minus*

⟨ scanning rules 3 ⟩ +≡ (134)

`glue` **return** GLUE;
`plus` **return** PLUS;
`minus` **return** MINUS;

⟨ parsing rules 5 ⟩ +≡ (135)

plus: { \$\$.f = 0.0; \$\$.o = 0; }
 | PLUS *stretch* { \$\$ = \$2; };
minus: { \$\$.f = 0.0; \$\$.o = 0; }
 | MINUS *stretch* { \$\$ = \$2; };
glue: *xdimen plus minus* { \$\$.w = \$1; \$\$.p = \$2; \$\$.m = \$3; };

```
content_node: start GLUE glue END {
        if (ZERO_GLUE($3)) { HPUT8(zero_skip_no);
          hput_tags($1, TAG(glue_kind, 0));
        }
        else hput_tags($1, hput_glue(&($3)));
        };
glue_node: start GLUE glue END
      { if (ZERO_GLUE($3)) { hpos--; $$ = false; }
        else { hput_tags($1, hput_glue(&($3))); $$ = true; } };
```

Writing the long format: ⟹ − − −

⟨ write functions 21 ⟩ +≡ (136)

```
void hwrite_plus(Stretch *p)
{ if (p→f ≠ 0.0) { hwritef("␣plus"); hwrite_stretch(p); }
}
void hwrite_minus(Stretch *m)
{ if (m→f ≠ 0.0) { hwritef("␣minus"); hwrite_stretch(m); }
}
void hwrite_glue(Glue *g)
{ hwrite_xdimen(&(g→w)); hwrite_plus(&g→p); hwrite_minus(&g→m);
}
void hwrite_ref_node(Kind k, uint8_t n);
void hwrite_glue_node(Glue *g)
{ if (ZERO_GLUE(*g)) hwrite_ref_node(glue_kind, zero_skip_no);
  else { hwrite_start(); hwritef("glue"); hwrite_glue(g); hwrite_end(); }
}
```

Reading the short format: ··· ⟹

⟨ cases to get content 20 ⟩ +≡ (137)

```
case TAG(glue_kind, b001):
  { Glue g; HGET_GLUE(b001, g); hwrite_glue(&g); } break;
case TAG(glue_kind, b010):
  { Glue g; HGET_GLUE(b010, g); hwrite_glue(&g); } break;
case TAG(glue_kind, b011):
  { Glue g; HGET_GLUE(b011, g); hwrite_glue(&g); } break;
case TAG(glue_kind, b100):
  { Glue g; HGET_GLUE(b100, g); hwrite_glue(&g); } break;
case TAG(glue_kind, b101):
  { Glue g; HGET_GLUE(b101, g); hwrite_glue(&g); } break;
case TAG(glue_kind, b110):
  { Glue g; HGET_GLUE(b110, g); hwrite_glue(&g); } break;
case TAG(glue_kind, b111):
  { Glue g; HGET_GLUE(b111, g); hwrite_glue(&g); } break;
```

⟨ get macros 19 ⟩ +≡ (138)

```
#define HGET_GLUE(I, G){
        if ((I) ≠ b111) {
          if ((I) & b100) HGET32((G).w.w); else (G).w.w = 0;
        }
        if ((I) & b010) HGET_STRETCH((G).p) else (G).p.f = 0.0, (G).p.o = 0;
        if ((I) & b001) HGET_STRETCH((G).m) else (G).m.f = 0.0, (G).m.o = 0;
        if ((I) ≡ b111) hget_xdimen_node(&((G).w));
        else (G).w.h = (G).w.v = 0.0; }
```

The *hget_glue_node* can cope with a glue node that is omitted and will supply a zero glue instead.

⟨ get functions 18 ⟩ +≡ (139)

```
  void hget_glue_node(void)
  { ⟨ read the start byte a 16 ⟩
    if (KIND(a) ≠ glue_kind) { hpos −−;
      hwrite_ref_node(glue_kind, zero_skip_no); return; }
    if (INFO(a) ≡ b000) { uint8_t n = HGET8; REF(glue_kind, n);
      hwrite_ref_node(glue_kind, n); }
    else { Glue g; HGET_GLUE(INFO(a), g); hwrite_glue_node(&g); }
    ⟨ read and check the end byte z 17 ⟩
  }
```

Writing the short format: ⟹ · · ·

⟨ put functions 14 ⟩ +≡ (140)

```
  Tag hput_glue(Glue *g)
  { Info info = b000;
    if (ZERO_GLUE(*g)) { HPUT8(zero_skip_no); info = b000;
    }
    else if ((g→w.w ≡ 0 ∧ g→w.h ≡ 0.0 ∧ g→w.v ≡ 0.0)) {
      if (g→p.f ≠ 0.0) { hput_stretch(&g→p); info |= b010; }
      if (g→m.f ≠ 0.0) { hput_stretch(&g→m); info |= b001; }
    }
    else if (g→w.h ≡ 0.0 ∧ g→w.v ≡ 0.0 ∧ (g→p.f ≡ 0.0 ∨ g→m.f ≡ 0.0)) {
      HPUT32(g→w.w); info = b100;
      if (g→p.f ≠ 0.0) { hput_stretch(&g→p); info |= b010; }
      if (g→m.f ≠ 0.0) { hput_stretch(&g→m); info |= b001; }
    }
    else
    { hput_stretch(&g→p); hput_stretch(&g→m);
      hput_xdimen_node(&(g→w));
      info = b111;
    }
    return TAG(glue_kind, info);
  }
```

4 Lists

When a node contains multiple other nodes, we package these nodes into a list node. It is important to note that list nodes never occur as individual nodes, they only occur as parts of other nodes. In total, we have three different types of lists: plain lists that use the kind-value *list_kind*, text lists that use the kind-value *list_kind* together with the info bit *b100*, and parameter lists that use the kind-value *param_kind*. A description of the first two types of lists follows here. Parameter lists are described in section 10.3.

Because lists are of variable size, it is not possible in the short format to tell from the kind and info bits of a tag byte the size of the list node. So advancing from the beginning of a list node to the next node after the list is not as simple as usual. To solve this problem, we store the size of the list immediately after the start byte and before the end byte. Alternatively we could require programs to traverse the entire list. The latter solution is more compact but inefficient for list with many nodes; our solution will cost some extra bytes, but the amount of extra bytes will only grow logarithmically with the size of the HINT file. It would be possible to allow both methods so that a HINT file could balance size and time trade-offs by making small lists—where the size can be determined easily by reading the entire list—without size information and making large lists with size information so that they can be skipped easily without reading them. But the added complexity seems too high a price to pay.

Now consider the problem of reading a content stream starting at an arbitrary position i in the middle of the stream. This situation occurs naturally when resynchronizing a content stream after an error has been detected, but implementing links poses a similar problem. We can inspect the byte at position i and see if it is a valid tag. If yes, we are faced with the problem of verifying that this is not a mere coincidence. So we determine the size s of the node. If the byte in question is a start byte, we should find a matching byte s bytes later in the stream; if it is an end byte, we should find the matching byte s bytes earlier in the stream; if we find no matching byte, this was neither a start nor an end byte. If we find exactly one matching byte, we can be quite confident (error probability 1/256 if assuming equal probability of all byte values) that we have found a tag, and we know whether it is the beginning or the end tag. If we find two matching byte, we have most likely the start or the end of a node, but we do not know which of the two. To find out which of the two possibilities is true or to reduce the probability of an error, we can check the start and end byte of the node immediately preceding a start byte or immediately following an end byte in a similar way. By testing two more byte, this additional check will reduce the error probability further to $1/2^{24}$ (under the same assumption as before). So checking more nodes is rarely necessary. This whole schema would, however, not work if we happen to find a tag byte that indicated either the begin or the end of a list without specifying the size of the list. Sure, we can verify the bytes before and after it to find out whether the byte following it is the begin of a node and the byte preceding it is the end of a node, but we still don't know if the byte itself starts a node list or ends a node list. Even reading along in either direction until finding a matching tag will not answer the question. The situation is better if we specify a size: we can read the suspected size after or before the tag and check if we find a matching tag and size at the position indicated.

In the short format, we use the two lower bits of the *info* value to indicate the number of byte used to store the list size: A list with *info* & #`3` = 1 uses 1 byte, with *info* & #`3` = 2 uses 2 byte, and with *info* & #`3` = 3 uses 4 byte. The *info* & #`3` value zero is reserved for references to predefined lists. An empty list is always represented using zero as the reference number. General predefined lists are currently implemented only for parameter lists.

Storing the list size immediately preceding the end tag creates a new problem: If we try to recover from an error, we might not know the size of the list and searching for the end of a list, we might be unable to tell the difference between the bytes that encode the list size and the start tag of a possible next node. If we parse the content backward, the problem is completely symmetric.

To solve the problem, we insert an additional byte immediately before the final size and after the initial size marking the size boundary. We choose the byte values #`FF`, #`FE`, and #`FD` which can not be confused with valid tag bytes and indicate that the size is stored using 1, 2, or 4 byte respectively. Under regular circumstances, these bytes are simply skipped. When searching for the list end (or start) these bytes would correspond to `TAG`(*penalty_kind*, i) with $7 \geq i \geq 5$ and can not be confused with valid penalty nodes which use only the info values 0, 1, and 2. An empty list always uses the info value 0 and the reference value 0.

We are a bit lazy when it comes to the internal representation of a list. Since we need the representation as a short format byte sequence anyway, it consists of the position p of the start of the byte sequence combined with an integer s giving the size of the byte sequence. If the list is empty, s is zero.

⟨ hint types 1 ⟩ +≡ (141)
typedef struct { **Tag** t; **uint32_t** p; **uint32_t** s; } **List**;

The major drawback of this choice of representation is that it ties together the reading of the long format and the writing of the short format; these are no longer independent. So starting with the present section, we have to take the short format representation of a node into account already when we parse the long format representation.

In the long format, we may start a list node with an estimate of the size needed to store the list in the short format. We do not want to require the exact size because this would make editing of long format HINT files almost impossible. Of course this makes it also impossible to derive the exact s value of the internal representation from the long format representation. Therefore we start by parsing the estimate of the list size and use it to reserve the necessary number of byte to store the size. Then we parse the *content_list*. As a side effect—and this is an important point—this will write the list content in short format into the output buffer. As mentioned above, whenever a node contains a list, we need to consider this side effect when we give the parsing rules. We will see examples for this in section 5.

The function *hput_list* will be called *after* the short format of the list is written to the output. Before we pass the internal representation of the list to the *hput_list* function, we update s and p. Further, we pass the position in the stream where the list size and its boundary mark is supposed to be. Before *hput_list* is called, space for the tag, the size, and the boundary mark is allocated based on the estimate. The function *hsize_bytes* computes the number of byte required to store the list size, and the function *hput_list_size* will later write the list size. If the estimate turns out to be wrong, the list data can be moved to make room for a larger or smaller size field.

If the long format does not specify a size estimate, a suitable default must be chosen. A statistical analysis shows that most plain lists need only a single byte to store the size; and even the total amount of data contained in these lists exceeds the amount of data stored in longer lists by a factor of about 3. Hence if we do not have an estimate, we reserve only a single byte to store the size of a list. The statistics looks different for lists stored as a text: The number of texts that require two byte for the size is slightly larger than the number of texts that need only one byte, and the total amount of data stored in these texts is larger by a factor of 2 to 7 than the total amount of data found in all other texts. Hence as a default, we reserve two byte to store the size for texts.

4.1 Plain Lists

Plain list nodes start and end with a tag of kind *list_kind*. Not uncommon are empty lists; these can be stored using $info = 0$ and a reference to the predefined empty list.

Writing the long format uses the fact that the function *hget_content_node*, as implemented in the `stretch` program, will output the node in the long format.

Reading the long format: $- - - \Longrightarrow$

$\langle$ symbols 2 $\rangle$ +≡ (142)

%**type** $<l>$ *list*

%**type** $<u>$ *position content_list*

$\langle$ parsing rules 5 $\rangle$ +≡ (143)

position: { `$$` $=$ *hpos* $-$ *hstart*; };

content_list: *position* | *content_list content_node*;

estimate: { *hpos* += 2; } | UNSIGNED { *hpos* += *hsize_bytes*(`$1`) + 1; };

list: *start estimate content_list* END

{ `$$`.*t* = `TAG`(*list_kind*, *b010*); `$$`.*p* = `$3`; `$$`.*s* = (*hpos* − *hstart*) − `$3`;
hput_tags(`$1`, *hput_list*(`$1` + 1, &(`$$`))); };

Writing the long format: ⟹ − − −

⟨ write functions 21 ⟩ +≡ (144)

```
void hwrite_list(List *l)
{ uint32_t h = hpos − hstart, e = hend − hstart;   /* save hpos and hend */
  hpos = l→p + hstart; hend = hpos + l→s;
  if (KIND(l→t) ≡ list_kind) {
    if (INFO(l→t) & b100) ⟨ write a text 155 ⟩
    else ⟨ write a list 145 ⟩
  }
  else QUIT("List␣expected␣got␣%s", content_name[KIND(l→t)]);
  hpos = hstart + h; hend = hstart + e;   /* restore hpos and hend */
}
```

⟨ write a list 145 ⟩ ≡ (145)

```
{ if (l→s ≡ 0) hwritef("␣<>");
  else
  { DBG(DBGNODE, "Write␣list␣at␣0x%x␣size=%u\n", l→p, l→s);
      hwrite_start(); if (section_no ≡ 2) hwrite_label();
    if (l→s > #FF) hwritef("%d", l→s);
    while (hpos < hend) hget_content_node();
    hwrite_end();
  }
}
```

Used in 144.

Reading the short format: · · · ⟹

⟨ shared get functions 53 ⟩ +≡ (146)

```
void hget_size_boundary(Info info)
{ uint32_t n;
  info = info & #3;
  if (info ≡ 0) return;
  n = HGET8;
  if (n ≠ #100 − info) QUIT("Non␣matching␣boundary␣byte␣0x%x␣with␣in\
        fo␣value␣%d␣at␣0x%x", n, info, (uint32_t)(hpos − hstart − 1));
}
uint32_t hget_list_size(Info info)
{ uint32_t n = 0;
  info = info & #3;
  if (info ≡ 0) return 0;
  else if (info ≡ 1) n = HGET8;
  else if (info ≡ 2) HGET16(n);
  else if (info ≡ 3) HGET32(n);
  else QUIT("List␣info␣%d␣must␣be␣0,␣1,␣2,␣or␣3", info);
  return n;
```

```
  }
  void hget_list(List *l)
  { if (KIND(*hpos) ≠ list_kind ∧ KIND(*hpos) ≠ param_kind)
        QUIT("List␣expected␣at␣0x%x", (uint32_t)(hpos − hstart));
    else { ⟨ read the start byte a  16 ⟩
        l→t = a;
        HGET_LIST(INFO(a), *l);
        ⟨ read and check the end byte z  17 ⟩
        DBG(DBGNODE, "Get␣list␣at␣0x%x␣size=%u\n", l→p, l→s);
    }
  }
```

If a list has the info value zero, the list is the empty list. Other list references are currently not implemented.

```
⟨ shared get macros  38 ⟩ +≡                                        (147)
#define HGET_LIST(I, L)
  if (((I) & #3) ≡ 0) { uint8_t n = HGET8; REF_RNG(KIND((L).t), n); (L).s = 0; }
  else { (L).s = hget_list_size(I);
    hget_size_boundary(I);
    (L).p = hpos − hstart;
    hpos = hpos + (L).s;
    hget_size_boundary(I);
    { uint32_t s = hget_list_size(I);
      if (s ≠ (L).s)
        QUIT("List␣sizes␣at␣0x%x␣and␣"SIZE_F"␣do␣not␣match␣0x%x␣\
            !=␣0x%x", node_pos + 1, hpos − hstart − I − 1, (L).s, s);
    }
  }
```

Writing the short format: $\Longrightarrow \cdots$

```
⟨ put functions  14 ⟩ +≡                                            (148)
  uint8_t hsize_bytes(uint32_t n)
  { if (n ≡ 0) return 0;
    else if (n < #100) return 1;
    else if (n < #10000) return 2;
    else return 4;
  }
  void hput_list_size(uint32_t n, int i)
  { if (i ≡ 0) return;
    else if (i ≡ 1) HPUT8(n);
    else if (i ≡ 2) HPUT16(n);
    else HPUT32(n);
  }
```

```
Tag hput_list(uint32_t start_pos, List *l)
{ if (l→s ≡ 0) { hpos = hstart + start_pos; HPUT8(0);
      return TAG(KIND(l→t), INFO(l→t) & b100); }
  else
  { uint32_t list_end = hpos − hstart;
    int i = l→p − start_pos − 1;         /* number of byte allocated for size */
    int j = hsize_bytes(l→s);              /* number of byte needed for size */
    Info k;
    if (j ≡ 4) k = 3;
    else k = j;
    DBG(DBGNODE, "Put␣list␣at␣0x%x␣size=%u\n", l→p, l→s);
    if (i > j ∧ l→s > #100) j = i;                  /* avoid moving large lists */
    if (i ≠ j)
    { int d = j − i;
      DBG(DBGNODE, "Moving␣%u␣byte␣by␣%d\n", l→s, d);
      if (d > 0) HPUTX(d);
      memmove(hstart + l→p + d, hstart + l→p, l→s);
      ⟨adjust label positions after moving a list 258⟩
      l→p = l→p + d; list_end = list_end + d;
    }
    hpos = hstart + start_pos; hput_list_size(l→s, j); HPUT8(#100 − k);
    hpos = hstart + list_end; HPUT8(#100 − k); hput_list_size(l→s, j);
    return TAG(KIND(l→t), k | (INFO(l→t) & b100));
  }
}
```

4.2 Texts

A Text is a list of nodes with a representation optimized for character nodes. In the long format, a sequence of characters like "Hello" is written "<glyph 'H' *0> <glyph 'e' *0> <glyph 'l' *0> <glyph 'l' *0> <glyph 'o' *0>", and even in the short format it requires 4 byte per character! As a text, the same sequence is written ""Hello"" in the long format and the short format requires usually just 1 byte per character. Indeed except the bytes with values from #00 to #20, which are considered control codes, all bytes and all UTF-8 multibyte sequences are simply considered character codes. They are equivalent to a glyph node in the "current font". The current font is font number 0 at the beginning of a text and it can be changed using the control codes. We introduce the concept of a "current font" because we do not expect the font to change too often, and it allows for a more compact representation if we do not store the font with every character code. It has an important disadvantage though: storing only font changes prevents us from parsing a text backwards; we always have to start at the beginning of the text, where the font is known to be font number 0.

Defining a second format for encoding lists of nodes adds another difficulty to the problem we had discussed at the beginning of section 4. When we try to recover from an error and start reading a content stream at an arbitrary position, the first thing we need to find out is whether at this position we have the tag byte of an ordinary node or whether we have a position inside a text.

Inside a text, character nodes start with a byte in the range `#21`–`#F7`. This is a wide range and it overlaps considerably with the range of valid tag bytes. It is however possible to choose the kind-values in such a way that the control codes do not overlap with the valid tag bytes that start a node. For this reason, the values *list_kind* ≡ 0, *param_kind* ≡ 1, *range_kind* ≡ 2, *xdimen_kind* ≡ 3, and *adjust_kind* ≡ 4 were chosen on page 5. Lists, parameter lists, and extended dimensions occur only *inside* of content nodes, but are not content nodes in their own right; page ranges occur only in the definition section; so the values `#00` to `#1F` are not used as tag bytes of content nodes. The value `#20` would, as a tag byte, indicate an adjust node (*adjust_kind* ≡ 4) with info value zero. Because there are no predefined adjustments, `#20` is not used as a tag byte either. (An alternative choice would be to use the kind value 4 for paragraph nodes because there are no predefined paragraphs.)

The largest byte that starts an UTF8 code is `#F7`; hence, there are eight possible control codes, from `#F8` to `#FF`, available. The first three values `#F8`, `#F9`, and `#FA` are actually used for penalty nodes with info values, 0, 1, and 2. The last three `#FD`, `#FE`, and `#FF` are used as boundary marks for the text size and therefore we can use only `#FB` and `#FC` as control codes.

In the long format, we do not provide a syntax for specifying a size estimate as we did for plain lists, because we expect text to be quite short. We allocate two byte for the size and hope that this will prove to be sufficient most of the time. Further, we will disallow the use of non-printable ASCII codes, because these are—by definition—not very readable, and we will give special meaning to some of the printable ASCII codes because we will need a notation for the beginning and ending of a text, for nodes inside a text, and the control codes.

Here are the details:

- In the long format, a text starts and ends with a double quote character "`"`". In the short format, texts are encoded similar to lists setting the info bit *b100*.
- Arbitrary nodes can be embedded inside a text. In the long format, they are enclosed in pointed brackets `<` ... `>` as usual. In the short format, an arbitrary node can follow the control code *txt_node* = `#1E`. Because text may occur in nodes, the scanner needs to be able to parse texts nested inside nodes nested inside nodes nested inside texts ... To accomplish this, we use the "stack" option of `flex` and include the pushing and popping of the stack in the macros `SCAN_START` and `SCAN_END`.
- The space character "␣" with ASCII value `#20` stands in both formats for the font specific interword glue node (control code *txt_glue*).
- The hyphen character "`-`" in the long format and the control code *txt_hyphen* = `#1F` in the short format stand for the font specific discretionary hyphenation node.
- In the long format, the backslash character "\" is used as an escape character. It is used to introduce notations for control codes, as described below, and to access the character codes of those ASCII characters that otherwise carry a

special meaning. For example “\"” denotes the character code of the double quote character “"”; and similarly “\\”, “\<”, “\>”, “\␣”, and “\-” denote the character codes of “\”, “<”, “>”, “␣”, and “-” respectively.

- In the long format, a TAB-character (ASCII code #09) is silently converted to a space character (ASCII code #20); a NL-character (ASCII code #0A), together with surrounding spaces, TAB-characters, and CR-characters (ASCII code #0D), is silently converted to a single space character. All other ASCII characters in the range #00 to #1F are not allowed inside a text. This rule avoids the problems arising from “invisible” characters embedded in a text and it allows to break texts into lines, even with indentation, at word boundaries.

  To allow breaking a text into lines without inserting spaces, a NL-character together with surrounding spaces, TAB-characters, and CR-characters is completely ignored if the whole group of spaces, TAB-characters, CR-characters, and the NL-character is preceded by a backslash character.

  For example, the text “`"There␣is␣no␣more␣gas␣in␣the␣tank."`” can be written as

  ```
  "There␣is␣
  →   no more g\␣␣
  →   as in the tank."
  ```

  To break long lines when writing a long format file, we use the variable *txt_length* to keep track of the approximate length of the current line.

- The control codes *txt_font* = #00, #01, #02, . . . , and #07 are used to change the current font to font number 0, 1, 2, . . . , and 7. In the long format these control codes are written \0, \1, \2, . . . , and \7.
- The control code *txt_global* = #08 is followed by a second parameter byte. If the value of the parameter byte is n, it will set the current font to font number n. In the long format, the two byte sequence is written “\F$n$\” where n is the decimal representation of the font number.
- The control codes #09, #0A, #0B, #0C, #0D, #0E, #0F, and #10 are also followed by a second parameter byte. They are used to reference the global definitions of penalty, kern, ligature, disc, glue, language, rule, and image nodes. The parameter byte contains the reference number. For example, the byte sequence #09 #03 is equivalent to the node `<penalty *3>`. In the long format these two-byte sequences are written, “\P$n$\” (penalty), “\K$n$\” (kern), “\L$n$\” (ligature), “\D$n$\” (disc), “\G$n$\” (glue), “\S$n$\” (speak or German “Sprache”), “\R$n$\” (rule), and “\I$n$\” (image), where n is the decimal representation of the parameter value.
- The control codes from *txt_local* = #11 to #1C are used to reference one of the 12 font specific parameters. In the long format they are written “\a”, “\b”, “\c”, . . . , “\j”, “\k”,“\l”.
- The control code *txt_cc* = #1D is used as a prefix for an arbitrary character code represented as an UTF-8 multibyte sequence. Its main purpose is providing a method for including character codes less or equal to #20 which otherwise would be considered control codes. In the long format, the byte sequence is written “\C$n$\” where n is the decimal representation of the character code.

- The control code *txt_node* = #`1E` is used as a prefix for an arbitrary node in short format. In the long format, it is written "`<`" and is followed by the node content in long format terminated by "`>`".
- The control code *txt_hyphen* = #`1F` is used to access the font specific discretionary hyphen. In the long format it is simply written as "`-`".
- The control code *txt_glue* = #`20` is the space character, it is used to access the font specific interword glue. In the long format, we use the space character "␣" as well.
- The control code *txt_ignore* = #`FB` is ignored, its position can be used in a link to specify a position between two characters. In the long format it is written as "`\@`".
- The control code #`FC` is currently unused.

For the control codes, we define an enumeration type and for references, a reference type.

⟨ hint types 1 ⟩ +≡ (149)

```
typedef enum {
  txt_font = #00, txt_global = #08, txt_local = #11, txt_cc = #1D,
      txt_node = #1E, txt_hyphen = #1F, txt_glue = #20, txt_ignore = #FB
} Txt;
```

Reading the long format: − − − ⟹

⟨ scanning definitions 24 ⟩ +≡ (150)

```
%x TXT
```

⟨ symbols 2 ⟩ +≡ (151)

```
%token TXT_START TXT_END TXT_IGNORE
%token TXT_FONT_GLUE TXT_FONT_HYPHEN
%token < u > TXT_FONT TXT_LOCAL
%token < rf > TXT_GLOBAL
%token < u > TXT_CC
%type < u > text
```

⟨ scanning rules 3 ⟩ +≡ (152)

```
\"           SCAN_TXT_START; return TXT_START;
< TXT > {
\"           SCAN_TXT_END; return TXT_END;
"<"          SCAN_START; return START;
">"          QUIT(">␣not␣allowed␣in␣text␣mode");
\\\\         yylval.u = '\\'; return TXT_CC;
\\\"         yylval.u = '"'; return TXT_CC;
\\"<"        yylval.u = '<'; return TXT_CC;
\\">"        yylval.u = '>'; return TXT_CC;
```

```
\\"␣"            yylval.u = '␣'; return TXT_CC;
\\"-"            yylval.u = '-'; return TXT_CC;
\\"@"            return TXT_IGNORE;
[␣\t\r]*(\n[␣\t\r]*)+   return TXT_FONT_GLUE;
\\[␣\t\r]*\n[␣\t\r]*   ;
\\[0-7]          yylval.u = yytext[1] - '0'; return TXT_FONT;
\\F[0-9]+\\      SCAN_REF(font_kind); return TXT_GLOBAL;
\\P[0-9]+\\      SCAN_REF(penalty_kind); return TXT_GLOBAL;
\\K[0-9]+\\      SCAN_REF(kern_kind); return TXT_GLOBAL;
\\L[0-9]+\\      SCAN_REF(ligature_kind); return TXT_GLOBAL;
\\D[0-9]+\\      SCAN_REF(disc_kind); return TXT_GLOBAL;
\\G[0-9]+\\      SCAN_REF(glue_kind); return TXT_GLOBAL;
\\S[0-9]+\\      SCAN_REF(language_kind); return TXT_GLOBAL;
\\R[0-9]+\\      SCAN_REF(rule_kind); return TXT_GLOBAL;
\\I[0-9]+\\      SCAN_REF(image_kind); return TXT_GLOBAL;
\\C[0-9]+\\      SCAN_UDEC(yytext + 2); return TXT_CC;
\\[a-l]          yylval.u = yytext[1] - 'a'; return TXT_LOCAL;
"␣"              return TXT_FONT_GLUE;
"-"              return TXT_FONT_HYPHEN;
{UTF8_1}         SCAN_UTF8_1(yytext); return TXT_CC;
{UTF8_2}         SCAN_UTF8_2(yytext); return TXT_CC;
{UTF8_3}         SCAN_UTF8_3(yytext); return TXT_CC;
{UTF8_4}         SCAN_UTF8_4(yytext); return TXT_CC;
  }
```

⟨ scanning macros 23 ⟩ +≡ (153)

```
#define SCAN_REF(K) yylval.rf.k = K; yylval.rf.n = atoi(yytext + 2)
  static int scan_level = 0;
#define SCAN_START yy_push_state(INITIAL); if (1 ≡ scan_level++)
    hpos0 = hpos;
#define SCAN_END
  if (scan_level--) yy_pop_state();
  else QUIT("Too␣many␣'>'␣in␣line␣%d", yylineno)
#define SCAN_TXT_START BEGIN(TXT)
#define SCAN_TXT_END BEGIN(INITIAL)
```

⟨ parsing rules 5 ⟩ +≡ (154)

```
  list: TXT_START position
        { hpos += 4;        /* start byte, two size byte, and boundary byte */
        } text TXT_END
        { $$.t = TAG(list_kind, b110); $$.p = $4; $$.s = (hpos − hstart) − $4;
        hput_tags($2, hput_list($2 + 1, &($$))); };
```

text: *position* | *text* *txt*;

txt: TXT_CC { *hput_txt_cc*($1); }
 | TXT_FONT { REF(*font_kind*, $1); *hput_txt_font*($1); }
 | TXT_GLOBAL { REF($1.*k*, $1.*n*); *hput_txt_global*(&($1)); }
 | TXT_LOCAL { RNG("Font␣parameter", $1, 0, 11); *hput_txt_local*($1); }
 | TXT_FONT_GLUE { HPUTX(1); HPUT8(*txt_glue*); }
 | TXT_FONT_HYPHEN { HPUTX(1); HPUT8(*txt_hyphen*); }
 | TXT_IGNORE { HPUTX(1); HPUT8(*txt_ignore*); }
 | { HPUTX(1); HPUT8(*txt_node*); } *content_node*;

The following function keeps track of the position in the current line. It the line gets too long it will break the text at the next space character. If no suitable space character comes along, the line will be broken after any regular character.

Writing the long format: ⟹ − − −

⟨ write a text 155 ⟩ ≡ (155)

```
{ if (l→s ≡ 0) hwritef("␣\"\"");
  else
  { int pos = nesting + 20;                         /* estimate */
    hwritef("␣\"");
    while (hpos < hend)
    { int i = hget_txt();
      if (i < 0) {
        if (pos++ < 70) hwritec('␣');
        else hwrite_nesting(), pos = nesting;
      }
      else if (i ≡ 1 ∧ pos ≥ 100)
      { hwritec('\\'); hwrite_nesting(); pos = nesting; }
      else pos += i;
    }
    hwritec('"');
  }
}
```

Used in 144.

The function returns the number of characters written because this information is needed in *hget_txt* below.

⟨ write functions 21 ⟩ +≡ (156)

```
int hwrite_txt_cc(uint32_t c)
{ if (c < #20) return hwritef("\\C%d\\", c);
  else switch (c) {
    case '\\': return hwritef("\\\\");
    case '"': return hwritef("\\\"");
    case '<': return hwritef("\\<");
    case '>': return hwritef("\\>");
```

```
    case '␣': return hwritef("\\␣");
    case '-': return hwritef("\\-");
    default: return option_utf8 ? hwrite_utf8(c) : hwritef("\\C%d\\", c);
    }
}
```

Reading the short format: · · · ⟹

⟨ get macros 19 ⟩ +≡ (157)

```
#define HGET_GREF(K, S)
  { uint8_t n = HGET8; REF(K, n); return hwritef("\\"S"%d\\", n); }
```

The function *hget_txt* reads a text element and writes it immediately. To enable the insertion of line breaks when writing a text, we need to keep track of the number of characters in the current line. For this purpose the function *hget_txt* returns the number of characters written. It returns −1 if a space character needs to be written providing a good opportunity for a break.

⟨ get functions 18 ⟩ +≡ (158)

```
int hget_txt(void)
{ if (*hpos ≥ #80 ∧ *hpos ≤ #F7) {
    if (option_utf8) return hwrite_utf8(hget_utf8());
    else return hwritef("\\C%d\\", hget_utf8());
  }
  else
  { uint8_t a;
    a = HGET8;
    switch (a) {
    case txt_font + 0: return hwritef("\\0");
    case txt_font + 1: return hwritef("\\1");
    case txt_font + 2: return hwritef("\\2");
    case txt_font + 3: return hwritef("\\3");
    case txt_font + 4: return hwritef("\\4");
    case txt_font + 5: return hwritef("\\5");
    case txt_font + 6: return hwritef("\\6");
    case txt_font + 7: return hwritef("\\7");
    case txt_global + 0: HGET_GREF(font_kind, "F");
    case txt_global + 1: HGET_GREF(penalty_kind, "P");
    case txt_global + 2: HGET_GREF(kern_kind, "K");
    case txt_global + 3: HGET_GREF(ligature_kind, "L");
    case txt_global + 4: HGET_GREF(disc_kind, "D");
    case txt_global + 5: HGET_GREF(glue_kind, "G");
    case txt_global + 6: HGET_GREF(language_kind, "S");
    case txt_global + 7: HGET_GREF(rule_kind, "R");
    case txt_global + 8: HGET_GREF(image_kind, "I");
    case txt_local + 0: return hwritef("\\a");
    case txt_local + 1: return hwritef("\\b");
```

```
      case txt_local + 2: return hwritef("\\c");
      case txt_local + 3: return hwritef("\\d");
      case txt_local + 4: return hwritef("\\e");
      case txt_local + 5: return hwritef("\\f");
      case txt_local + 6: return hwritef("\\g");
      case txt_local + 7: return hwritef("\\h");
      case txt_local + 8: return hwritef("\\i");
      case txt_local + 9: return hwritef("\\j");
      case txt_local + 10: return hwritef("\\k");
      case txt_local + 11: return hwritef("\\l");
      case txt_cc: return hwrite_txt_cc(hget_utf8());
      case txt_node:
        { int i;
          ⟨ read the start byte a 16 ⟩
          i = hwritef("<");
          i += hwritef("%s", content_name[KIND(a)]); hget_content(a);
          ⟨ read and check the end byte z 17 ⟩
          hwritec('>'); return i + 10;                /* just an estimate */
        }
      case txt_hyphen: hwritec('-'); return 1;
      case txt_glue: return −1;
      case '<': return hwritef("\\<");
      case '>': return hwritef("\\>");
      case '"': return hwritef("\\\"");
      case '-': return hwritef("\\-");
      case txt_ignore: return hwritef("\\@");
      default: hwritec(a); return 1;
      }
    }
  }
```

Writing the short format: ⟹ ···

```
⟨ put functions 14 ⟩ +≡                                              (159)
  void hput_txt_cc(uint32_t c)
  { if (c ≤ #20) { HPUTX(2);
      HPUT8(txt_cc); HPUT8(c); }
    else hput_utf8(c);
  }
  void hput_txt_font(uint8_t f)
  { if (f < 8) HPUTX(1), HPUT8(txt_font + f);
    else QUIT("Use␣\\F%d\\␣instead␣of␣\\%d␣for␣font␣%d␣in␣a␣text", f,
          f, f);
  }
```

```
void hput_txt_global(Ref *d)
{ HPUTX(2);
  switch (d->k) {
  case font_kind: HPUT8(txt_global + 0); break;
  case penalty_kind: HPUT8(txt_global + 1); break;
  case kern_kind: HPUT8(txt_global + 2); break;
  case ligature_kind: HPUT8(txt_global + 3); break;
  case disc_kind: HPUT8(txt_global + 4); break;
  case glue_kind: HPUT8(txt_global + 5); break;
  case language_kind: HPUT8(txt_global + 6); break;
  case rule_kind: HPUT8(txt_global + 7); break;
  case image_kind: HPUT8(txt_global + 8); break;
  default:
    QUIT("Kind %s not allowed as a global reference in a text",
        NAME(d->k));
  }
  HPUT8(d->n);
}
void hput_txt_local(uint8_t n)
{ HPUTX(1);
  HPUT8(txt_local + n);
}
```

5 Composite Nodes

The nodes that we consider in this section can contain one or more list nodes. When we implement the parsing routines for composite nodes in the long format, we have to take into account that parsing such a list node will already write the list node to the output. So we split the parsing of composite nodes into several parts and store the parts immediately after parsing them. On the parse stack, we will only keep track of the info value. This new strategy is not as transparent as our previous strategy used for simple nodes where we had a clean separation of reading and writing: reading would store the internal representation of a node and writing the internal representation to output would start only after reading is completed. The new strategy, however, makes it easier to reuse the grammar rules for the component nodes.

Another rule applies to composite nodes: in the short format, the subnodes will come at the end of the node, and especially a list node that contains content nodes comes last. This helps when traversing the content section as we will see in appendix A.

5.1 Boxes

The central structuring elements of TEX are boxes. Boxes have a height h, a depth d, and a width w. The shift amount a shifts the contents of the box, the glue ratio r is a factor applied to the glue inside the box, the glue order o is its order of stretchability, and the glue sign s is -1 for shrinking, 0 for rigid, and $+1$ for stretching. Most importantly, a box contains a list l of content nodes inside the box.

⟨ hint types 1 ⟩ +≡ (160)

typedef struct
{ **Dimen** h, d, w, a; **float32_t** r; **int8_t** s, o; **List** l; } **Box**;

There are two types of boxes: horizontal boxes and vertical boxes. The difference between the two is simple: a horizontal box aligns the reference points of its content nodes horizontally, and a positive shift amount a shifts the box down; a vertical box aligns the reference points vertically, and a positive shift amount a shifts the box right.

Not all box parameters are used frequently. In the short format, we use the info bits to indicated which of the parameters are used. Where as the width of a horizontal box is most of the time (80%) nonzero, other parameters are most of the time zero, like the shift amount (99%) or the glue settings (99.8%). The depth is zero in about 77%, the height in about 53%, and both together are zero in about 47%. The results for vertical boxes, which constitute about 20% of all boxes, are similar, except that the depth is zero in about 89%, but the height and width are almost never zero. For this reason we use bit *b001* to indicate a nonzero depth, bit *b010* for a nonzero shift amount, and *b100* for nonzero glue settings. Glue sign and glue order can be packed as two nibbles in a single byte.

Reading the long format: − − − ⟹

⟨ symbols 2 ⟩ +≡ (161)

```
%token HBOX "hbox"
%token VBOX "vbox"
%token SHIFTED "shifted"
%type <info> box box_dimen box_shift box_glue_set
```

⟨ scanning rules 3 ⟩ +≡ (162)

```
hbox        return HBOX;
vbox        return VBOX;
shifted     return SHIFTED;
```

⟨ parsing rules 5 ⟩ +≡ (163)

```
box_dimen: dimension dimension dimension
    { $$ = hput_box_dimen($1, $2, $3); };
box_shift: { $$ = b000; } | SHIFTED dimension { $$ = hput_box_shift($2); };
box_glue_set: { $$ = b000; }
  | PLUS stretch { $$ = hput_box_glue_set(+1, $2.f, $2.o); }
  | MINUS stretch { $$ = hput_box_glue_set(−1, $2.f, $2.o); };
box: box_dimen box_shift box_glue_set list { $$ = $1 | $2 | $3; };
hbox_node: start HBOX box END { hput_tags($1, TAG(hbox_kind, $3)); };
vbox_node: start VBOX box END { hput_tags($1, TAG(vbox_kind, $3)); };
content_node: hbox_node | vbox_node;
```

Writing the long format: ⟹ − − −

⟨ write functions 21 ⟩ +≡ (164)

```
void hwrite_box(Box *b)
{ hwrite_dimension(b→h);
  hwrite_dimension(b→d);
  hwrite_dimension(b→w);
  if (b→a ≠ 0) { hwritef("␣shifted"); hwrite_dimension(b→a); }
  if (b→r ≠ 0.0 ∧ b→s ≠ 0)
  { if (b→s > 0) hwritef("␣plus"); else hwritef("␣minus");
      hwrite_float64(b→r, false); hwrite_order(b→o);
  }
  hwrite_list(&(b→l));
}
```

Reading the short format: $\cdots \Longrightarrow$

⟨ cases to get content 20 ⟩ +≡ (165)

```
case TAG(hbox_kind, b000):
  { Box b; HGET_BOX(b000, b); hwrite_box(&b); } break;
case TAG(hbox_kind, b001):
  { Box b; HGET_BOX(b001, b); hwrite_box(&b); } break;
case TAG(hbox_kind, b010):
  { Box b; HGET_BOX(b010, b); hwrite_box(&b); } break;
case TAG(hbox_kind, b011):
  { Box b; HGET_BOX(b011, b); hwrite_box(&b); } break;
case TAG(hbox_kind, b100):
  { Box b; HGET_BOX(b100, b); hwrite_box(&b); } break;
case TAG(hbox_kind, b101):
  { Box b; HGET_BOX(b101, b); hwrite_box(&b); } break;
case TAG(hbox_kind, b110):
  { Box b; HGET_BOX(b110, b); hwrite_box(&b); } break;
case TAG(hbox_kind, b111):
  { Box b; HGET_BOX(b111, b); hwrite_box(&b); } break;
case TAG(vbox_kind, b000):
  { Box b; HGET_BOX(b000, b); hwrite_box(&b); } break;
case TAG(vbox_kind, b001):
  { Box b; HGET_BOX(b001, b); hwrite_box(&b); } break;
case TAG(vbox_kind, b010):
  { Box b; HGET_BOX(b010, b); hwrite_box(&b); } break;
case TAG(vbox_kind, b011):
  { Box b; HGET_BOX(b011, b); hwrite_box(&b); } break;
case TAG(vbox_kind, b100):
  { Box b; HGET_BOX(b100, b); hwrite_box(&b); } break;
case TAG(vbox_kind, b101):
  { Box b; HGET_BOX(b101, b); hwrite_box(&b); } break;
case TAG(vbox_kind, b110):
  { Box b; HGET_BOX(b110, b); hwrite_box(&b); } break;
case TAG(vbox_kind, b111):
  { Box b; HGET_BOX(b111, b); hwrite_box(&b); } break;
```

⟨ get macros 19 ⟩ +≡ (166)

```
#define HGET_BOX(I, B) HGET32 (B.h);
  if ((I) & b001) HGET32(B.d); else B.d = 0;
  HGET32(B.w);
  if ((I) & b010) HGET32(B.a); else B.a = 0;
  if ((I) & b100)
  { B.r = hget_float32(); B.s = HGET8; B.o = B.s & #F; B.s = B.s >> 4; }
  else { B.r = 0.0; B.o = B.s = 0; }
  hget_list(&(B.l));
```

```
⟨ get functions 18 ⟩ +≡                                                    (167)
  void hget_hbox_node(void)
  { Box b;
    ⟨ read the start byte a 16 ⟩
    if (KIND(a) ≠ hbox_kind)
      QUIT("Hbox␣expected␣at␣0x%x␣got␣%s", node_pos, NAME(a));
    HGET_BOX(INFO(a), b);
    ⟨ read and check the end byte z 17 ⟩
    hwrite_start(); hwritef("hbox"); hwrite_box(&b); hwrite_end();
  }
  void hget_vbox_node(void)
  { Box b;
    ⟨ read the start byte a 16 ⟩
    if (KIND(a) ≠ vbox_kind)
      QUIT("Vbox␣expected␣at␣0x%x␣got␣%s", node_pos, NAME(a));
    HGET_BOX(INFO(a), b);
    ⟨ read and check the end byte z 17 ⟩
    hwrite_start(); hwritef("vbox"); hwrite_box(&b); hwrite_end();
  }
```

Writing the short format: ⟹ ···

```
⟨ put functions 14 ⟩ +≡                                                    (168)
  Info hput_box_dimen(Dimen h, Dimen d, Dimen w)
  { Info i; HPUT32(h);
    if (d ≠ 0) { HPUT32(d); i = b001; } else i = b000;
    HPUT32(w);
    return i;
  }
  Info hput_box_shift(Dimen a)
  { if (a ≠ 0) { HPUT32(a); return b010; } else return b000;
  }
  Info hput_box_glue_set(int8_t s, float32_t r, Order o)
  { if (r ≠ 0.0 ∧ s ≠ 0) { hput_float32(r); HPUT8((s ≪ 4) | o); return b100; }
    else return b000;
  }
```

5.2 Extended Boxes

HiTEX produces two kinds of extended horizontal boxes, *hpack_kind* and *hset_kind*, and the same for vertical boxes using *vpack_kind* and *vset_kind*. Let us focus on horizontal boxes; the handling of vertical boxes is completely parallel.

The *hpack* procedure of HiTEX produces an extended box of *hset_kind* either if it is given an extended dimension as its width or if it discovers that the width of its content is an extended dimension. After the final width of the box has been computed in the viewer, it just remains to set the glue; a very simple operation indeed.

If the *hpack* procedure of HiTEX can not determine the natural dimensions of the box content because it contains paragraphs or other extended boxes, it produces a box of *hpack_kind*. Now the viewer needs to traverse the list of content nodes to determine the natural dimensions. Even the amount of stretchability and shrinkability has to be determined in the viewer. For example, the final stretchability of a paragraph with some stretchability in the baseline skip will depend on the number of lines which, in turn, depends on `hsize`. It is not possible to merge these traversals of the box content with the traversal necessary when displaying the box. The latter needs to convert glue nodes into positioning instructions which requires a fixed glue ratio. The computation of the glue ratio, however, requires a complete traversal of the content.

In the short format of a box node of type *hset_kind*, *vset_kind*, *hpack_kind*, or *vpack_kind*, the info bit *b100* indicates, if set, a complete extended dimension, and if unset, a reference to a predefined extended dimension for the target size; the info bit *b010* indicates a nonzero shift amount. For a box of type *hset_kind* or *vset_kind*, the info bit *b001* indicates, if set, a nonzero depth. For a box of type *hpack_kind* or *vpack_kind*, the info bit *b001* indicates, if set, an additional target size, and if unset, an exact target size. For a box of type *vpack_kind* also the maximum depth is given. If in the long format the maximum depth is omitted, the value `MAX_DIMEN` is used.

The reference point of a vertical box is usually the reference point of the last box inside it and multiple vertical boxes are aligned along this common baseline. Occasionaly, however, we want to align vertical boxes using the baselines of their first box. We indicate this alternative setting of the reference point using the keyword `top` in the long form. In the short form, we use the fact the the absolut value of any dimension is less or equal to `MAX_DIMEN` which is equal to #`3fffffff`. This means that the two most significant bits are always the same. So a vtop node can be marked by toggling the second of these bits.

Reading the long format: $- - - \Longrightarrow$

⟨ symbols 2 ⟩ +≡ (169)
%**token** HPACK "hpack"
%**token** HSET "hset"
%**token** VPACK "vpack"
%**token** VSET "vset"
%**token** DEPTH "depth"
%**token** ADD "add"
%**token** TO "to"
%**type** < *info* > *box_options box_goal hpack vpack vbox_dimen*
%**type** < *d* > *max_depth*

⟨ scanning rules 3 ⟩ +≡ (170)
`hpack` **return** HPACK;
`hset` **return** HSET;

```
vpack          return VPACK;
vset           return VSET;
add            return ADD;
to             return TO;
depth          return DEPTH;
```

⟨ parsing rules 5 ⟩ +≡ (171)

box_flex: *plus minus* { *hput_stretch*(&($1)); *hput_stretch*(&($2)); };

box_options: *box_shift box_flex xdimen_ref list* { $$ = $1; }
  | *box_shift box_flex xdimen_node list* { $$ = $1 | *b100*; };

hxbox_node: *start* HSET *box_dimen box_options* END {
    hput_tags($1, TAG(*hset_kind*, $3 | $4)); };

vbox_dimen: *box_dimen*
  | TOP *dimension dimension dimension*
    { $$ = *hput_box_dimen*($2, $3 ⊕ #40000000, $4); };

vxbox_node: *start* VSET *vbox_dimen box_options* END {
    hput_tags($1, TAG(*vset_kind*, $3 | $4)); };

box_goal: TO *xdimen_ref* { $$ = *b000*; }
  | ADD *xdimen_ref* { $$ = *b001*; }
  | TO *xdimen_node* { $$ = *b100*; }
  | ADD *xdimen_node* { $$ = *b101*; };

hpack: *box_shift box_goal list* { $$ = $2; };

hxbox_node: *start* HPACK *hpack* END { *hput_tags*($1, TAG(*hpack_kind*, $3)); };

max_depth: { $$ = MAX_DIMEN; }
  | MAX DEPTH *dimension* { $$ = $3; };

vpack: *max_depth* { HPUT32($1); } *box_shift box_goal list* { $$ = $3 | $4; }
  | TOP *max_depth* { HPUT32($2 ⊕ #40000000); }
    box_shift box_goal list { $$ = $4 | $5; };

vxbox_node: *start* VPACK *vpack* END { *hput_tags*($1, TAG(*vpack_kind*, $3)); };

content_node: *vxbox_node*
  | *hxbox_node*;

Reading the short format: $\cdots \Longrightarrow$

$\langle$ cases to get content 20 $\rangle$ +≡ (172)

```
case TAG(hset_kind, b000): HGET_SET(hset_kind, b000); break;
case TAG(hset_kind, b001): HGET_SET(hset_kind, b001); break;
case TAG(hset_kind, b010): HGET_SET(hset_kind, b010); break;
case TAG(hset_kind, b011): HGET_SET(hset_kind, b011); break;
case TAG(hset_kind, b100): HGET_SET(hset_kind, b100); break;
case TAG(hset_kind, b101): HGET_SET(hset_kind, b101); break;
case TAG(hset_kind, b110): HGET_SET(hset_kind, b110); break;
case TAG(hset_kind, b111): HGET_SET(hset_kind, b111); break;

case TAG(vset_kind, b000): HGET_SET(vset_kind, b000); break;
case TAG(vset_kind, b001): HGET_SET(vset_kind, b001); break;
case TAG(vset_kind, b010): HGET_SET(vset_kind, b010); break;
case TAG(vset_kind, b011): HGET_SET(vset_kind, b011); break;
case TAG(vset_kind, b100): HGET_SET(vset_kind, b100); break;
case TAG(vset_kind, b101): HGET_SET(vset_kind, b101); break;
case TAG(vset_kind, b110): HGET_SET(vset_kind, b110); break;
case TAG(vset_kind, b111): HGET_SET(vset_kind, b111); break;

case TAG(hpack_kind, b000): HGET_PACK(hpack_kind, b000); break;
case TAG(hpack_kind, b001): HGET_PACK(hpack_kind, b001); break;
case TAG(hpack_kind, b010): HGET_PACK(hpack_kind, b010); break;
case TAG(hpack_kind, b011): HGET_PACK(hpack_kind, b011); break;
case TAG(hpack_kind, b100): HGET_PACK(hpack_kind, b100); break;
case TAG(hpack_kind, b101): HGET_PACK(hpack_kind, b101); break;
case TAG(hpack_kind, b110): HGET_PACK(hpack_kind, b110); break;
case TAG(hpack_kind, b111): HGET_PACK(hpack_kind, b111); break;

case TAG(vpack_kind, b000): HGET_PACK(vpack_kind, b000); break;
case TAG(vpack_kind, b001): HGET_PACK(vpack_kind, b001); break;
case TAG(vpack_kind, b010): HGET_PACK(vpack_kind, b010); break;
case TAG(vpack_kind, b011): HGET_PACK(vpack_kind, b011); break;
case TAG(vpack_kind, b100): HGET_PACK(vpack_kind, b100); break;
case TAG(vpack_kind, b101): HGET_PACK(vpack_kind, b101); break;
case TAG(vpack_kind, b110): HGET_PACK(vpack_kind, b110); break;
case TAG(vpack_kind, b111): HGET_PACK(vpack_kind, b111); break;
```

$\langle$ get macros 19 $\rangle$ +≡ (173)

```
#define HGET_SET(K, I)
  { Dimen h, d; HGET32(h);
    if ((I) & b001) HGET32(d); else d = 0;
    if (K ≡ vset_kind ∧ (d > MAX_DIMEN ∨ d < −MAX_DIMEN)) { hwritef("␣top");
      d ⊕= #40000000;
    }
    hwrite_dimension(h);
    hwrite_dimension(d); }
```

```
{ Dimen w; HGET32(w); hwrite_dimension(w); }
if ((I) & b010) { Dimen a; HGET32(a);
   hwritef("␣shifted"); hwrite_dimension(a); }
{ Stretch p; HGET_STRETCH(p); hwrite_plus(&p); }
{ Stretch m; HGET_STRETCH(m); hwrite_minus(&m); }
if ((I) & b100) { Xdimen x; hget_xdimen_node(&x); hwrite_xdimen_node(&x);
   }
else HGET_REF(xdimen_kind);
{ List l; hget_list(&l); hwrite_list(&l); }
#define HGET_PACK(K, I)
if (K ≡ vpack_kind) { Dimen d;
   HGET32(d);
   if (d > MAX_DIMEN ∨ d < −MAX_DIMEN) { hwritef("␣top");
      d ⊕= #40000000;
   }
   if (d ≠ MAX_DIMEN) { hwritef("␣max␣depth"); hwrite_dimension(d);
   }
}
if ((I) & b010) { Dimen s;
   HGET32(s);
   hwritef("␣shifted"); hwrite_dimension(s);
}
if ((I) & b001) hwritef("␣add"); else hwritef("␣to");
if ((I) & b100) { Xdimen x; hget_xdimen_node(&x); hwrite_xdimen_node(&x);
   }
else HGET_REF(xdimen_kind);
{ List l; hget_list(&l); hwrite_list(&l); }
```

5.3 Leaders

Leaders are a special type of glue that is best explained by a few examples. Where as ordinary glue fills its designated space with whiteness, leaders fill their designated space with either a rule ________________ or some sort of repeated . . . . content. In multiple leaders, the dots . . . . . are usually aligned across lines, as in the last . . . . three lines. Unless you specify centered . . . . leaders or you specify expanded . . . . leaders. The former pack the repeated content tight and center the repeated content in the available space, the latter distributes the extra space between all the repeated instances.

In the short format, the two lowest info bits store the type of leaders: 1 for aligned, 2 for centered, and 3 for expanded. The *b100* info bit is usually set and only zero in the unlikely case that the glue is zero and therefore not present.

Reading the long format: $- - - \Longrightarrow$

⟨ symbols 2 ⟩ +≡ (174)
%**token** LEADERS "leaders"
%**token** ALIGN "align"
%**token** CENTER "center"
%**token** EXPAND "expand"
%**type** < *info* > *leaders*
%**type** < *info* > *ltype*

⟨ scanning rules 3 ⟩ +≡ (175)

```
leaders     return LEADERS;
align       return ALIGN;
center      return CENTER;
expand      return EXPAND;
```

⟨ parsing rules 5 ⟩ +≡ (176)

ltype: { $$ = 1; }
  | ALIGN { $$ = 1; } | CENTER { $$ = 2; } | EXPAND { $$ = 3; };

leaders: *glue_node ltype rule_node* { **if** ($1) $$ = $2 | *b100*; **else** $$ = $2; }
  | *glue_node ltype hbox_node* { **if** ($1) $$ = $2 | *b100*; **else** $$ = $2; }
  | *glue_node ltype vbox_node* { **if** ($1) $$ = $2 | *b100*; **else** $$ = $2; };

content_node: *start* LEADERS *leaders* END
    { *hput_tags*($1, TAG(*leaders_kind*, $3)); }

Writing the long format: $\Longrightarrow - - -$

⟨ write functions 21 ⟩ +≡ (177)

void *hwrite_leaders_type*(**int** *t*)
{ **if** ($t \equiv 2$) *hwritef*("␣center");
  else if ($t \equiv 3$) *hwritef*("␣expand");
}

Reading the short format: $\cdots \Longrightarrow$

⟨ cases to get content 20 ⟩ +≡ (178)

case TAG(*leaders_kind*, 1): HGET_LEADERS(1); **break**;
case TAG(*leaders_kind*, 2): HGET_LEADERS(2); **break**;
case TAG(*leaders_kind*, 3): HGET_LEADERS(3); **break**;
case TAG(*leaders_kind*, *b100* | 1): HGET_LEADERS(*b100* | 1); **break**;
case TAG(*leaders_kind*, *b100* | 2): HGET_LEADERS(*b100* | 2); **break**;
case TAG(*leaders_kind*, *b100* | 3): HGET_LEADERS(*b100* | 3); **break**;

⟨ get macros 19 ⟩ +≡ (179)

```
#define HGET_LEADERS(I)
  if ((I) & b100) hget_glue_node();
  hwrite_leaders_type((I) & b011);
  if (KIND(*hpos) ≡ rule_kind) hget_rule_node();
  else if (KIND(*hpos) ≡ hbox_kind) hget_hbox_node();
  else hget_vbox_node();
```

5.4 Baseline Skips

Baseline skips are small amounts of glue inserted between two consecutive lines of text. To get nice looking pages, the amount of glue inserted must take into account the depth of the line above the glue and the height of the line below the glue to achieve a constant distance of the baselines. For example, if we have the lines

> "There is no
> more gas
> in the tank."

TEX will insert 7.69446pt of baseline skip between the first and the second line and 3.11111pt of baseline skip between the second and the third line. This is due to the fact that the first line has no descenders, its depth is zero, the second line has no ascenders but the "g" descends below the baseline, and the third line has ascenders ("t", "h",...) so it is higher than the second line. TEX's choice of baseline skips ensures that the baselines are exactly 12pt apart in both cases.

Things get more complicated if the text contains mathematical formulas because then a line can get so high or deep that it is impossible to keep the distance between baselines constant without two adjacent lines touching each other. In such cases, TEX will insert a small minimum line skip glue.

For the whole computation, TEX uses three parameters: `baselineskip`, `lineskiplimit`, and `lineskip`. `baselineskip` is a glue value; its size is the normal distance of two baselines. TEX adjusts the size of the `baselineskip` glue for the height and the depth of the two lines and then checks the result against `lineskiplimit`. If the result is smaller than `lineskiplimit` it will use the `lineskip` glue instead.

Because the depth and the height of lines depend on the outcome of the line breaking routine, baseline computations must be done in the viewer. The situation gets even more complicated because TEX can manipulate the insertion of baseline skips in various ways. Therefore HINT requires the insertion of baseline nodes wherever the viewer is supposed to perform a baseline skip computation.

In the short format of a baseline definition, we store only the nonzero components and use the info bits to mark them: *b100* implies $bs \neq 0$, *b010* implies $ls \neq 0$, and *b001* implies $lslimit \neq 0$. If the baseline has only zero components, we put a reference to baseline number 0 in the output.

⟨ hint basic types 6 ⟩ +≡ (180)

```
typedef struct { Glue bs, ls; Dimen lsl; } Baseline;
```

Reading the long format: - - - ⟹

⟨ symbols 2 ⟩ +≡ (181)

```
%token BASELINE "baseline"
%type < info > baseline
```

⟨ scanning rules 3 ⟩ +≡ (182)

```
baseline        return BASELINE;
```

⟨ parsing rules 5 ⟩ +≡ (183)

```
  baseline:  dimension {
          if ($1 ≠ 0) HPUT32($1);
          } glue_node glue_node
        { $$ = b000;
          if ($1 ≠ 0) $$ |= b001;
          if ($3) $$ |= b100;
          if ($4) $$ |= b010; };
  content_node:  start BASELINE baseline END
        { if ($3 ≡ b000) HPUT8(0); hput_tags($1, TAG(baseline_kind, $3)); };
```

Reading the short format: ⋯ ⟹

⟨ cases to get content 20 ⟩ +≡ (184)

```
  case TAG(baseline_kind, b001):
    { Baseline b; HGET_BASELINE(b001, b); } break;
  case TAG(baseline_kind, b010):
    { Baseline b; HGET_BASELINE(b010, b); } break;
  case TAG(baseline_kind, b011):
    { Baseline b; HGET_BASELINE(b011, b); } break;
  case TAG(baseline_kind, b100):
    { Baseline b; HGET_BASELINE(b100, b); } break;
  case TAG(baseline_kind, b101):
    { Baseline b; HGET_BASELINE(b101, b); } break;
  case TAG(baseline_kind, b110):
    { Baseline b; HGET_BASELINE(b110, b); } break;
  case TAG(baseline_kind, b111):
    { Baseline b; HGET_BASELINE(b111, b); } break;
```

⟨ get macros 19 ⟩ +≡ (185)

```
#define HGET_BASELINE(I, B)
  if ((I) & b001) HGET32((B).lsl); else B.lsl = 0;
  hwrite_dimension(B.lsl);
  if ((I) & b100) hget_glue_node();
  else { B.bs.p.o = B.bs.m.o = B.bs.w.w = 0;
    B.bs.w.h = B.bs.w.v = B.bs.p.f = B.bs.m.f = 0.0;
    hwrite_glue_node(&(B.bs)); }
  if ((I) & b010) hget_glue_node();
```

```
else { B.ls.p.o = B.ls.m.o = B.ls.w.w = 0;
  B.ls.w.h = B.ls.w.v = B.ls.p.f = B.ls.m.f = 0.0;
  hwrite_glue_node(&(B.ls)); }
```

Writing the short format: ⟹ ⋯

⟨ put functions 14 ⟩ +≡ (186)

```
Tag hput_baseline(Baseline *b)
{ Info info = b000;
  if (¬ZERO_GLUE(b→bs)) info |= b100;
  if (¬ZERO_GLUE(b→ls)) info |= b010;
  if (b→lsl ≠ 0) { HPUT32(b→lsl); info |= b001; }
  return TAG(baseline_kind, info);
}
```

5.5 Ligatures

Ligatures occur only in horizontal lists. They specify characters that combine the glyphs of several characters into one specialized glyph. For example in the word "*difficult*" the three letters "*ffi*" are combined into the ligature "*ﬃ*". Hence, a ligature is very similar to a simple glyph node; the characters that got replaced are, however, retained in the ligature because they might be needed for example to support searching. Since ligatures are therefore only specialized list of characters and since we have a very efficient way to store such lists of characters, namely as a *text*, input and output of ligatures is quite simple.

The info value zero is reserved for references to a ligature. If the info value is between 1 and 6, it gives the number of bytes used to encode the characters in UTF8. Note that a ligature will always include a glyph byte, so the minimum size is 1. A typical ligature like "*ﬁ*" will need 3 byte: the ligature character "*ﬁ*", and the replacement characters "f" and "i". More byte might be required if the character codes exceed #7F since we use the UTF8 encoding scheme for larger character codes. If the info value is 7, a full text node follows the font byte. In the long format, we give the font, the character code, and then the replacement characters represented as a text.

⟨ hint types 1 ⟩ +≡ (187)

```
typedef struct { uint8_t f; List l; } Lig;
```

Reading the long format: $- - - \Longrightarrow$

⟨ symbols 2 ⟩ +≡ (188)

%token LIGATURE `"ligature"`
%type $< u >$ *lig_cc*
%type $< lg >$ *ligature*
%type $< u >$ *ref*

⟨ scanning rules 3 ⟩ +≡ (189)

`ligature` **return** LIGATURE;

⟨ parsing rules 5 ⟩ +≡ (190)

cc_list: | *cc_list* TXT_CC { *hput_utf8*(`$2`); };

lig_cc: UNSIGNED { `RNG("UTF-8␣code", $1, 0, #1FFFFF)`; `$$` = *hpos* − *hstart*;
hput_utf8(`$1`); };

lig_cc: CHARCODE { `$$` = *hpos* − *hstart*; *hput_utf8*(`$1`); };

ref: REFERENCE { `HPUT8($1)`; `$$ = $1`; };

ligature: *ref* { `REF`(*font_kind*, `$1`); } *lig_cc* TXT_START *cc_list* TXT_END
{ `$$`.*f* = `$1`; `$$`.*l.p* = `$3`; `$$`.*l.s* = (*hpos* − *hstart*) − `$3`;
`RNG("Ligature␣size", $$.l.s, 0, 255)`; };

content_node: *start* LIGATURE *ligature* END {
hput_tags(`$1`, *hput_ligature*(&(`$3`))); };

Writing the long format: $\Longrightarrow - - -$

⟨ write functions 21 ⟩ +≡ (191)

```
void hwrite_ligature(Lig *l)
{ uint32_t pos = hpos − hstart;
   hwrite_ref(l→f);
   hpos = l→l.p + hstart;
   hwrite_charcode(hget_utf8());
   hwritef("␣\"");
   while (hpos < hstart + l→l.p + l→l.s) hwrite_txt_cc(hget_utf8());
   hwritec('"');
   hpos = hstart + pos;
}
```

Reading the short format: $\cdots \Longrightarrow$

⟨ cases to get content 20 ⟩ +≡ (192)

```
case TAG(ligature_kind, 1): { Lig l; HGET_LIG(1, l); } break;
case TAG(ligature_kind, 2): { Lig l; HGET_LIG(2, l); } break;
case TAG(ligature_kind, 3): { Lig l; HGET_LIG(3, l); } break;
case TAG(ligature_kind, 4): { Lig l; HGET_LIG(4, l); } break;
case TAG(ligature_kind, 5): { Lig l; HGET_LIG(5, l); } break;
case TAG(ligature_kind, 6): { Lig l; HGET_LIG(6, l); } break;
case TAG(ligature_kind, 7): { Lig l; HGET_LIG(7, l); } break;
```

⟨ get macros 19 ⟩ +≡ (193)

```
#define HGET_LIG(I, L)
  (L).f = HGET8;
  REF(font_kind, (L).f);
  if ((I) ≡ 7) hget_list(&((L).l));
  else { (L).l.s = (I);
    (L).l.p = hpos − hstart; hpos += (L).l.s;
  }
  hwrite_ligature(&(L));
```

Writing the short format: $\Longrightarrow \cdots$

⟨ put functions 14 ⟩ +≡ (194)

```
Tag hput_ligature(Lig *l)
{ if (l→l.s < 7) return TAG(ligature_kind, l→l.s);
  else
  { uint32_t pos = l→l.p;
    hput_tags(pos, hput_list(pos + 1, &(l→l)));
    return TAG(ligature_kind, 7);
  }
}
```

5.6 Discretionary breaks

HINT is capable to break lines into paragraphs. It does this primarily at interword spaces but it might also break a line in the middle of a word if it finds a discretionary line break there. These discretionary breaks are usually provided by an automatic hyphenation algorithm but they might be also explicitly inserted by the author of a document.

When a line break occurs at such a discretionary break, the line before the break ends with a *pre_break* list of nodes, the line after the break starts with a *post_break* list of nodes, and the next *replace_count* nodes after the discretionary break will be ignored. Both lists must consist entirely of glyphs, kerns, boxes, rules, or ligatures. For example, an ordinary discretionary break will have a *pre_break* list containing "-", an empty *post_break* list, and a *replace_count* of zero.

The long format starts with an optional "!", indicating an explicit discretionary break, followed by the replace-count. Then comes the pre-break list followed by the post-break list. The replace-count can be omitted if it is zero; an empty post-break list may be omitted as well. Both list may be omitted only if both are empty.

In the short format, the three components of a disc node are stored in this order: *replace_count*, *pre_break* list, and *post_break* list. The *b100* bit in the info value indicates the presence of a replace-count, the *b010* bit the presence of a *pre_break* list, and the *b001* bit the presence of a *post_break* list. Since the info value *b000* is reserved for references, at least one of these must be specified; so we represent a node with empty lists and a replace count of zero using the info value *b100* and a zero byte for the replace count.

Replace counts must be in the range 0 to 31; so the short format can set the high bit of the replace count to indicate an explicit break.

⟨ hint types 1 ⟩ +≡ (195)

typedef struct { **bool** *x*; **List** *p*, *q*; **uint8_t** *r*; } **Disc**;

Reading the long format: − − − ⟹

⟨ symbols 2 ⟩ +≡ (196)

%**token** DISC "disc"
%**type** < *dc* > *disc*
%**type** < *u* > *replace_count*

⟨ scanning rules 3 ⟩ +≡ (197)

disc **return** DISC;

⟨ parsing rules 5 ⟩ +≡ (198)

replace_count: *explicit* { **if** ($1) { $$ = #80; HPUT8(#80); } **else** $$ = #00; }
 | *explicit* UNSIGNED { RNG("Replace␣count", $2, 0, 31);
  $$ = ($2) | (($1) ? #80 : #00); **if** ($$ ≠ 0) HPUT8($$); };

disc: *replace_count list list* { $$.*r* = $1; $$.*p* = $2; $$.*q* = $3;
  if ($3.*s* ≡ 0) { *hpos* = *hpos* − 3; **if** ($2.*s* ≡ 0) *hpos* = *hpos* − 3; } }
 | *replace_count list* { $$.*r* = $1; $$.*p* = $2;
  if ($2.*s* ≡ 0) *hpos* = *hpos* − 3; $$.*q*.*s* = 0; }
 | *replace_count* { $$.*r* = $1; $$.*p*.*s* = 0; $$.*q*.*s* = 0; };

disc_node: *start* DISC *disc* END { *hput_tags*($1, *hput_disc*(&($3))); };

content_node: *disc_node*;

Writing the long format: ⟹ − − −

```
⟨ write functions 21 ⟩ +≡                                                    (199)
  void hwrite_disc(Disc *h)
  { hwrite_explicit(h→x);
    if (h→r ≠ 0) hwritef("␣%d", h→r);
    if (h→p.s ≠ 0 ∨ h→q.s ≠ 0) hwrite_list(&(h→p));
    if (h→q.s ≠ 0) hwrite_list(&(h→q));
  }
  void hwrite_disc_node(Disc *h)
  { hwrite_start(); hwritef("disc"); hwrite_disc(h); hwrite_end();
  }
```

Reading the short format: · · · ⟹

```
⟨ cases to get content 20 ⟩ +≡                                               (200)
  case TAG(disc_kind, b001):
    { Disc h; HGET_DISC(b001, h); hwrite_disc(&h); } break;
  case TAG(disc_kind, b010):
    { Disc h; HGET_DISC(b010, h); hwrite_disc(&h); } break;
  case TAG(disc_kind, b011):
    { Disc h; HGET_DISC(b011, h); hwrite_disc(&h); } break;
  case TAG(disc_kind, b100):
    { Disc h; HGET_DISC(b100, h); hwrite_disc(&h); } break;
  case TAG(disc_kind, b101):
    { Disc h; HGET_DISC(b101, h); hwrite_disc(&h); } break;
  case TAG(disc_kind, b110):
    { Disc h; HGET_DISC(b110, h); hwrite_disc(&h); } break;
  case TAG(disc_kind, b111):
    { Disc h; HGET_DISC(b111, h); hwrite_disc(&h); } break;
```

```
⟨ get macros 19 ⟩ +≡                                                         (201)
#define HGET_DISC(I, Y)
  if ((I) & b100) { uint8_t r = HGET8;
    (Y).r = r & #7F; RNG("Replace␣count", (Y).r, 0, 31); (Y).x = (r & #80) ≠ 0;
    } else { (Y).r = 0; (Y).x = false; }
  if ((I) & b010) hget_list(&((Y).p));
  else { (Y).p.p = hpos − hstart; (Y).p.s = 0; (Y).p.t = TAG(list_kind, b000); }
  if ((I) & b001) hget_list(&((Y).q));
  else { (Y).q.p = hpos − hstart; (Y).q.s = 0; (Y).q.t = TAG(list_kind, b000); }
```

```
⟨ get functions 18 ⟩ +≡                                                      (202)
  void hget_disc_node(Disc *h)
  { ⟨ read the start byte a 16 ⟩
    if (KIND(a) ≠ disc_kind ∨ INFO(a) ≡ b000)
      QUIT("Hyphen␣expected␣at␣0x%x␣got␣%s,%d", node_pos, NAME(a),
          INFO(a));
```

```
  HGET_DISC(INFO(a), *h);
  ⟨ read and check the end byte z 17 ⟩
}
```

When *hput_disc* is called, the node is already written to the output, but empty lists might have been deleted, and the info value needs to be determined. Because the info value *b000* is reserved for references, a zero reference count is written to avoid this case.

Writing the short format: ⟹ · · ·

⟨ put functions 14 ⟩ +≡ (203)

```
Tag hput_disc(Disc *h)
{ Info info = b000;
  if (h→r ≠ 0) info |= b100;
  if (h→q.s ≠ 0) info |= b011;
  else if (h→p.s ≠ 0) info |= b010;
  if (info ≡ b000) { info |= b100; HPUT8(0); }
  return TAG(disc_kind, info);
}
```

5.7 Paragraphs

The most important procedure that the HINT viewer inherits from TEX is the line breaking routine. If the horizontal size of the paragraph is not known, breaking the paragraph into lines must be postponed and this is done by creating a paragraph node. The paragraph node must contain all information that TEX's line breaking algorithm needs to do its job.

Besides the horizontal list describing the content of the paragraph and the extended dimension describing the horizontal size, this is the set of parameters that guide the line breaking algorithm:

- Integer parameters:
  `pretolerance` (badness tolerance before hyphenation),
  `tolerance` (badness tolerance after hyphenation),
  `line_penalty` (added to the badness of every line, increase to get fewer lines),
  `hyphen_penalty` (penalty for break after hyphenation break),
  `ex_hyphen_penalty` (penalty for break after explicit break),
  `double_hyphen_demerits` (demerits for double hyphen break),
  `final_hyphen_demerits` (demerits for final hyphen break),
  `adj_demerits` (demerits for adjacent incompatible lines),
  `looseness` (make the paragraph that many lines longer than its optimal size),
  `inter_line_penalty` (additional penalty between lines),
  `club_penalty` (penalty for creating a club line),
  `widow_penalty` (penalty for creating a widow line),
  `display_widow_penalty` (ditto, just before a display),

`broken_penalty` (penalty for breaking a page at a broken line),
`hang_after` (start/end hanging indentation at this line).

- Dimension parameters:
  `line_skip_limit` (threshold for `line_skip` instead of `baseline_skip`),
  `hang_indent` (amount of hanging indentation),
  `emergency_stretch` (stretchability added to every line in the final pass of line breaking).
- Glue parameters:
  `baseline_skip` (desired glue between baselines),
  `line_skip` (interline glue if `baseline_skip` is infeasible),
  `left_skip` (glue at left of justified lines),
  `right_skip` (glue at right of justified lines),
  `par_fill_skip` (glue on last line of paragraph).

For a detailed explanation of these parameters and how they influence line breaking, you should consult the TEXbook[8]; TEX's `parshape` feature is currently not implemented. There are default values for all of these parameters (see section 11), and therefore it might not be necessary to specify any of them. Any local adjustments are contained in a list of parameters contained in the paragraph node.

A further complication arises from displayed formulas that interrupt a paragraph. Such displays are described in the next section.

To summarize, a paragraph node in the long format specifies an extended dimension, a parameter list, and a node list. The extended dimension is given either as an *xdimen* node (info bit *b100*) or as a reference; similarly the parameter list can be embedded in the node (info bit *b010*) or again it is given by a reference.

Reading the long format: − − − ⟹

⟨ symbols 2 ⟩ +≡ (204)
%**token** PAR `"par"`
%**type** < *info* > *par*

⟨ scanning rules 3 ⟩ +≡ (205)
`par` **return** PAR;

The following parsing rules are slightly more complicated than I would like them to be, but it seems more important to achieve a regular layout of the short format nodes where all sub nodes are located at the end of a node. In this case, I want to put a *param_ref* before an *xdimen* node, but otherwise have the *xdimen_ref* before a *param_list*. The *par_dimen* rule is introduced only to avoid a reduce/reduce conflict in the parser.

⟨ parsing rules 5 ⟩ +≡ (206)
par_dimen: *xdimen* { *hput_xdimen_node*(&($1)); };
par: *xdimen_ref* *param_ref* *list* { $$ = *b000*; }
 | *xdimen_ref* *param_list* *list* { $$ = *b010*; }
 | *xdimen* *param_ref* { *hput_xdimen_node*(&($1)); } *list* { $$ = *b100*; }
 | *par_dimen* *param_list* *list* { $$ = *b110*; };

```
content_node: start PAR par END { hput_tags($1, TAG(par_kind, $3)); };
```

Reading the short format: ··· ⟹

⟨ cases to get content 20 ⟩ +≡ (207)

```
case TAG(par_kind, b000): HGET_PAR(b000); break;
case TAG(par_kind, b010): HGET_PAR(b010); break;
case TAG(par_kind, b100): HGET_PAR(b100); break;
case TAG(par_kind, b110): HGET_PAR(b110); break;
```

⟨ get macros 19 ⟩ +≡ (208)

```
#define HGET_PAR(I)
  { uint8_t n;
    if ((I) ≡ b100) { n = HGET8; REF(param_kind, n); }
    if ((I) & b100) { Xdimen x; hget_xdimen_node(&x); hwrite_xdimen(&x); }
    else HGET_REF(xdimen_kind);
    if ((I) & b010) { List l; hget_param_list(&l); hwrite_param_list(&l); }
    else if ((I) ≠ b100) HGET_REF(param_kind)
    else hwrite_ref(n);
    { List l; hget_list(&l); hwrite_list(&l); }
  }
```

5.8 Mathematics

Being able to handle mathematics nicely is one of the primary features of TEX and so you should expect the same from HINT. We start here with the more complex case—displayed equations—and finish with the simpler case of mathematical formulas that are part of the normal flow of text.

Displayed equations occur inside a paragraph node. They interrupt normal processing of the paragraph and the paragraph processing is resumed after the display. Positioning of the display depends on several parameters, the shape of the paragraph, and the length of the last line preceding the display. Displayed formulas often feature an equation number which can be placed either left or right of the formula. Also the size of the equation number will influence the placement of the formula.

In a HINT file, the parameter list is followed by a list of content nodes, representing the formula, and an optional horizontal box containing the equation number.

In the short format, we use the info bit *b100* to indicate the presence of a parameter list (which might be empty—so it's actually the absence of a reference to a parameter list); the info bit *b010* to indicate the presence of a left equation number; and the info bit *b001* for a right equation number.

In the long format, we use "`eqno`" or "`left eqno`" to indicate presence and placement of the equation number.

Reading the long format: $---\Longrightarrow$

⟨ symbols 2 ⟩ +≡ (209)
```
%token MATH "math"
%type < info > math
```

⟨ scanning rules 3 ⟩ +≡ (210)
```
math        return MATH;
```

⟨ parsing rules 5 ⟩ +≡ (211)
```
math: param_ref list { $$ = b000; }
  | param_ref list hbox_node { $$ = b001; }
  | param_ref hbox_node list { $$ = b010; }
  | param_list list { $$ = b100; }
  | param_list list hbox_node { $$ = b101; }
  | param_list hbox_node list { $$ = b110; };
content_node: start MATH math END
    { hput_tags($1, TAG(math_kind, $3)); };
```

Reading the short format: $\cdots\Longrightarrow$

⟨ cases to get content 20 ⟩ +≡ (212)
```
case TAG(math_kind, b000): HGET_MATH(b000); break;
case TAG(math_kind, b001): HGET_MATH(b001); break;
case TAG(math_kind, b010): HGET_MATH(b010); break;
case TAG(math_kind, b100): HGET_MATH(b100); break;
case TAG(math_kind, b101): HGET_MATH(b101); break;
case TAG(math_kind, b110): HGET_MATH(b110); break;
```

⟨ get macros 19 ⟩ +≡ (213)
```
#define HGET_MATH(I)
  if ((I) & b100) { List l; hget_param_list(&l); hwrite_param_list(&l); }
  else HGET_REF(param_kind);
  if ((I) & b010) hget_hbox_node();
  { List l; hget_list(&l); hwrite_list(&l); }
  if ((I) & b001) hget_hbox_node();
```

Things are much simpler if mathematical formulas are embedded in regular text. Here it is just necessary to mark the beginning and the end of the formula because glue inside a formula is not a possible point for a line break. To break the line within a formula you can insert a penalty node.

In the long format, such a simple math node just consists of the keyword “on” or “off”. In the short format, there are two info values still unassigned: we use *b011* for “off” and *b111* for “on”.

Reading the long format: $- - - \Longrightarrow$

⟨ symbols 2 ⟩ +≡ (214)
%**token** ON "on"
%**token** OFF "off"
%**type** $<i>$ *on_off*

⟨ scanning rules 3 ⟩ +≡ (215)

```
on        return ON;
off       return OFF;
```

⟨ parsing rules 5 ⟩ +≡ (216)

on_off: ON { $$ = 1; }
  | OFF { $$ = 0; };
math: *on_off* { $$ = *b011* | ($1 ≪ 2); };

Reading the short format: $\cdots \Longrightarrow$

⟨ cases to get content 20 ⟩ +≡ (217)

case TAG(*math_kind*, *b111*): *hwritef*("␣on"); **break**;
case TAG(*math_kind*, *b011*): *hwritef*("␣off"); **break**;

Note that TEX allows math nodes to specify a width using the current value of mathsurround. If this width is nonzero, it is equivalent to inserting a kern node before the math on node and after the math off node.

5.9 Adjustments

An adjustment occurs only in paragraphs. When the line breaking routine finds an adjustment, it inserts the vertical material contained in the adjustment node right after the current line. Adjustments simply contain a list node.

Reading the long format: $- - - \Longrightarrow$
Writing the short format: $\Longrightarrow \cdots$

⟨ symbols 2 ⟩ +≡ (218)
%**token** ADJUST "adjust"

⟨ scanning rules 3 ⟩ +≡ (219)

```
adjust    return ADJUST;
```

⟨ parsing rules 5 ⟩ +≡ (220)

content_node: *start* ADJUST *list* END { *hput_tags*($1, TAG(*adjust_kind*, 1)); };

Reading the short format: $\cdots \Longrightarrow$
Writing the long format: $\Longrightarrow - - -$

⟨ cases to get content 20 ⟩ +≡ (221)

case TAG(*adjust_kind*, 1): { **List** *l*; *hget_list*(&*l*); *hwrite_list*(&*l*); } **break**;

5.10 Tables

As long as a table contains no dependencies on `hsize` and `vsize`, HiTEX can expand an alignment into a set of nested horizontal and vertical boxes and no special processing is required. As long as only the size of the table itself but neither the tabskip glues nor the table content depends on `hsize` or `vsize`, the table just needs an outer node of type *hset_kind* or *vset_kind*. If there is non aligned material inside the table that depends on `hsize` or `vsize`, a vpack or hpack node is still sufficient.

While it is reasonable to restrict the tabskip glues to be ordinary glue values without `hsize` or `vsize` dependencies, it might be desirable to have content in the table that does depend on `hsize` or `vsize`. For the latter case, we need a special kind of table node. Here is why:

As soon as the dimension of an item in the table is an extended dimension, it is no longer possible to compute the maximum natural with of a column, because it is not possible to compare extended dimensions without knowing `hsize` and `vsize`. Hence the computation of maximum widths needs to be done in the viewer. After knowing the width of the columns, the setting of tabskip glues is easy to compute.

To implement these extended tables, we will need a table node that specifies a direction, either horizontal or vertical; a list of tabskip glues, with the provision that the last tabskip glue in the list is repeated as long as necessary; and a list of table content. The table's content is stacked, either vertical or horizontal, orthogonal to the alignment direction of the table. The table's content consists of nonaligned content, for example extra glue or rules, and aligned content. Each element of aligned content is called an outer item and it consist of a list of inner items. For example in a horizontal alignment, each row is an outer item and each table entry in that row is an inner item. An inner item contains a box node (of kind *hbox_kind*, *vbox_kind*, *hset_kind*, *vset_kind*, *hpack_kind*, or *vpack_kind*) followed by an optional span count.

The glue of the boxes in the inner items will be reset so that all boxes in the same column reach the same maximum column with. The span counts will be replaced by the appropriate amount of empty boxes and tabskip glues. Finally the glue in the outer item will be set to obtain the desired size of the table.

The definitions below specify just a *list* for the list of tabskip glues and a list for the outer table items. This is just for convenience; the first list must contain glue nodes and the second list must contain nonaligned content and inner item nodes.

We reuse the H and V tokens, defined as part of the specification of extended dimensions, to indicate the alignment direction of the table. To tell a reference to an extended dimension from a reference to an ordinary dimension, we prefix the former with an XDIMEN token; for the latter, the DIMEN token is optional. The scanner will recognize not only "item" as an ITEM token but also "row" and "column". This allows a more readable notation, for example by marking the outer items as rows and the inner items as columns.

In the short format, the *b010* bit is used to mark a vertical table and the *b101* bits indicate how the table size is specified; an outer item node has the info value *b000*, an inner item node with info value *b111* contains an extra byte for the span count, otherwise the info value is equal to the span count.

Reading the long format: − − − ⟹

⟨ symbols 2 ⟩ +≡ (222)

```
%token TABLE "table"
%token ITEM "item"
%type < info > table span_count
```

⟨ scanning rules 3 ⟩ +≡ (223)

```
table           return TABLE;
item            return ITEM;
row             return ITEM;
column          return ITEM;
```

⟨ parsing rules 5 ⟩ +≡ (224)

```
span_count: UNSIGNED { $$ = hput_span_count($1); };
content_node: start ITEM content_node END {
        hput_tags($1, TAG(item_kind, 1)); };
content_node: start ITEM span_count content_node END {
        hput_tags($1, TAG(item_kind, $3)); };
content_node: start ITEM list END { hput_tags($1, TAG(item_kind, b000)); };
table: H box_goal list list { $$ = $2; };
table: V box_goal list list { $$ = $2 | b010; };
content_node: start TABLE table END { hput_tags($1, TAG(table_kind, $3)); };
```

Reading the short format: · · · ⟹

⟨ cases to get content 20 ⟩ +≡ (225)

```
case TAG(table_kind, b000): HGET_TABLE(b000); break;
case TAG(table_kind, b001): HGET_TABLE(b001); break;
case TAG(table_kind, b010): HGET_TABLE(b010); break;
case TAG(table_kind, b011): HGET_TABLE(b011); break;
case TAG(table_kind, b100): HGET_TABLE(b100); break;
case TAG(table_kind, b101): HGET_TABLE(b101); break;
case TAG(table_kind, b110): HGET_TABLE(b110); break;
case TAG(table_kind, b111): HGET_TABLE(b111); break;
case TAG(item_kind, b000): { List l; hget_list(&l); hwrite_list(&l); } break;
case TAG(item_kind, b001): hget_content_node(); break;
case TAG(item_kind, b010): hwritef("␣2"); hget_content_node(); break;
case TAG(item_kind, b011): hwritef("␣3"); hget_content_node(); break;
case TAG(item_kind, b100): hwritef("␣4"); hget_content_node(); break;
case TAG(item_kind, b101): hwritef("␣5"); hget_content_node(); break;
case TAG(item_kind, b110): hwritef("␣6"); hget_content_node(); break;
```

```
  case TAG(item_kind, b111): hwritef("␣%u",HGET8); hget_content_node();
    break;
```

⟨ get macros 19 ⟩ +≡ (226)

```
#define HGET_TABLE(I)
  if (I & b010) hwritef("␣v"); else hwritef("␣h");
  if ((I) & b001) hwritef("␣add"); else hwritef("␣to");
  if ((I) & b100) { Xdimen x;
    hget_xdimen_node(&x); hwrite_xdimen_node(&x); }
  else HGET_REF(xdimen_kind)
  { List l; hget_list(&l); hwrite_list(&l); }                    /* tabskip */
  { List l; hget_list(&l); hwrite_list(&l); }                    /* items */
```

Writing the short format: ⟹ ···

⟨ put functions 14 ⟩ +≡ (227)

```
  Info hput_span_count(uint32_t n)
  {
    if (n ≡ 0) QUIT("Span␣count␣in␣item␣must␣not␣be␣zero");
    else if (n < 7) return n;
    else if (n > #FF) QUIT("Span␣count␣%d␣must␣be␣less␣than␣255",n);
    else { HPUT8(n);
      return 7;
    }
  }
```

6 Extensions

6.1 Images

In the first implementation attempt, images behaved pretty much like glue. They could stretch (or shrink) together with the surrounding glue to fill a horizontal or vertical box. While I thought this would be in line with TEX's concepts, it proved to be a bad decission because images, as opposed to glue, would stretch or shrink horizontally *and* vertically at the same time. This would require a two pass algorithm to pack boxes: first to determine the glue setting and a second pass to determine the proper image dimensions. Otherwise incorrect width or height values would propagate all the way through a sequence of nested boxes. Even worse so, this two pass algorithm would be needed in the viewer if images were contained in boxes that had extended dimensions.

The new design described below allows images with extended dimensions. This covers the case of stretchable or shrinkable images inside of extended boxes. The given extended dimensions are considered maximum values. The stretching or shrinking of images will always preserve the aspect ratio = width/height.

For convenience, we allow missing values in the long format, for example the aspect ratio, if they can be determined from the image data. In the short format, the necessary information for a correct layout must be available without using the image data.

In the long format, the only required parts of an image node are: the number of the auxiliary section where the image data can be found and the descriptive text which is there to make the document more accessible. The section number is followed by the optional aspect ratio, width, and height of the image. If some of these values are missing, it must be possible to determine them from the image data. The node ends with the description.

The short format, starts with the section number of the image data and ends with the description. Missing values for aspect ratio, width, and height are only allowed if they can be recomputed from the image data. A missing width or height is represented by a reference to the zero extended dimension. If the *b100* bit is set, the aspect ratio is present as a 32 bit floating point value followed by extended dimensions for width and height. The info value *b100* indicates a width reference followed by a height reference; the value *b111* indicates a width node followed by a height node; the value *b110* indicates a height reference followed by a width node; and the value *b101* indicates a width reference followed by a height node. The last two rules reflect the requirement that subnodes are always located at the end of a node.

The remaining info values are used as follows: The value *b000* is used for a reference to an image. The value *b011* indicates an immediate width and an immediate height. The value *b010* indicates an aspect ratio and an immediate width. The value *b001* indicates an aspect ratio and an immediate height.

The following data type stores image information. The width and height are either given as extended dimensions either directly in w and h or as references in wr and hr.

⟨ hint types 1 ⟩ +≡ (228)

typedef struct { **uint16_t** n; **float32_t** a; **Xdimen** w, h; **uint8_t** wr, hr;
} **Image**;

Reading the long format: − − − ⟹

⟨ symbols 2 ⟩ +≡ (229)

%**token** IMAGE `"image"`
%**token** `WIDTH` `"width"`
%**token** `HEIGHT` `"height"`
%**type** < *xd* > *image_width image_height*
%**type** < *f* > *image_aspect*
%**type** < *info* > *image_spec image*

⟨ scanning rules 3 ⟩ +≡ (230)

`image` **return** IMAGE;
`width` **return** `WIDTH`;
`height` **return** `HEIGHT`;

⟨ parsing rules 5 ⟩ +≡ (231)

image_aspect: *number* { $$ = $1; }
| { $$ = 0.0; };
image_width: `WIDTH` *xdimen* { $$ = $2; }
| { $$ = *xdimen_defaults*[*zero_xdimen_no*]; };
image_height: `HEIGHT` *xdimen* { $$ = $2; }
| { $$ = *xdimen_defaults*[*zero_xdimen_no*]; };
image_spec: UNSIGNED *image_aspect image_width image_height* {
$$ = *hput_image_spec*($1, $2, 0, &($3), 0, &($4)); }
| UNSIGNED *image_aspect* `WIDTH` REFERENCE *image_height* {
$$ = *hput_image_spec*($1, $2, $4, `NULL`, 0, &($5)); }
| UNSIGNED *image_aspect image_width* `HEIGHT` REFERENCE {
$$ = *hput_image_spec*($1, $2, 0, &($3), $5, `NULL`); }
| UNSIGNED *image_aspect* `WIDTH` REFERENCE `HEIGHT` REFERENCE {
$$ = *hput_image_spec*($1, $2, $4, `NULL`, $6, `NULL`); };
image: *image_spec list* { $$ = $1; };

content_node: *start* IMAGE *image* END { *hput_tags*($1, TAG(*image_kind*, $3));
};

When a short format file is generated, the image width and height must be determined if necessary from the image file. The following function will write this information into the long format file. Editing the image file at a later time and converting the short format file back to a long format file will preserve the old information. This is not allways a desirable effect. It would be possible to eliminate information about the image size when writing the long format if that information can be derived from the image file. The latter solution might have the disadvantage, that infomation about a desired image size might get lost when editing an image file.

Writing the long format: ⟹ − − −

⟨ write functions 21 ⟩ +≡ (232)

```
void hwrite_image(Image *x)
{ RNG("Section␣number", x→n, 3, max_section_no);
  hwritef("␣%u", x→n);
  if (x→a ≠ 0.0) hwrite_float64(x→a, false);
  if (x→wr ≠ 0) hwritef("␣width␣*%u", x→wr);
  else if (x→w.w ≠ 0 ∨ x→w.h ≠ 0.0 ∨ x→w.v ≠ 0.0) { hwritef("␣width");
    hwrite_xdimen(&x→w);
  }
  if (x→hr ≠ 0) hwritef("␣height␣*%u", x→hr);
  else if (x→h.w ≠ 0 ∨ x→h.h ≠ 0.0 ∨ x→h.v ≠ 0.0) { hwritef("␣height");
    hwrite_xdimen(&x→h);
  }
}
```

Reading the short format: · · · ⟹

⟨ cases to get content 20 ⟩ +≡ (233)

```
case TAG(image_kind, b001): HGET_IMAGE(b001); break;
case TAG(image_kind, b010): HGET_IMAGE(b010); break;
case TAG(image_kind, b011): HGET_IMAGE(b011); break;
case TAG(image_kind, b100): HGET_IMAGE(b100); break;
case TAG(image_kind, b101): HGET_IMAGE(b101); break;
case TAG(image_kind, b110): HGET_IMAGE(b110); break;
case TAG(image_kind, b111): HGET_IMAGE(b111); break;
```

⟨ get macros 19 ⟩ +≡ (234)

```
#define HGET_IMAGE(I)
  { Image x = {0};
    HGET16(x.n);
    if ((I) & b100) { x.a = hget_float32();
      if ((I) ≡ b111) { hget_xdimen_node(&x.w);
```

```
      hget_xdimen_node(&x.h);
    }
    else if ((I) ≡ b110) { x.hr = HGET8;
      hget_xdimen_node(&x.w);
    }
    else if ((I) ≡ b101) { x.wr = HGET8;
      hget_xdimen_node(&x.h);
    }
    else { x.wr = HGET8;
      x.hr = HGET8;
    }
  }
  else if ((I) ≡ b011) { HGET32(x.w.w);
    HGET32(x.h.w);
  }
  else if ((I) ≡ b010) { x.a = hget_float32();
    HGET32(x.w.w);
  }
  else if ((I) ≡ b001) { x.a = hget_float32();
    HGET32(x.h.w);
  }
  hwrite_image(&x);
  { List d;
    hget_list(&d);
    hwrite_list(&d);
  }
}
```

Writing the short format: ⟹ · · ·

⟨ put functions 14 ⟩ +≡ (235)

```
⟨ image functions 236 ⟩
Info hput_image_spec(uint32_t n, float32_t a, uint32_t wr, Xdimen
      *w, uint32_t hr, Xdimen *h)
{ HPUT16(n);
  if (w ≠ NULL ∧ h ≠ NULL) {
    if (w→h ≡ 0.0 ∧ w→v ≡ 0.0 ∧ h→h ≡ 0.0 ∧ h→v ≡ 0.0) return
        hput_image_dimens(n, a, w→w, h→w);
    else { hput_image_aspect(n, a);
      hput_xdimen_node(w);
      hput_xdimen_node(h);
      return b111;
    }
  }
  else if (w ≠ NULL ∧ h ≡ NULL) {
```

```
    if (w→h ≡ 0.0 ∧ w→v ≡ 0.0 ∧ hr ≡ zero_xdimen_no)
      return hput_image_dimens(n, a, w→w, 0);
    else { hput_image_aspect(n, a);
      HPUT8(hr);
      hput_xdimen_node(w);
      return b110;
    }
  }
  else if (w ≡ NULL ∧ h ≠ NULL) {
    if (wr ≡ zero_xdimen_no ∧ h→h ≡ 0.0 ∧ h→v ≡ 0.0)
      return hput_image_dimens(n, a, 0, h→w);
    else { hput_image_aspect(n, a);
      HPUT8(wr);
      hput_xdimen_node(h);
      return b101;
    }
  }
  else {
    if (wr ≡ zero_xdimen_no ∧ hr ≡ zero_xdimen_no)
      return hput_image_dimens(n, a, 0, 0);
    else { hput_image_aspect(n, a);
      HPUT8(wr);
      HPUT8(hr);
      return b100;
    }
  }
}
```

If extended dimensions are involved, the long format might very well specify different values than stored in the image. In this case the given dimensions are interpreted as maximum dimensions. If the aspect ratio is missing, we use *hextract_image_dimens* to extract it from the image file.

⟨ image functions 236 ⟩ ≡ (236)

```
static void hput_image_aspect(int n, double a)
{
  if (a ≡ 0.0) { Dimen w, h;
    hextract_image_dimens(n, &a, &w, &h);
  }
  if (a ≠ 0.0) hput_float32(a);
  else QUIT("Unable␣to␣determine␣aspect␣ratio␣of␣image␣%s",
        dir[n].file_name);
}
```

Used in 235.

If no extended dimensions are involved in an image specification, we use *hput_image_dimen*. Because the long format can omit part of the image specification, we use *hextract_image_dimens* to extract information from the image file and merge this information with the data supplied in the long format.

```
⟨ image functions 236 ⟩ +≡                                                     (237)
  ⟨ auxiliar image functions 239 ⟩
  static Info hput_image_dimens(int n, float32_t a, Dimen w, Dimen h)
  { Dimen iw, ih;
    double ia;
    if (w > 0 ∧ h > 0) { HPUT32(w);
      HPUT32(h);
      return b011;
    }
    else if (a > 0 ∧ w > 0) { hput_float32((float32_t) a);
      HPUT32(w);
      return b010;
    }
    else if (a > 0 ∧ h > 0) { hput_float32((float32_t) a);
      HPUT32(h);
      return b001;
    }
    hextract_image_dimens(n, &ia, &iw, &ih);
    ⟨ merge stored image dimensions with dimensions given 238 ⟩
    if (iw > 0) { hput_float32((float32_t) ia);
      HPUT32(iw);
      return b010;
    }
    else if (ih > 0) { hput_float32((float32_t) ia);
      HPUT32(ih);
      return b001;
    }
    else { iw = −iw;
      ih = −h;                          /* we accept the default resolution */
      HPUT32(iw);
      HPUT32(ih);
      return b011;
    }
  }
```

If the width, height or aspect ratio is stored in the image file, we can merge this information with the information given in the long format. It is considered an error, if the function *hextract_image_dimens* can not extract the aspect ratio. Absolute width and height values, however, might be missing. If the aspect ratio is computed from the number of horizontal and vertical pixels, *hextract_image_dimens* makes the reasonable assumption that the intended resolution is 72dpi and converts the image dimensions to scaled points. It negates these values to indicate that the resolution is just a guess. This allows other programs to used different default resolutions if desired.

⟨ merge stored image dimensions with dimensions given 238 ⟩ ≡ (238)

```
{
  if (ia ≡ 0.0) {
    if (a ≠ 0.0) ia = a;
    else if (w ≠ 0 ∧ h ≠ 0) ia = (double) w/(double) h;
    else QUIT("Unable␣to␣determine␣aspect␣ratio␣of␣image␣%s",
          dir[n].file_name);
  }                                  /* here the aspect ratio ia is known */
  if (w ≡ 0 ∧ h ≡ 0)                /* neither width nor height specified */
  {
    if (ih > 0) iw = round(ih * ia);
    else if (iw > 0) ih = round(iw/ia);
  }
  else if (h ≡ 0)                                      /* width specified */
  { iw = w; ih = round(w/ia); }
  else if (w ≡ 0)                                     /* height specified */
  { ih = h; iw = round(h * ia); }
  else                                                  /* both specified */
  { ih = h; iw = w; }
}
```

Used in 237.

Before we present the code to extract image dimensions from various types of image files, we define a few macros and variables for the reading these image files.

⟨ auxiliar image functions 239 ⟩ ≡ (239)

```
#define IMG_BUF_MAX 54
#define IMG_HEAD_MAX 2
  static unsigned char img_buf[IMG_BUF_MAX];
  static size_t img_buf_size;
#define LittleEndian32(X) (img_buf[(X)] + (img_buf[(X) + 1] ≪
        8) + (img_buf[(X) + 2] ≪ 16) + (img_buf[(X) + 3] ≪ 24))
#define BigEndian16(X) (img_buf[(X) + 1] + (img_buf[(X)] ≪ 8))
#define BigEndian32(X) (img_buf[(X) + 3] + (img_buf[(X) + 2] ≪
        8) + (img_buf[(X) + 1] ≪ 16) + (img_buf[(X)] ≪ 24))
#define Match2 (X, A, B) ((img_buf[(X)] ≡ (A)) ∧ (img_buf[(X) + 1] ≡ (B)))
#define Match4 (X, A, B, C, D) (Match2(X, A, B) ∧ Match2((X) + 2, C, D))
#define GET_IMG_BUF(X)
  if (img_buf_size < X) {
    size_t i = fread(img_buf + img_buf_size, 1, (X) − img_buf_size, f);
    if (i < 0) QUIT("Unable␣to␣read␣image␣%s", fn);
    else if (i ≡ 0) QUIT("Unable␣to␣read␣image␣header␣%s", fn);
    else img_buf_size += i;
  }
```

Used in 237.

Considering the different image formats, we start with Windows Bitmaps. A Windows bitmap file usually has the extension `.bmp` but the better way to check for a Windows bitmap file ist to examine the first two byte of the file: the ASCII codes for 'B' and 'M'. Once we have verified the file type, we find the width and height of the bitmap in pixels at offsets #`12` and #`16` stored as little-endian 32 bit integers. At offsets #`26` and #`2A`, we find the horizontal and vertical resolution in pixel per meter stored in the same format. This is sufficient to compute the true width and height of the image in scaled points.

The Windows Bitmap format is easy to process but not very efficient. So the support for this format in the HINT format is deprecated and will disappear. You should use one of the formats described next.

⟨ auxiliar image functions 239 ⟩ +≡ (240)

```
static bool get_BMP_info(FILE *f, char *fn, double *a, Dimen
        *w, Dimen *h)
{ double wpx, hpx;
  double xppm, yppm;
  GET_IMG_BUF(2);
  if (¬Match2(0, 'B', 'M')) return false;
  GET_IMG_BUF(#2E);
  wpx = (double) LittleEndian32(#12);                    /* width in pixel */
  hpx = (double) LittleEndian32(#16);                    /* height in pixel */
  xppm = (double) LittleEndian32(#26);       /* horizontal pixel per meter */
  yppm = (double) LittleEndian32(#2A);         /* vertical pixel per meter */
  *w = floor(0.5 + ONE * (72.00 * 1000.0/25.4) * wpx/xppm);
  *h = floor(0.5 + ONE * (72.00 * 1000.0/25.4) * hpx/yppm);
  *a = (wpx/xppm)/(hpx/yppm);
  return true;
}
```

Now we repeat this process for image files using the Portable Network Graphics file format. This file format is well suited to simple graphics that do not use color gradients. These images usually have the extension `.png` and start with an eight byte signature: #`89` followed by the ASCII Codes 'P', 'N', 'G', followd by a carriage return (#`0D` and line feed (#`0A`), an DOS end-of-file character (#`1A`) and final line feed (#`0A`). After the signature follows a list of chunks. The first chunk is the image header chunk. Each chunk starts with the size of the chunk stored as big-endian 32 bit integer, followed by the chunk name stored as four ASCII codes followed by the chunk data and a CRC. The size, as stored in the chunk, does not include the size itself, nor the name, and neither the CRC. The first chunk is the IHDR chunk. The chunk data of the IHDR chunk starts with the width and the height of the image in pixels stored as 32 bit big-endian integers.

Finding the image resolution takes some more effort. The image resolution is stored in an optional chunk named "pHYs" for the physical pixel dimensions. All we know is that this chunk, if it exists, will appear after the IHDR chunk and before the (required) IDAT chunk. The pHYs chunk contains two 32 bit big-endian integers, giving the horizontal and vertical pixels per unit, and a one byte unit specifier, which is either 0 for an undefined unit or 1 for the meter as unit. With an undefined unit, only the aspect ratio of the pixels and hence the aspect ratio of the image can be determined. It is not uncommon, however, that the resolution in such a case is given as dots per inch. So we decide to assume the latter.

If there is resolution can not be determined, we assume a resolution of 72dpi and negate width and height to inform the calling procedure of this arbitrary choice.

⟨ auxiliar image functions 239 ⟩ +≡ (241)

```
static bool get_PNG_info(FILE *f, char *fn, double *a, Dimen *w, Dimen
        *h)
{ int pos, size;
  double wpx, hpx;                    /* width and height in pixel */
  double xppu, yppu;         /* pixel per unit in x and y direction */
  int unit;
  GET_IMG_BUF(24);
  if (¬Match4(0, #89, 'P', 'N', 'G') ∨ ¬Match4(4, #0D, #0A, #1A, #0A))
    return false;
  size = BigEndian32(8);
  if (¬Match4(12, 'I', 'H', 'D', 'R')) return false;
  wpx = (double) BigEndian32(16);
  hpx = (double) BigEndian32(20);
  pos = 20 + size;
  while (true) {
    if (fseek(f, pos, SEEK_SET) ≠ 0) return false;
    img_buf_size = 0;
    GET_IMG_BUF(17);
    size = BigEndian32(0);
    if (Match4(4, 'p', 'H', 'Y', 's'))       /* must occur before IDAT chunk */
    { xppu = (double) BigEndian32(8);
      yppu = (double) BigEndian32(12);
      unit = img_buf[16];
      if (unit ≡ 0)                              /* assuming unit is inch */
      { *w = floor(0.5 + ONE * 72.27 * wpx/xppu);
        *h = floor(0.5 + ONE * 72.27 * hpx/yppu);
        *a = (wpx/xppu)/(hpx/yppu);
        return true;
      }
      else if (unit ≡ 1)                              /* unit is meter */
      { *w = floor(0.5 + ONE * (72.27/0.0254) * wpx/xppu);
        *h = floor(0.5 + ONE * (72.27/0.0254) * hpx/yppu);
        *a = (wpx/xppu)/(hpx/yppu);
        return true;
      }
      else break;
    }
```

```
    else if (Match4(4, 'I', 'D', 'A', 'T')) break;
    else pos = pos + 12 + size;
  }                             /* we assume 72dpi and negate the results */
  *w = -floor(0.5 + ONE * 72.27 * wpx/72.0);
  *h = -floor(0.5 + ONE * 72.27 * hpx/72.0);
  *a = wpx/hpx;
  return true;
}
```

For photographs, the JPEG File Interchange Format (JFIF) is more appropriate. JPEG files come with all sorts of file extensions like `.jpg`, `.jpeg`, or `.jfif`. We check the file siganture: it starts with the the SOI (Start of Image) marker #`FF`, #`D8`.

Most likely it will be followed by the JIFI-Tag. The JIFI-Tag starts with the segment marker APP0 (#`FF`, #`E0`) followed by the 2 byte segment size, followed by the ASCII codes 'J', 'F', 'I', 'F' followed by a zero byte. Next is a two byte version number which we do not read. Before the resolution proper there is a resolution unit indicator byte (0 = no units, 1 = dots per inch, 2 = dots per cm) and then comes the horizontal and vertical resolution both as 16 Bit big-endian integers.

Instead of the JIFI-Tag, there might as well be a Exif-Tag which starts with the segment marker APP1 (#`FF`, #`E1`) followed by the 2 byte segment size. Currently this tag is not decoded.

To find the actual width and height, we have to search for a start of frame marker (#`FF`, #`C0`+n with $0 \le n \le 15$). Which is followed by the 2 byte segment size, the 1 byte sample precission, the 2 byte height and the 2 byte width.

If the resolution was given explicitely in the JIFI-Tag, we use it. If there was no such tag or the uint was undefined, we proceed as we did for the PNG file.

⟨ auxiliar image functions 239 ⟩ +≡ (242)

```
  static bool get_JPG_info(FILE *f, char *fn, double *a, Dimen *w, Dimen
        *h)
  { int pos, size;
    double wpx, hpx;
    double xppu = 72.0, yppu = 72.0;
    int unit;
    GET_IMG_BUF(18);
    if (¬Match2(0, #FF, #D8))                      /* SOI Start of Image */
      return false;
    pos = 2;
    while (true) {
      if (fseek(f, pos, SEEK_SET) ≠ 0) return false;
      img_buf_size = 0;
      GET_IMG_BUF(16);
      if (img_buf[0] ≠ #FF) return false;      /* Not the start of a segment */
      if (img_buf[1] ≡ #E0 ∧ Match4(4, 'J', 'F', 'I', 'F'))  /* APP0 JFIF Tag
          */
      { unit = img_buf[11];
```

```
      xppu = (double) BigEndian16(12);
      yppu = (double) BigEndian16(14);
      if (unit ≡ 1) ;                                          /* allready in dpi */
      else if (unit ≡ 2) { xppu = xppu * 2.54; /* convert dot per cm to dpi */
        yppu = yppu * 2.54;
      }
      else { yppu = 72.0 * yppu/xppu;                             /* assume 72dpi */
        xppu = 72.0;
      }
    }
    else if (img_buf[1] ≡ #C0 ∨ img_buf[1] ≡ #C2) /* SOF Start of Frame */
    { hpx = (double) BigEndian16(5);
      wpx = (double) BigEndian16(7);
      *w = floor(0.5 + ONE * 72.27 * wpx/xppu);
      *h = floor(0.5 + ONE * 72.27 * hpx/yppu);
      *a = (wpx/xppu)/(hpx/yppu);
      return true;
    }
    else if (img_buf[1] ≡ #D9)                              /* EOI End of Image */
      return false;
    size = BigEndian16(2);
    pos = pos + 2 + size;
  }
  return false;
}
```

There is still one image format missing: scalable vector graphics. In the moment, I tend not to include a further image format into the definition of the HINT file format but instead use the PostScript subset that is used for Type 1 fonts to encode vector graphics. Any HINT viewer must support Type 1 PostScript fonts and hence it has already the necessary interpreter. So it seems reasonable to put the burden of converting vector graphics into a Type 1 PostScript font on the generator of HINT files and keep the HINT viewer as small and simple as possible. An alternative which would impose only a slight burden on the HINT file viewer is the use of the rsvg library.

After having considered the various types of image files, we now determine width, height and aspect ratio based on such an image file.

We combine all the above functions into the *hextract_image_dimens* function.

⟨ image functions 236 ⟩ +≡ (243)

```
void hextract_image_dimens(int n, double *a, Dimen *w, Dimen *h)
{ char *fn;
  FILE *f;

  *a = 0.0;
  *w = *h = 0;
  fn = dir[n].file_name;
  f = fopen(fn, "rb");
```

```
  if (f ≠ NULL) { img_buf_size = 0;
    if (¬get_BMP_info(f, fn, a, w,
          h) ∧ ¬get_PNG_info(f, fn, a, w, h) ∧ ¬get_JPG_info(f, fn, a, w, h))
      DBG(DBGDEF, "Unknown␣image␣type␣%s", fn);
    fclose(f);
    DBG(DBGDEF, "image␣%d:␣width=␣%fpt␣height=␣%fpt␣aspect=%f\n", n,
        *w/(double) ONE, *h/(double) ONE, *a);
  }
}
```

6.2 Positions, Outlines, Links, and Labels

A viewer can usually not display the entire content section of a HINT file. Instead it will display a page of content and will give its user various means to change the page. This might be as simple as a "page down" or "page up" button (or gesture) and as sophisticated as searching using regular expressions. More traditional ways to navigate the content include the use of a table of content or an index of keywords. All these methods of changing a page have in common that a part of the content that fits nicely in the screen area provided by the output device must be rendered given a position inside the content section.

Let's assume that the viewer uses a HINT file in short format—after all that's the format designed for precisely this use. A position inside the content section is then the position of the starting byte of a node. Such a position can be stored as a 32 bit number. Because even the smallest node contains two tag bytes, the position of any node is strictly smaller than the maximum 32 bit number which we can conveniently use as a "non position".

⟨ hint macros 13 ⟩ +≡ (244)

```
#define HINT_NO_POS #FFFFFFFF
```

To render a page starting at a given position is not difficult: We just read content nodes, starting at the given position and feed them to TEX's page builder until the page is complete. To implement a "clickable" table of content this is good enough. We store with every entry in the table of content the position of the section header, and when the user clicks the entry, the viewer can display a new page starting exactly with that section header.

Things are slightly more complex if we want to implement a "page down" button. If we press this button, we want the next page to start exactly where the current page has ended. This is typically in the middle of a paragraph node, and it might even be in the middle of an hyphenated word in that paragraph. Fortunately, paragraph and table nodes are the only nodes that can be broken across page boundaries. But broken paragraph nodes are a common case non the less, and unless we want to search for the enclosing node, we need to augment in this case the primary 32 bit position inside the content section with a secondary position. Most of the time, 16 bit will suffice for this secondary position if we give it relative to the primary position. Further, if the list of nodes forming the paragraph is given as a text, we need to know the current font at the secondary position. Of course, the viewer can find it by scanning the initial part of the text, but when we think of a page down button, the viewer might already know it from rendering the previous page.

Similar is the case of a "page up" button. Only here we need a page that ends precisely where our current page starts. Possibly even with the initial part of a hyphenated word. Here we need a reverse version of TEX's page builder that assembles a "good" page from the bottom up instead of from the top down. Sure the viewer can cache the start position of the previous page (or the rendering of the entire page) if the reader has reached the current page using the page down button. But this is not possible in all cases. The reader might have reached the current page using the table of content or even an index or a search form.

This is the most complex case to consider: a link from an index or a search form to the position of a keyword in the main text. Let's assume someone looks up the word "München". Should the viewer then generate a page that starts in the middle of a sentence with the word "München"? Probably not! We want a page that shows at least the whole sentence if not the whole paragraph. Of course the program that generates the link could specify the position of the start of the paragraph instead of the position of the word. But that will not solve the problem. Just imagine reading the groundbreaking masterpiece of a German philosopher on a small hand-held device: the paragraph will most likely be very long and perhaps only part of the first sentence will fit on the small screen. So the desired keyword might not be found on the page that starts with the beginning of the paragraph; it might not even be on the next or next to next page. Only the viewer can decide what is the best fragment of content to display around the position of the given keyword.

To summarize, we need three different ways to render a page for a given position:

- A page that starts exactly at the given position.
- A page that ends exactly at the given position.
- The "best" page that contains the given position somewhere in the middle.

A possible way to find the "best" page for the latter case could be the following:

- If the position is inside a paragraph, break the paragraph into lines. One line will contain the given position. Let's call this the destination line.
- If the paragraph will not fit entirely on the page, start the page with the beginning of the paragraph if that will place the destination line on the page, otherwise start with a line in the paragraph that is about half a page before the destination line.
- Else traverse the content list backward for about 2/3 of the page height and forward for about 2/3 of the page height, searching for the smallest negative penalty node. Use the penalty node found as either the beginning or ending of the page.
- If there are several equally low negative penalty nodes. Prefer penalties preceding the destination line over penalty nodes following it. A good page start is more important than a good page end.
- If there are are still several equally low negative penalty nodes, choose the one whose distance to the destination line is closest to 1/2 of the page height.

- If no negative penalty nodes could be found, start the page with the paragraph containing the destination line.
- Once the page start (or end) is found, use TeX's page builder (or its reverse variant) to complete the page.

We call content nodes that reference some position inside the content section "link" nodes. The position that is referenced is called the destination of the link. Link nodes occur always in pairs of an "start" link followed by a corresponding "end" link that both reference the same position and no other link nodes between them. The content between the two will constitute the visible part of the link.

To encode a position inside the content section that can be used as the destination of a link node, an other kind of node is needed which we call a "label".

Links are not the only way to navigate inside a large document. The user interface can also present an "outline" of the document that can be used for navigation. An outline node implements an association between a name displayed by the user interface of the HINT viewer and the destination position in the HINT document.

It is possible though that outline nodes, link nodes, and label nodes can share the same kind-value and we have *outline_kind* $\equiv$ *link_kind* $\equiv$ *label_kind*. To distinguish an outline node from a label node—both occur in the short format definition section—the *b100* info bit is set in an outline node.

⟨ get functions 18 ⟩ +≡ (245)

void *hget_outline_or_label_def* (**Info** *i*, **uint32_t** *node_pos*)
{ **if** (*i* & *b100*) ⟨ get and write an outline node 277 ⟩
  else ⟨ get and store a label node 261 ⟩
}

The next thing we need to implement is a new maximum number for outline nodes. We store this number in the variable *max_outline* and limit it to a 16 bit value.

In the short format, the value of *max_outline* is stored with the other maximum values using the kind value *outline_kind* $\equiv$ *label_kind* and the info value *b100* for single byte and *b101* for a two byte value.

Reading the Short Format: $\cdots \Longrightarrow$

⟨ cases of getting special maximum values 246 ⟩ ≡ (246)

case `TAG`(*outline_kind*, *b100*): **case** `TAG`(*outline_kind*, *b101*): *max_outline* = *n*;
  `DBG`(`DBGDEF` | `DBGLABEL`, `"max(outline)␣=␣%d\n"`, *max_outline*);
  break;

Used in 388.

Writing the Short Format: ⟹ ···

⟨ cases of putting special maximum values 247 ⟩ ≡ (247)

```
if (max_outline > −1) { uint32_t pos = hpos++ − hstart;
   DBG(DBGDEF | DBGLABEL, "max(outline)␣=␣%d\n", max_outline);
   hput_tags(pos, TAG(outline_kind, b100 | (hput_n(max_outline) − 1)));
}
```

Used in 389.

Writing the Long Format: ⟹ − − −

⟨ cases of writing special maximum values 248 ⟩ ≡ (248)

```
case label_kind:
   if (max_ref[label_kind] > −1)
   { hwrite_start();
      hwritef("label␣%d", max_ref[label_kind]);
      hwrite_end(); }
   if (max_outline > −1)
   { hwrite_start();
      hwritef("outline␣%d", max_outline);
      hwrite_end(); }
   break;
```

Used in 387.

Reading the Long Format: − − − ⟹

⟨ parsing rules 5 ⟩ +≡ (249)

```
max_value: OUTLINE UNSIGNED { max_outline = $2;
      RNG("max␣outline", max_outline, 0, #FFFF); DBG(DBGDEF | DBGLABEL,
         "Setting␣max␣outline␣to␣%d\n", max_outline); };
```

After having seen the maximum values, we now explain labels, then links, and finally outlines.

To store labels, we define a data type *Label* and an array *labels* indexed by the labels reference number.

⟨ hint basic types 6 ⟩ +≡ (250)

```
typedef struct { uint32_t pos;                  /* position */
   uint32_t pos0;                               /* secondary position */
   uint8_t where;                               /* where on the rendered page */
   bool used;                                   /* label used in a link or an outline */
   int next;                                    /* reference in a linked list */
   uint8_t f;                                   /* font, currently not used */
} Label;
```

The *where* field indicates where the label position should be on the rendered page: at the top, at the bottom, or somewhere in the middle. An undefined label has *where* equal to zero.

⟨ hint macros 13 ⟩ +≡ (251)

```
#define LABEL_UNDEF 0
#define LABEL_TOP 1
#define LABEL_BOT 2
#define LABEL_MID 3
```

⟨ common variables 252 ⟩ ≡ (252)

```
  Label *labels = NULL;
  int first_label = −1;
```

Used in 549, 551, 554, 555, and 557.

The variable *first_label* will be used together with the *next* field of a label to construct a linked list of labels.

⟨ initialize definitions 253 ⟩ ≡ (253)

```
  if (max_ref[label_kind] ≥ 0)
    ALLOCATE(labels, max_ref[label_kind] + 1, Label);
```

Used in 379 and 385.

The implementation of labels has to solve the problem of forward links: a link node that references a label that is not yet defined. We solve this problem by keeping all labels in the definition section. So for every label at least a definition is available before we start with the content section and we can fill in the position when the label is found. If we restrict labels to the definition section and do not have an alternative representation, the number of possible references is a hard limit on the number of labels in a document. Therefore label references are allowed to use 16 bit reference numbers. In the short format, the *b001* bit indicates a two byte reference number if set, and a one byte reference number otherwise.

In the short format, the complete information about a label is in the definition section. In the long format, this is not possible because we do not have node positions. Therefore we will put label nodes at appropriate points in the content section and compute the label position when writing the short format.

Reading the long format: − − − ⟹

⟨ symbols 2 ⟩ +≡ (254)

```
%token LABEL "label"
%token BOT "bot"
%token MID "mid"
%type <i> placement
```

⟨ scanning rules 3 ⟩ +≡ (255)

```
label     return LABEL;
bot       return BOT;
mid       return MID;
```

A label node specifies the reference number and a placement.

⟨ parsing rules 5 ⟩ +≡ (256)

```
placement: TOP { $$ = LABEL_TOP; }
    | BOT { $$ = LABEL_BOT; }
    | MID { $$ = LABEL_MID; }
    | { $$ = LABEL_MID; };
content_node: START LABEL REFERENCE placement END
        { hset_label($3, $4); };
```

After parsing a label, the function *hset_label* is called.

⟨ put functions 14 ⟩ +≡ (257)

```
void hset_label(int n, int w)
{ Label *t;
  REF_RNG(label_kind, n);
  t = labels + n;
  if (t→where ≠ LABEL_UNDEF)
    MESSAGE("Duplicate␣definition␣of␣label␣%d\n", n);
  t→where = w;
  t→pos = hpos − hstart;
  t→pos0 = hpos0 − hstart;
  t→next = first_label;
  first_label = n;
}
```

The above function will simply store the data obtained in the *labels* array. The generation of the short format output is postponed until the entire content section has been parsed and the positions of all labels are known.

One more complication needs to be considered: The *hput_list* function is allowed to move lists in the output stream and if positions inside the list were recorded in a label, these labels need an adjustment. To find out quickly if any labels are affected, the *hset_label* function constructs a linked list of labels starting with the reference number of the most recent label in *first_label* and the reference number of the label preceding label i in *labels*[i].*next*. Because labels are recorded with increasing positions, the list will be sorted with positions decreasing.

⟨ adjust label positions after moving a list 258 ⟩ ≡ (258)

```
{ int i;
  for (i = first_label; i ≥ 0 ∧ labels[i].pos ≥ l→p; i = labels[i].next) {
    DBG(DBGNODE | DBGLABEL, "Moving␣label␣*%d␣by␣%d\n", i, d);
    labels[i].pos += d;
    if (labels[i].pos0 ≥ l→p) labels[i].pos0 += d;
  }
}
```

Used in 148.

The *hwrite_label* function is the reverse of the above parsing rule. Note that it is different from the usual *hwrite_*... functions. And we will see shortly why that is so.

Writing the long format: $\Longrightarrow$ − − −

⟨ write functions 21 ⟩ +≡ (259)

```
void hwrite_label(void)    /* called in hwrite_end and at the start of a list */
{ while (first_label ≥ 0 ∧ (uint32_t)(hpos − hstart) ≥ labels[first_label].pos)
  { Label *t = labels + first_label;
    DBG(DBGLABEL, "Inserting␣label␣*%d\n", first_label);
    hwrite_start( );
    hwritef("label␣*%d", first_label);
    if (t→where ≡ LABEL_TOP) hwritef("␣top");
    else if (t→where ≡ LABEL_BOT) hwritef("␣bot");
    nesting−−;
    hwritec('>');                 /* avoid a recursive call to hwrite_end */
    first_label = labels[first_label].next;
  }
}
```

The short format specifies the label positions in the definition section. This is not possible in the long format because there are no “positions” in the long format. Therefore long format label nodes must be inserted in the content section just before those nodes that should come after the label. The function *hwrite_label* is called in *hwrite_end*. At that point *hpos* is the position of the next node and it can be compared with the positions of the labels taken from the definition section. Because *hpos* is strictly increasing while reading the content section, the comparison can be made efficient by sorting the labels. The sorting uses the *next* field in the array of *labels* to construct a linked list. After sorting, the value of *first_label* is the index of the label with the smallest position; and for each i, the value of *labels*[i].*next* is the index of the label with the next bigger position. If *labels*[i].*next* is negative, there is no next bigger position. Currently a simple insertion sort is used. The insertion sort will work well if the labels are already mostly in ascending order. If we expect lots of labels in random order, a more sophisticated sorting algorithm might be appropriate.

⟨ write functions 21 ⟩ +≡ (260)

```
void hsort_labels(void)
{ int i;
  if (max_ref[label_kind] < 0) { first_label = −1;
    return; }                                   /* empty list */
  first_label = max_ref[label_kind];
  while (first_label ≥ 0 ∧ labels[first_label].where ≡ LABEL_UNDEF)
    first_label−−;
  if (first_label < 0) return;                  /* no defined labels */
  labels[first_label].next = −1;
  DBG(DBGLABEL, "Sorting␣%d␣labels\n", first_label + 1);
  for (i = first_label − 1; i ≥ 0; i−−)         /* insert label i */
    if (labels[i].where ≠ LABEL_UNDEF)
    { uint32_t pos = labels[i].pos;
```

```
      if (labels[first_label].pos ≥ pos)
      { labels[i].next = first_label;
        first_label = i; }                                    /* new smallest */
      else
      { int j;
        for (j = first_label; labels[j].next ≥ 0 ∧ labels[labels[j].next].pos < pos;
              j = labels[j].next) continue;
        labels[i].next = labels[j].next;
        labels[j].next = i;
      }
    }
}
```

The following code is used to get label information from the definition section and store it in the *labels* array. The *b010* bit indicates the presence of a secondary position for the label.

Reading the short format: ··· ⟹

⟨ get and store a label node 261 ⟩ ≡ (261)

```
{ Label *t;
  int n;
  if (i & b001) HGET16(n); else n = HGET8;
  REF_RNG(label_kind, n);
  t = labels + n;
  if (t→where ≠ LABEL_UNDEF) DBG(DBGLABEL,
        "Duplicate␣definition␣of␣label␣%d␣at␣0x%x\n", n, node_pos);
  HGET32(t→pos);
  t→where = HGET8;
  if (t→where ≡ LABEL_UNDEF ∨ t→where > LABEL_MID)
    DBG(DBGLABEL, "Label␣%d␣where␣value␣invalid:␣%d␣at␣0x%x\n", n,
        t→where, node_pos);
  if (i & b010)                                    /* secondary position */
  { HGET32(t→pos0);
    t→f = HGET8; }
  else t→pos0 = t→pos;
  DBG(DBGLABEL, "Defining␣label␣%d␣at␣0x%x/0x%x\n", n, t→pos0, t→pos);
}
```

Used in 245.

The function *hput_label* is simply the reverse of the above code.

Writing the short format: ⟹ ···

⟨ put functions 14 ⟩ +≡ (262)

```
Tag hput_label(int n, Label *l)
{ Info i = b000;
  HPUTX(13);
  if (n > #FF) { i |= b001;
    HPUT16(n); } else HPUT8(n);
  HPUT32(l→pos);
  HPUT8(l→where);
  if (l→pos ≠ l→pos0) { i |= b010;
    HPUT32(l→pos0);
    HPUT8(l→f); }
  DBG(DBGLABEL, "Defining␣label␣%d␣at␣0x%x/0x%x\n", n, l→pos0, l→pos);
  return TAG(label_kind, i);
}
```

hput_label_defs is called by the parser after the entire content section has been processed; it appends the label definitions to the definition section. The outlines are stored after the labels because they reference the labels.

⟨ put functions 14 ⟩ +≡ (263)

```
extern void hput_definitions_end(void);
extern Tag hput_outline(Outline *t);

void hput_label_defs(void)
{ int n;
  section_no = 1;
  hstart = dir[1].buffer;
  hend = hstart + dir[1].bsize;
  hpos = hstart + dir[1].size;
  ⟨ output the label definitions 264 ⟩
  ⟨ output the outline definitions 284 ⟩
  hput_definitions_end();
}
```

⟨ output the label definitions 264 ⟩ ≡ (264)

```
for (n = 0; n ≤ max_ref[label_kind]; n++)
{ Label *l = labels + n;
  uint32_t pos;
  if (l→used)
  { pos = hpos++ − hstart;
    hput_tags(pos, hput_label(n, l));
    if (l→where ≡ LABEL_UNDEF)
      MESSAGE("WARNING:␣Label␣*%d␣is␣used␣but␣not␣defined\n", n);
    else DBG(DBGDEF | DBGLABEL, "Label␣*%d␣defined␣0x%x\n", n, pos);
  }
  else {
```

```
      if (l→where ≠ LABEL_UNDEF) { pos = hpos++ − hstart;
        hput_tags(pos, hput_label(n, l));
        DBG(DBGDEF | DBGLABEL, "Label␣*%d␣defined␣but␣not␣used␣0x%x\n",
            n, pos);
      }
    }
  }
```

Used in 263.

Links are simpler than labels. They are found only in the content section and resemble pretty much what we have seen for other content nodes. Let's look at them next. When reading a short format link node, we use again the *b001* info bit to indicate a 16 bit reference number to a label.

To help a reader tell a link from ordinary text, links should be visualy different. This is supported in the HINT file format by associating a different color scheme to a link. In the short format, the *b100* bit indicates that a color set reference (see section 6.3) follows after the label reference. A color reference to 1 in the start node and to #FF in the end node is the default and is omitted.

Because color changes are local to the enclosing box or paragraph, a link is local as well. Without further mentioning, here and in the following, when we say "box" it also mean "paragraph". A link starts with a "start" link and ends with either an "end" link or the end of the enclosing box. Links must not be nested. It is an error to have two start links in the same box without an end link between them. An application can choose to continue a link in the next box by inserting a copy of the start link node at the begining of the new box. In short: "end" links are mandatory when separating two links but optional if they just preceede the end of the box. The *b010* info bit indicates a "start" link; otherwise it is an "end" link.

Reading the short format: · · · ⟹

⟨ get macros 19 ⟩ +≡ (265)

```
#define HGET_LINK(I)
  { int n, c;
    if (I & b001) HGET16(n); else n = HGET8;
    if (I & b100) c = HGET8;
    else c = (I & b010) ? 1 : #FF;
    hwrite_link(n, c, I & b010); }
```

⟨ cases to get content 20 ⟩ +≡ (266)

```
  case TAG(link_kind, b000): HGET_LINK(b000); break;
  case TAG(link_kind, b001): HGET_LINK(b001); break;
  case TAG(link_kind, b010): HGET_LINK(b010); break;
  case TAG(link_kind, b011): HGET_LINK(b011); break;
  case TAG(link_kind, b100): HGET_LINK(b100); break;
  case TAG(link_kind, b101): HGET_LINK(b101); break;
  case TAG(link_kind, b110): HGET_LINK(b110); break;
  case TAG(link_kind, b111): HGET_LINK(b111); break;
```

The function *hput_link* will insert the link in the output stream and return the appropriate tag.

Writing the short format: $\Longrightarrow \cdots$

⟨ put functions 14 ⟩ +≡ (267)

```
Tag hput_link(int n, int c, int on)
{ Info i;
  REF_RNG(label_kind, n);
  labels[n].used = true;
  if (on) i = b010; else i = b000;
  if (n > #FF) { i |= b001;
    HPUT16(n); } else HPUT8(n);
  if ((on ∧ c ≠ 1) ∨ (¬on ∧ c ≠ #FF)) { i |= b100;
    HPUT8(c);
  }
  return TAG(link_kind, i);
}
```

Reading the long format: $- - - \Longrightarrow$

⟨ symbols 2 ⟩ +≡ (268)

```
%token LINK "link"
```

⟨ scanning rules 3 ⟩ +≡ (269)

```
link            return LINK;
```

⟨ parsing rules 5 ⟩ +≡ (270)

```
content_node: start LINK REFERENCE on_off END {
        hput_tags($1, hput_link($3, $4 ? 1 : #FF, $4)); };
content_node: start LINK REFERENCE on_off REFERENCE END {
        hput_tags($1, hput_link($3, $5, $4)); };
```

Writing the long format: $\Longrightarrow - - -$

⟨ write functions 21 ⟩ +≡ (271)

```
void hwrite_link(int n, int c, uint8_t on)
{ REF_RNG(label_kind, n);
  if (labels[n].where ≡ LABEL_UNDEF)
    MESSAGE("WARNING:␣Link␣to␣an␣undefined␣label␣%d\n", n);
  hwrite_ref(n);
  if (on) hwritef("␣on");
  else hwritef("␣off");
  if ((on ∧ c ≠ 1) ∨ (¬on ∧ c ≠ #FF)) { REF_RNG(color_kind, c);
    hwrite_ref(c);
  }
}
```

Now we look at the outline nodes which are found only in the definition section. Every outline node is associated with a label node, giving the position in the document, and a unique title that should tell the user what to expect when navigating to this position. For example an item with the title "Table of Content" should navigate to the page that shows the table of content. The sequence of outline nodes found in the definition section gets a tree structure by assigning to each item a depth level.

⟨ hint types 1 ⟩ +≡ (272)

```
typedef struct { uint8_t *t;    /* title */
  int s;                        /* title size */
  int d;                        /* depth */
  uint16_t r;                   /* reference to a label */
} Outline;
```

⟨ shared put variables 273 ⟩ ≡ (273)

```
Outline *outlines;
```

Used in 551, 554, 555, and 557.

⟨ initialize definitions 253 ⟩ +≡ (274)

```
if (max_outline ≥ 0)
  ALLOCATE(outlines, max_outline + 1, Outline);
```

Child items follow their parent item and have a bigger depth level. In the short format, the first item must be a root item, with a depth level of 0. Further, if any item has the depth d, then the item following it must have either the same depth d in which case it is a sibling, or the depth $d+1$ in which case it is a child, or a depth d' with $0 \le d' < d$ in which case it is a sibling of the latest ancestor with depth d'. Because the depth is stored in a single byte, the maximum depth is #FF.

In the long format, the depth assignments are more flexible. We allow any signed integer, but insist that the depth assignments can be compressed to depth levels for the short format using the following algorithm:

⟨ compress long format depth levels 275 ⟩ ≡ (275)

```
n = 0; while (n ≤ max_outline) n = hcompress_depth(n, 0);
```

Used in 284.

Outline items must be listed in the order in which they should be displayed. The function *hcompress_depth*(n, c) will compress the subtree starting at n with root level d to a new tree with the same structure and root level c. It returns the outline number of the following subtree.

⟨ put functions 14 ⟩ +≡ (276)

```
int hcompress_depth(int n, int c)
{ int d = outlines[n].d;
  if (c > #FF)
    QUIT("Outline␣%d,␣depth␣level␣%d␣to␣%d␣out␣of␣range", n, d, c);
  while (n ≤ max_outline)
    if (outlines[n].d ≡ d) outlines[n++].d = c;
```

```
      else if (outlines[n].d > d) n = hcompress_depth(n, c + 1);
      else break;
    return n;
  }
```

For an outline node, the *b001* bit indicates a two byte reference to a label. There is no reference number for an outline item itself: it is never referenced anywhere in an HINT file.

Reading the short format: $\cdots \Longrightarrow$
Writing the long format: $\Longrightarrow ---$

⟨ get and write an outline node 277 ⟩ ≡ (277)

```
  { int r, d;
    List l;
    static int outline_no = −1;
    hwrite_start( ); hwritef("outline");
    ++outline_no;
    RNG("outline", outline_no, 0, max_outline);
    if (i & b001) HGET16(r); else r = HGET8;
    REF_RNG(link_kind, r);
    if (labels[r].where ≡ LABEL_UNDEF)
      MESSAGE("WARNING:␣Outline␣with␣undefined␣label␣%d␣at␣0x%x\n",
          r, node_pos);
    hwritef("␣*%d", r);
    d = HGET8;
    hwritef("␣%d", d);
    hget_list(&l);
    hwrite_list(&l);
    hwrite_end( );
  }
```

Used in 245.

When parsing an outline definition in the long format, we parse the outline title as a *list* which will write the representation of the list to the output stream. Writing the outline definitions, however, must be postponed until the label have found their way into the definition section. So we save the list's representation in the outline node for later use and remove it again from the output stream.

Reading the long format: − − − ⟹

⟨ symbols 2 ⟩ +≡ (278)

```
%token OUTLINE "outline"
```

⟨ scanning rules 3 ⟩ +≡ (279)

```
outline        return OUTLINE;
```

⟨ parsing rules 5 ⟩ +≡ (280)

```
def_node: START OUTLINE REFERENCE integer position list END { static
          int outline_no = −1;
      $$.k = outline_kind; $$.n = $3;
      if ($6.s ≡ 0)
        QUIT("Outline␣with␣empty␣title␣in␣line␣%d", yylineno);
      outline_no++; hset_outline(outline_no, $3, $4, $5); };
```

⟨ put functions 14 ⟩ +≡ (281)

```
void hset_outline(int m, int r, int d, uint32_t pos)
{ Outline *t;
  RNG("Outline", m, 0, max_outline);
  t = outlines + m;
  REF_RNG(label_kind, r);
  t→r = r;
  t→d = d;
  t→s = hpos − (hstart + pos);
  hpos = (hstart + pos);
  ALLOCATE(t→t, t→s, uint8_t);
  memmove(t→t, hpos, t→s);
  labels[r].used = true;
}
```

To output the title, we need to move the list back to the output stream. Before doing so, we allocate space (and make sure there is room left for the end tag of the outline node), and after doing so, we release the memory used to save the title.

⟨ output the title of outline *t 282 ⟩ ≡ (282)

```
memmove(hpos, t→t, t→s);
hpos = hpos + t→s;
free(t→t);
```

Used in 283.

We output all outline definitions from 0 to *max_outline* and check that every one of them has a title. Thereby we make sure that in the short format *max_outline* matches the number of outline definitions.

Writing the short format: $\Longrightarrow \cdots$

⟨ put functions 14 ⟩ +≡ (283)

```
Tag hput_outline(Outline *t)
{ Info i = b100;
  HPUTX(t→s + 4);
  if (t→r > #FF) { i |= b001; HPUT16(t→r); } else HPUT8(t→r);
  labels[t→r].used = true;
  HPUT8(t→d);
  ⟨ output the title of outline *t 282 ⟩
  return TAG(outline_kind, i);
}
```

⟨ output the outline definitions 284 ⟩ ≡ (284)

```
⟨ compress long format depth levels 275 ⟩
for (n = 0; n ≤ max_outline; n++) { Outline *t = outlines + n;
  uint32_t pos;
  pos = hpos++ − hstart;
  if (t→s ≡ 0 ∨ t→t ≡ NULL)
    QUIT("Definition␣of␣outline␣%d␣has␣an␣empty␣title", n);
  DBG(DBGDEF | DBGLABEL, "Outline␣*%d␣defined\n", n);
  hput_tags(pos, hput_outline(t));
}
```

Used in 263.

6.3 Colors

This is the third draft of implementing color specifications in a HINT file.

According to the initial philosophy of a HINT file, a viewer must be capable of rendering a page given just any valid position in the content section without reading the entire file. This makes it impossible to use global information; only the information that is localy available can be used. Given a file position, the viewer will compute a representation of the page, insert it into a page template, and pass it to the renderer. Color will not effect the position of glyphs or rules and so it is sufficient to process the color information when rendering the page. The renderer will, however, render the page always from the top down and from left to right. As a consequence of the rendering order, it is very well possible to work with a color state within the top level boxes.

A separate issue is the specification of color changes on the top level. While a vertical list contains no character nodes, a color specification might still affect the background color and the foreground color of rules. Because we still want to avoid the search for color nodes on the top level, we restrict the scope of a color node on the top level. It will extend only to the next possible page break and applications like HiTEX must repeat a top level color node after every node that could be used as a page break.

The nesting of boxes on a page together with the transparency of colors leads to the problem of stacking several layers of color one on top of the other. Here is an example: An outer box might specify blue as a background color and white as a forground color while an inside box specifies a transparent grey background and a transparent black foreground. Then we expect text in the outer box to have white letters on blue background. Further we want to see the inner box casting a grey shadown on the blue background, resulting in a mix of blue and gray, with black letters on top of it that are not completely black but let the background shine through.

To limit the complexity, the HINT file format will allow this stacking of colors only when nesting boxes. But inside a box, there is at any position only one foreground and one background color; a color change inside a box will simply replace the current colors.

If an application like HiTEX wants to implement nesting colors inside a box, it has to implement its own color stack and compute the necessary color mixtures. There is only one exception to this concept: When a new box starts, the current colors will be those of the enclosing box. These colors can be restored after a color change by using the ⟨color off⟩ command.

The limited complexity is necessary to simplifies the spliting of boxes, for example by the line breaking routine. Repeating the last color node before the split just after the split is sufficient.

Inside a horizontal list, a background color will extend from top to bottom; inside a vertical list a background color will extend from the left edge to the right edge. If the document does not want to change the background color, a completely transparent color should be used.

While the current implementation of searching does not use the background color, a color set will still specify background and foreground for all colors. This is simpler, easier to extend at a later time, and the overhead is small.

After these preliminaries let's turn our attention to the design of a suitable color concept.

Colors come in sets. A color set supports two modes: day and night mode. In future extensions it might be possible for an author to invent color sets for winter or summer, fall or spring, or any other resonable or unreasonable purpose. For each mode a color set specifies three color styles: one for normal text, one for marked text and one for in-focus text. The switching between different modes and different styles is left to the user interface.

We store a color set as an array of 12 words. The first 6 words are for day mode the next 6 byte are for night mode; For each mode we have three color pairs and each pair consists of a forgraound and a backgraund color each stored as an RGBA value.

⟨ hint basic types 6 ⟩ +≡ (285)

typedef uint32_t ColorSet$[2*3*2]$;

To extract the various sub-arrays, we have the following macros:

⟨ hint macros 13 ⟩ +≡ (286)

#**define** `CURCOLOR` (M,S,C) $((C)+6*(M)+2*(S))$
#**define** `DAY` (C) `CURCOLOR` $(0,0,C)$
#**define** `NIGHT` (C) `CURCOLOR` $(1,0,C)$

```
#define HIGH (C) CURCOLOR (0,1,C)
#define FOCUS (C) CURCOLOR (0,2,C)
#define FG (C) ((C)[0])
#define BG (C) ((C)[1])
```

We will allow up to 255 color sets that are stored in the definition section and are referenced in the content section by a single byte. The definition of different color sets and the switching between them is left to the document author.

The color set with reference number zero specifies the default colors. At the root of a page template, the default color set is selected and the whole page is filled with the background color for normal text. For links, by default the color set with number one is used. Section 11.11 specifies default values for both color sets; the default colors can be overwritten. The color sets with reference numbers zero and one are not stored in the definition section of a short format file if they are the same as the default values. This makes files not using colors compatible with older versions of the HINT file format.

Now we are ready for the implementation.

Reading the long format: − − − ⟹
Writing the short format: ⟹ · · ·

⟨symbols 2⟩ +≡ (287)
```
%token COLOR "color"
```

⟨scanning rules 3⟩ +≡ (288)
```
color          return COLOR;
```

Colors can be specified as a single number, preferably in hexadecimal notation, giving the red, green, blue, and alpha value in a single number. For example `0xFF0000FF` would be pure red, and `0x00FF0080` would be transparent green. Of course even decimal values can be used. A good example is the value `0` which is equivalent to but a bit shorter than `0x0` or `0x00000000` which describes a completely transparent black. It is invisible because the alpha value is zero.

Alternatively, colors can be given as a list of three or four numbers enclosed in pointed brackets `<` ... `>`. If only three numbers are given, the color is opaque with an alpha value equivalent to `0xFF`. Using this format the same colors as before can be written `<0xFF 0 0>` (pure red), `<0 0xFF 0 0x80>` (transparent green) and `<0 0 0 0>` (transparent black).

The parser will put the color definition into *colors_n* using the index *colors_i*. As we will see later, the *colors_n* array is initialized with the colors in *colors_0* which in turn is initialized from *color_defaults*[0]. *colors_0* can be changed but only if that change occurs before any other color definition.

⟨common variables 252⟩ +≡ (289)
```
ColorSet colors_0, colors_n;   /* default and current color set */
int colors_i;                  /* current color */
```

⟨initialize definitions 253⟩ +≡ (290)
```
{ int i;
```

```
    for (i = 0; i < sizeof(ColorSet)/4; i++) colors_0[i] = color_defaults[0][i];
  }
```

⟨ parsing rules 5 ⟩ +≡ (291)

```
color: START UNSIGNED UNSIGNED UNSIGNED UNSIGNED END
      { RNG("red", $2, 0, #FF); RNG("green", $3, 0, #FF);
      RNG("blue", $4, 0, #FF); RNG("alpha", $5, 0, #FF);
      colors_n[colors_i++] = ($2 << 24) | ($3 << 16) | ($4 << 8) | $5; }
   | START UNSIGNED UNSIGNED UNSIGNED END { RNG("red", $2, 0, #FF);
      RNG("green", $3, 0, #FF); RNG("blue", $4, 0, #FF);
      colors_n[colors_i++] = ($2 << 24) | ($3 << 16) | ($4 << 8) | #FF; };
color: UNSIGNED { colors_n[colors_i++] = $1; };
```

Colors are always specified in pairs: a foreground color folowed by background color enclosed in pointed brackets < . . . > as usual. For convenience, the background color can be omited; in this case a completely transparent background is assumed.

⟨ parsing rules 5 ⟩ +≡ (292)

```
color_pair: START color color END
   | START color END { colors_n[colors_i++] = 0; };
color_unset:
  { colors_i += 2;
  }
  ;
```

A complete color set consists of six color pairs organized in two *color_tripple*s: the first three pairs for normal, mark, and focus text in day mode are followed by the three pairs in night mode. The *color_tripple* for night mode is optional; and within a *color_tripple* all color pairs except the first one are optional. An omited color is replaced by the corresponding color from the color set zero. To make the replacement process more predictable, the specification of color set zero—if present—must come first. If the default color set itself is redefined, an unspecified color will not change the default color.

To be open to future changes, color set definitions in the short format will contain after the reference number the number of color pairs that follow. Currently this value is always six.

⟨ parsing rules 5 ⟩ +≡ (293)

```
color_tripple: START color_pair color_unset color_unset END
   | START color_pair color_pair color_unset END
   | START color_pair color_pair color_pair END;
color_set: color_tripple color_tripple;
color_set: color_tripple color_unset color_unset color_unset; def_node: start
           COLOR ref { HPUT8(6); color_init(); } color_set END {
           DEF($$, color_kind, $3); hput_color_def($1, $3); };
```

```
⟨ put functions 14 ⟩ +≡                                                    (294)
  void color_init(void)
  { int i;
    for (i = 0; i < sizeof(ColorSet)/4; i++) colors_n[i] = colors_0[i];
    colors_i = 0;
  }
  static Tag hput_color_set(int n)
  { static bool first_color = true;
    int i;
    if (n ≡ 0) {
      if (first_color)
        for (i = 0; i < sizeof(ColorSet)/4; i++) colors_0[i] = colors_n[i];
      else QUIT("Redefinition␣of␣color␣set␣0␣must␣be␣the\
            ␣first␣color␣definition");
    }
    first_color = false;
    HPUTX(sizeof(ColorSet) + 1);
    for (i = 0; i < sizeof(ColorSet)/4; i++) HPUT32(colors_n[i]);
    return TAG(color_kind, b000);
  }
```

The *hput_color_def* checks if color sets zero or one need to be written. If not, the function will reset *hpos* to undo the writing of the tag and the number of colors in the set.

```
⟨ put functions 14 ⟩ +≡                                                    (295)
  static bool colors_equal(ColorSet a, ColorSet b)
  { int i;
    for (i = 0; i < sizeof(ColorSet)/4; i++)
      if (a[i] ≠ b[i]) return false;
    return true;
  }
  void hput_color_def(uint32_t pos, int n)
  {
    if ((n ≡ 0 ∧ colors_equal(color_defaults[0],
          colors_n)) ∨ (n ≡ 1 ∧ colors_equal(color_defaults[1], colors_n))) {
      hpos = hstart + pos;
      return;
    }
    hput_tags(pos, hput_color_set(n));
  }
```

Compared to the definitions, the content nodes are pretty simple. The special color number #FF is reserved to indicate an ⟨ color off ⟩ node in the short format.

⟨ parsing rules 5 ⟩ +≡ (296)

```
content_node: start COLOR ref END { REF_RNG(color_kind, $3);
        hput_tags($1, TAG(color_kind, b000)); };
content_node: start COLOR OFF END { HPUT8(#FF);
        hput_tags($1, TAG(color_kind, b000)); };
```

Writing the long format: ⟹ − − −
Reading the short format: ··· ⟹

We contine with the color content nodes:

⟨ cases to get content 20 ⟩ +≡ (297)

```
case TAG(color_kind, b000):
  { uint8_t n = HGET8; if (n ≡ #FF) hwritef("␣off");
    else { REF(color_kind, n); hwrite_ref(n); }
  }
  break;
```

And now we turn to the color definitions:

⟨ get functions 18 ⟩ +≡ (298)

```
void hwrite_color_pair(uint32_t f, uint32_t b)
{ hwritec('<');
  if (f ≡ 0) hwritec('0');
  else hwritef("0x%08X", f);
  if (b ≠ 0) hwritef("␣0x%08X", b);
  hwritec('>');
}
void hget_color_set(uint32_t node_pos, ColorSet cs)
{ int i, m;
  for (i = 0; i < sizeof(ColorSet)/4; i++) HGET32(cs[i]);
  for (m = 0; m < 2; m++) { uint32_t *c, *d;
    bool diff_high, diff_focus;
    if (m ≡ 0) { c = cs;
      d = color_defaults[0];
    }
    else { c = NIGHT(cs);
      d = NIGHT(color_defaults[0]);
      if (memcmp(c, d, sizeof(ColorSet)/2) ≡ 0) return;
    }
    hwrite_start();
    diff_high = FG(HIGH(c)) ≠ FG(HIGH(d)) ∨ BG(HIGH(c)) ≠ BG(HIGH(d));
    diff_focus = FG(FOCUS(c)) ≠ FG(FOCUS(d)) ∨ BG(FOCUS(c)) ≠ BG(FOCUS(d));
    hwrite_color_pair(FG(c), BG(c));
    if (diff_high ∨ diff_focus) { hwritec('␣');
      hwrite_color_pair(FG(HIGH(c)), BG(HIGH(c)));
    }
```

```
    if (diff_focus) { hwritec('␣');
      hwrite_color_pair(FG(FOCUS(c)), BG(FOCUS(c)));
    }
    hwrite_end();
  }
}
```

⟨ cases to get definitions for *color_kind* 299 ⟩ ≡ (299)

```
case b000:
  { int k;
    ColorSet c;
    static bool first_color = true;

    k = HGET8;
    if (k < 6) QUIT("Definition␣%d␣of␣color␣set␣needs␣6␣colo\
          r␣pairs␣only␣%d␣given\n", n, k);
    hget_color_set(node_pos, c);
    if (n ≡ 0) {
      if (¬first_color)
        QUIT("Definition␣of␣color␣set␣zero␣must␣be␣first");
      memcpy(&color_defaults[0], &c, sizeof(ColorSet));
    }
    first_color = false;
  }
  break;
```

Used in 396.

6.4 Rotation

When it comes to rotation, there is a big difference between printed books and computer displays. For example, if a book contains a table that is rotated to fill a page in landscape mode, the reader can rotate the book and read the table. If you are looking at the same page displayed on a big computer monitor, you will most likely not turn the whole monitor. Instead your viewing application will be able to perform the rotation for you before displaying the page. A smart phone, on the other hand, is easy to turn. But very likely, it will try to be smart and rerenders the content on the display to keep the same orientation.

Occasionaly, however, rotation of text is a desirable feature. For example, if a table has lots of tall columns with lenghty column headers. It might be usefull to rotate the column headers in order to keep the column width within reasonable limits.

A simple solution therefore would be optional parameters for boxes specifying center and angle for rotating the box.

6.5 Unknown Extensions

Starting with the inclusion in the TEX Live 2022 distribution, the HINT file format became accessible to a wider audience which brought the constant rewrite and upgrade cycle to a sudden halt. Except for bug fixes, pretty much nothing happened for a about a year. When the TEX Live 2023 distribution started to appear on the horizon, one extension that I had on my wish-list already for a long time—the support of TEX's *vtop* boxes—was definitely due for implementation. Adding new tag bytes to the specification of the short file format will, however, invalidate all HINT file viewers and requires everybody to upgrade the viewing application. Because the HINT file format is still in its infancy, more such additions are to be expected and the new version 2.0 file format needs a way to handle such yet unknown extensions gracefuly. For this purpose the definition section now may specify additional entries for the *hnode_size* array. All HINT file viewers starting with version 2.0 will use these entries to skip unknown nodes and display the remaining content of HINT files.

Reading the long format: $- - - \Longrightarrow$

In the long format, unknown nodes, whether in the definition or the content section, start with the keyword *unknown*.

⟨ symbols 2 ⟩ +≡ (300)

%**token** UNKNOWN `"unknown"`

⟨ scanning rules 3 ⟩ +≡ (301)

`unknown` **return** UNKNOWN;

In the definition section, the keyword is followed by the tag and the length of the initial part of the node (not counting the start byte), after which follows optionaly the number of trailing nodes embedded in the unknown node. There is no need for a maximum value, because the information is stored directly in the *hnode_size* array.

⟨ parsing rules 5 ⟩ +≡ (302)

def_node: *start* UNKNOWN UNSIGNED UNSIGNED END {
hput_tags(\$1, *hput_unknown_def*(\$3, \$4, 0)); }
| *start* UNKNOWN UNSIGNED UNSIGNED UNSIGNED END {
hput_tags(\$1, *hput_unknown_def*(\$3, \$4, \$5)); };

In the content section, the keyword is followed by the tag value, the remaining byte values belonging to the initial part and the nodes belonging to the trailing part. The end byte, which is equal to the start byte, is omited from the long format.

⟨ symbols 2 ⟩ +≡ (303)

%**type** < *u* > *unknown_bytes*
%**type** < *u* > *unknown_nodes*

⟨ parsing rules 5 ⟩ +≡ (304)

```
content_node: start UNKNOWN UNSIGNED unknown_bytes unknown_nodes END
        { hput_tags($1, hput_unknown($1, $3, $4, $5)); }
    unknown_bytes:
      { $$ = 0;
      }
    | unknown_bytes UNSIGNED { RNG("byte", $2, 0, #FF); HPUT8($2);
        $$ = $1 + 1; };
  unknown_node: content_node
| xdimen_node
| list
| named_param_list;
unknown_nodes:
  { $$ = 0;
  }
| unknown_nodes unknown_node
  { RNG("unknown␣subnodes", $1, 0, 3);
    $$ = $1 + 1;
  }
  ;
```

Writing the short format: ⟹ ⋯

In the short format, definitions for unknown nodes are marked with TAG(*unknown_kind*, *b100*). This tag is not used elsewhere (see also page 53). We do not check for multiple definitions of the same tag. But only the first of them is considered valid. After the start byte follows the unknown tag and the corresponding entry in the *hnode_size* array.

⟨ put functions 14 ⟩ +≡ (305)

```
uint32_t hput_unknown_def(uint32_t t, uint32_t b, uint32_t n)
{
  if (n ≡ 0) {
    RNG("unknown␣tag", t, TAG(param_kind, 7) + 1, TAG(int_kind, 0) − 1);
    RNG("unknown␣initial␣bytes", b, 0, #7F − 2);
    HPUT8(t);
    HPUT8(b + 2);                        /* adding start and end byte */
    if (hnode_size[t] ≡ 0) { hnode_size[t] = NODE_SIZE(b, 0);
      DBG(DBGTAGS,
          "Defining␣unknown␣node␣size␣%d,%d␣for␣tag␣0x%x\n", b, n, t);
    }
  }
  else { int i;
    RNG("unknown␣tag", t, TAG(param_kind, 7) + 1, TAG(int_kind, 0) − 1);
    RNG("unknown␣initial␣bytes", b, 0, #1F − 1);
    RNG("unknown␣trailing␣nodes", n, 1, 4);
```

```
      HPUT8(t);
      i = NODE_SIZE(b, n);
      HPUT8(i);
      if (hnode_size[t] ≡ 0) { hnode_size[t] = i;
        DBG(DBGTAGS,
            "Defining␣unknown␣node␣size␣%d,%d␣for␣tag␣0x%x\n", b, n, t);
      }
    }
    return TAG(unknown_kind, b100);
  }
```

In the content section, the unknown nodes are of course marked with their unknown tag.

⟨ put functions 14 ⟩ +≡ (306)

```
  Tag hput_unknown(uint32_t pos, uint32_t t, uint32_t b, uint32_t n)
  { int s;
    RNG("unknown␣tag", t, TAG(param_kind, 7) + 1, TAG(int_kind, 0) − 1);
    if (n ≡ 0) { RNG("unknown␣initial␣bytes", b, 0, #7F − 2);
      s = NODE_SIZE(b, 0);
    }
    else { RNG("unknown␣initial␣bytes", b, 0, #1F − 2);
      RNG("unknown␣trailing␣nodes", n, 1, 4);
      s = NODE_SIZE(b, n);
    }
    DBG(DBGTAGS, "Adding␣unknown␣node␣size␣%d,%d␣tag␣0x%x␣at␣0x%x\n",
        b, n, t, pos);
    if (hnode_size[t] ≠ s) QUIT("Size␣%d␣of␣unknown␣\
          node␣[%s,%d]␣at␣"SIZE_F"␣does␣not␣match␣%d\n", s, NAME(t),
          INFO(t), hpos − hstart, hnode_size[t]);
    return (Tag) t;
  }
```

Reading the short format: $\cdots \Longrightarrow$
Writing the long format: $\Longrightarrow - - -$

⟨ get functions 18 ⟩ +≡ (307)

```
  void hget_unknown_def(void)
  { Tag t;
    signed char i, b = 0, n = 0;
    t = HGET8;
    i = HGET8;
    if (i ≡ 0) QUIT("Zero␣not␣allowed␣for␣unknown␣node␣size␣at␣0x%x\n",
          (uint32_t)(hpos − hstart − 2));
    hwrite_start(); hwritef("unknown");
    b = NODE_HEAD(i);
    n = NODE_TAIL(i);
```

```
    if (n ≡ 0) hwritef("␣0x%02X␣%d", t, b);
    else hwritef("␣0x%02X␣%d␣%d", t, b, n);
    if (hnode_size[t] ≡ 0) { hnode_size[t] = i;
      DBG(DBGTAGS, "Defining␣node␣size␣%d,%d␣for␣tag␣0x%x\n", b, n, t);
    }
    hwrite_end();
  }
```

The *hget_unknown* funktion tries to process a unknown node with the help of an entry in the *hnode_size* array. The definition section can be used to provide this extra information. If successful the function returns 1 else 0.

⟨ get functions 18 ⟩ +≡ (308)

```
  int hget_unknown(Tag a)
  { int b, n;
    int8_t s;

    s = hnode_size[a];
    DBG(DBGTAGS, "Trying␣unknown␣tag␣0x%x␣at␣0x%x\n", a,
        (uint32_t)(hpos − hstart − 1));
    if (s ≡ 0) return 0;
    b = NODE_HEAD(s);
    n = NODE_TAIL(s);
    DBG(DBGTAGS, "Trying␣unknown␣node␣size␣%d␣%d\n", b, n);
    hwritef("unknown␣0x%02X", a);
    while (b > 0) { a = HGET8;
      hwritef("␣0x%02X", a);
      b−−;
    }
    while (n > 0) { a = ∗hpos;
      if (KIND(a) ≡ xdimen_kind) { Xdimen x;
        hget_xdimen_node(&x); hwrite_xdimen_node(&x);
      }
      else if (KIND(a) ≡ param_kind) { List l; hget_param_list(&l);
        hwrite_named_param_list(&l); }
      else if (KIND(a) ≤ list_kind) { List l; hget_list(&l); hwrite_list(&l); }
      else hget_content_node();
      n−−;
    }
    return 1;
  }
```

7 Replacing TEX's Page Building Process

TEX uses an output routine to finalize the page. It uses the accumulated material from the page builder, found in `box255`, attaches headers, footers, and floating material like figures, tables, and footnotes. The latter material is specified by insert nodes while headers and footers are often constructed using mark nodes. Running an output routine requires the full power of the TEX engine and will not be part of the HINT viewer. Therefore, HINT replaces output routines by page templates. As TEX can use different output routines for different parts of a book—for example the index might use a different output routine than the main body of text—HINT will allow multiple page templates. To support different output media, the page templates will be named and a suitable user interface may offer the user a selection of possible page layouts. In this way, the page layout remains in the hands of the book designer, and the user has still the opportunity to pick a layout that best fits the display device.

TEX uses insertions to describe floating content that is not necessarily displayed where it is specified. Three examples may illustrate this:

- Footnotes* are specified in the middle of the text but are displayed at the bottom of the page. Several footnotes on the same page are collected and displayed together. The page layout may specify a short rule to separate footnotes from the main text, and if there are many short footnotes, it may use two columns to display them. In extreme cases, the page layout may demand a long footnote to be split and continued on the next page.
- Illustrations may be displayed exactly where specified if there is enough room on the page, but may move to the top of the page, the bottom of the page, the top of next page, or a separate page at the end of the chapter.
- Margin notes are displayed in the margin on the same page starting at the top of the margin.

HINT uses page templates and content streams to achieve similar effects. But before I describe the page building mechanisms of HINT, let me summarize TEX's page builder.

TEX's page builder ignores leading glue, kern, and penalty nodes until the first box or rule is encountered; whatsit nodes do not really contribute anything to a page; mark nodes are recorded for later use. Once the first box, rule, or insert arrives, TEX makes copies of all parameters that influence the page building process

* Like this one.

and uses these copies. These parameters are the *page_goal* and the *page_max_depth*. Further, the variables *page_total*, *page_shrink*, *page_stretch*, *page_depth*, and *insert_penalties* are initialized to zero. The top skip adjustment is made when the first box or rule arrives—possibly after an insert.

Now the page builder accumulates material: normal material goes into `box255` and will change *page_total*, *page_shrink*, *page_stretch*, and *page_depth*. The latter is adjusted so that is does not exceed *page_max_depth*.

The handling of inserts is more complex. TEX creates an insert class using `newinsert`. This reserves a number n and four registers: `box`n for the inserted material, `count`n for the magnification factor f, `dimen`n for the maximum size per page d, and `skip`n for the extra space needed on a page if there are any insertions of class n.

For example plain TEX allocates $n = 254$ for footnotes and sets `count254` to 1000, `dimen254` to 8in, and `skip254` to `\bigskipamount`.

An insertion node will specify the insertion class n, some vertical material, its natural height plus depth x, a *split_top_skip*, a *split_max_depth*, and a *floating_penalty*.

Now assume that an insert node with subtype 254 arrives at the page builder. If this is the first such insert, TEX will decrease the *page_goal* by the width of skip254 and adds its stretchability and shrinkability to the total stretchability and shrinkability of the page. Later, the output routine will add some space and the footnote rule to fill just that much space and add just that much shrinkability and stretchability to the page. Then TEX will normally add the vertical material in the insert node to box254 and decrease the *page_goal* by $x \times f/1000$.

Special processing is required if TEX detects that there is not enough space on the current page to accommodate the complete insertion. If already a previous insert did not fit on the page, simply the *floating_penalty* as given in the insert node is added to the total *insert_penalties*. Otherwise TEX will test that the total natural height plus depth of box254 including x does not exceed the maximum size d and that the $\mathit{page\_total} + \mathit{page\_depth} + x \times f/1000 - \mathit{page\_shrink} \le \mathit{page\_goal}$. If one of these tests fails, the current insertion is split in such a way as to make the size of the remaining insertions just pass the tests just stated.

Whenever a glue node, or penalty node, or a kern node that is followed by glue arrives at the page builder, it rates the current position as a possible end of the page based on the shrinkability of the page and the difference between *page_total* and *page_goal*. As the page fills, the page breaks tend to become better and better until the page starts to get overfull and the page breaks get worse and worse until they reach the point where they become *awful_bad*. At that point, the page builder returns to the best page break found so far and fires up the output routine.

Let's look next at the problems that show up when implementing a replacement mechanism for HINT.

1. An insertion node can not always specify its height x because insertions may contain paragraphs that need to be broken in lines and the height of a paragraph depends in some non obvious way on its width.
2. Before the viewer can compute the height x, it needs to know the width of the

insertion. Just imagine displaying footnotes in two columns or setting notes in the margin. Knowing the width, it can pack the vertical material and derive its height and depth.

3. TEX's plain format provides an insert macro that checks whether there is still space on the current page, and if so, it creates a contribution to the main text body, otherwise it creates a topinsert. Such a decision needs to be postponed to the HINT viewer.
4. HINT has no output routines that would specify something like the space and the rule preceding the footnote.
5. TEX's output routines have the ability to inspect the content of the boxes, split them, and distribute the content over the page. For example, the output routine for an index set in two column format might expect a box containing index entries up to a height of $2 \times$ `vsize`. It will split this box in the middle and display the top part in the left column and the bottom part in the right column. With this approach, the last page will show two partly filled columns of about equal size.
6. HINT has no mark nodes that could be used to create page headers or footers. Marks, like output routines, contain token lists and need the full TEX interpreter for processing them. Hence, HINT does not support mark nodes.

Here now is the solution I have chosen for HINT:

Instead of output routines, HINT will use page templates. Page templates are basically vertical boxes with placeholders marking the positions where the content of the box registers, filled by the page builder, should appear. To output the page, the viewer traverses the page template, replaces the placeholders by the appropriate box content, and sets the glue. Inside the page template, we can use insert nodes to act as placeholders.

It is only natural to treat the page's main body, the inserts, and the marks using the same mechanism. We call this mechanism a content stream. Content streams are identified by a stream number in the range 0 to 254; the number 255 is used to indicate an invalid stream number. The stream number 0 is reserved for the main content stream; it is always defined. Besides the main content stream, there are three types of streams:

- normal streams correspond to TEX's inserts and accumulate content on the page,
- first streams correspond to TEX's first marks and will contain only the first insertion of the page,
- last streams correspond to TEX's bottom marks and will contain only the last insertion of the page, and
- top streams correspond to TEX's top marks. Top streams are not yet implemented.

Nodes from the content section are considered contributions to stream 0 except for insert nodes which will specify the stream number explicitly. If the stream is not defined or is not used in the current page template, its content is simply ignored.

The page builder needs a mechanism to redirect contributions from one content stream to another content stream based on the availability of space. Hence a HINT content stream can optionally specify a preferred stream number, where content should go if there is still space available, a next stream number, where content should go if the present stream has no more space available, and a split ratio if the content is to be split between these two streams before filling in the template.

Various stream parameters govern the treatment of contributions to the stream and the page building process.

- The magnification factor f: Inserting a box of height h to this stream will contribute $h \times f/1000$ to the height of the page under construction. For example, a stream that uses a two column format will have an f value of 500; a stream that specifies notes that will be displayed in the page margin will have an f value of zero.
- The height h: The extended dimension h gives the maximum height this stream is allowed to occupy on the current page. To continue the previous example, a stream that will be split into two columns will have $h = 2 \cdot \texttt{vsize}$, and a stream that specifies notes that will be displayed in the page margin will have $h = 1 \cdot \texttt{vsize}$. You can restrict the amount of space occupied by footnotes to the bottom quarter by setting the corresponding h value to $h = 0.25 \cdot \texttt{vsize}$.
- The depth d: The dimension d gives the maximum depth this stream is allowed to have after formatting.
- The width w: The extended dimension w gives the width of this stream when formatting its content. For example margin notes should have the width of the margin less some surrounding space.
- The "before" list b: If there are any contributions to this stream on the current page, the material in list b is inserted *before* the material from the stream itself. For example, the short line that separates the footnotes from the main page will go, together with some surrounding space, into the list b.
- The top skip glue g: This glue is inserted between the material from list b and the first box of the stream, reduced by the height of the first box. Hence it specifies the distance between the material in b and the first baseline of the stream content.
- The "after" list a: The list a is treated like list b but its material is placed *after* the material from the stream itself.
- The "preferred" stream number p: If $p \neq 255$, it is the number of the *preferred* stream. If stream p has still enough room to accommodate the current contribution, move the contribution to stream p, otherwise keep it. For example, you can move an illustration to the main content stream, provided there is still enough space for it on the current page, by setting $p = 0$.
- The "next" stream number n: If $n \neq 255$, it is the number of the *next* stream. If a contribution can not be accommodated in stream p nor in the current stream, treat it as an insertion to stream n. For example, you can move contributions to the next column after the first column is full, or move illustrations to a separate page at the end of the chapter.
- The split ratio r: If r is positive, both p and n must be valid stream numbers and contents is not immediately moved to stream p or n as described before.

Instead the content is kept in the stream itself until the current page is complete. Then, before inserting the streams into the page template, the content of this stream is formatted as a vertical box, the vertical box is split into a top fraction and a bottom fraction in the ratio $r/1000$ for the top and $(1000 - r)/1000$ for the bottom, and finally the top fraction is moved to stream p and the bottom fraction to stream n. You can use this feature for example to implement footnotes arranged in two columns of about equal size. By collecting all the footnotes in one stream and then splitting the footnotes with $r = 500$ before placing them on the page into a right and left column. Even three or more columns can be implemented by cascades of streams using this mechanism.

7.1 Stream Definitions

There are four types of streams: normal streams that work like TeX's inserts; and first, last, and top streams that work like TeX's marks. For the latter types, the long format uses a matching keyword and the short format the two least significant info bits. All stream definitions start with the stream number. In definitions of normal streams after the number follows in this order

- the maximum insertion height,
- the magnification factor, and
- information about splitting the stream. It consists of: a preferred stream, a next stream, and a split ratio. An asterisk indicates a missing stream reference, in the short format the stream number 255 serves the same purpose.

All stream definitions finish with

- the "before" list,
- an extended dimension node specifying the width of the inserted material,
- the top skip glue,
- the "after" list,
- and the total height, stretchability, and shrinkability of the material in the "before" and "after" list.

A special case is the stream definition for stream 0, the main content stream. None of the above information is necessary for it so it is omitted. Stream definitions, including the definition of stream 0, occur only inside page template definitions where they occur twice in two different roles: In the stream definition list, they define properties of the stream and in the template they mark the insertion point (see section 7.3). In the latter case, stream nodes just contain the stream number. Because a template looks like ordinary vertical material, we like to use the same functions for parsing it. But stream definitions are very different from stream content nodes. To solve the problem for the long format, the scanner will return two different tokens when it sees the keyword "`stream`". In the definition section, it will return STREAMDEF and in the content section STREAM. The same problem is solved in the short format by using the *b100* bit to mark a definition.

Reading the long format: $---\Longrightarrow$
Writing the short format: $\Longrightarrow\cdots$

⟨ symbols 2 ⟩ +≡ (309)

```
%token STREAM "stream"
%token STREAMDEF "stream␣(definition)"
%token FIRST "first"
%token LAST "last"
%token TOP "top"
%token NOREFERENCE "*"
%type < info > stream_type
%type < u > stream_ref
%type < rf > stream_def_node
```

⟨ scanning rules 3 ⟩ +≡ (310)

```
stream      if (section_no ≡ 1) return STREAMDEF;
            else return STREAM;
first       return FIRST;
last        return LAST;
top         return TOP;
\*          return NOREFERENCE;
```

⟨ parsing rules 5 ⟩ +≡ (311)

```
stream_link: ref { REF_RNG(stream_kind, $1); }
  | NOREFERENCE { HPUT8(255); };
stream_split: stream_link stream_link UNSIGNED
    { RNG("split␣ratio", $3, 0, 1000); HPUT16($3); };
stream_info: xdimen_node UNSIGNED
    { RNG("magnification␣factor", $2, 0, 1000); HPUT16($2); } stream_split;
stream_type: stream_info { $$ = 0; }
  | FIRST { $$ = 1; } | LAST { $$ = 2; } | TOP { $$ = 3; };
stream_def_node: start STREAMDEF ref stream_type
    list xdimen_node glue_node list glue_node END
    { DEF($$, stream_kind, $3); hput_tags($1, TAG(stream_kind, $4 | b100)); };
stream_ins_node: start STREAMDEF ref END
    { RNG("Stream␣insertion", $3, 0, max_ref[stream_kind]);
      hput_tags($1, TAG(stream_kind, b100)); };
content_node: stream_def_node | stream_ins_node;
```

Reading the short format: $\cdots \Longrightarrow$
Writing the long format: $\Longrightarrow$ − − −

⟨ get stream information for normal streams 312 ⟩ ≡ (312)

```
{ Xdimen x;
  uint16_t f, r;
  uint8_t n;
  DBG(DBGDEF, "Defining␣normal␣stream␣%d␣at␣"SIZE_F"\n", *(hpos − 1),
      hpos − hstart − 2);
  hget_xdimen_node(&x); hwrite_xdimen_node(&x);
  HGET16(f); RNG("magnification␣factor", f, 0, 1000); hwritef("␣%d", f);
  n = HGET8;
  if (n ≡ 255) hwritef("␣*");
  else { REF_RNG(stream_kind, n); hwrite_ref(n); }
  n = HGET8;
  if (n ≡ 255) hwritef("␣*");
  else { REF_RNG(stream_kind, n); hwrite_ref(n); }
  HGET16(r);
  RNG("split␣ratio", r, 0, 1000);
  hwritef("␣%d", r);
}
```

Used in 313.

⟨ get functions 18 ⟩ +≡ (313)

```
static bool hget_stream_def(void)
{ if (KIND(*hpos) ≠ stream_kind ∨ ¬(INFO(*hpos) & b100)) return false;
  else { Ref df;
    ⟨ read the start byte a 16 ⟩
    DBG(DBGDEF, "Defining␣stream␣%d␣at␣"SIZE_F"\n", *hpos,
        hpos − hstart − 1);
    DEF(df, stream_kind, HGET8);
    hwrite_start(); hwritef("stream"); hwrite_ref(df.n);
    if (df.n > 0) { Xdimen x; List l;
      if (INFO(a) ≡ b100) ⟨ get stream information for normal streams 312 ⟩
      else if (INFO(a) ≡ b101) hwritef("␣first");
      else if (INFO(a) ≡ b110) hwritef("␣last");
      else if (INFO(a) ≡ b111) hwritef("␣top");
      hget_list(&l); hwrite_list(&l);
      hget_xdimen_node(&x); hwrite_xdimen_node(&x);
      hget_glue_node(); hget_list(&l); hwrite_list(&l); hget_glue_node();
    }
    ⟨ read and check the end byte z 17 ⟩
    hwrite_end();
    return true;
  }
```

```
}
```

When stream definitions are part of the page template, we call them stream insertion points. They contain only the stream reference and are parsed by the usual content parsing functions.

⟨ cases to get content 20 ⟩ +≡ (314)

```
case TAG(stream_kind, b100):
  { uint8_t n = HGET8; REF_RNG(stream_kind, n); hwrite_ref(n); break; }
```

7.2 Stream Content

Stream nodes occur in the content section where they must not be inside other nodes except toplevel paragraph nodes. A normal stream node contains in this order: the stream reference number, the optional stream parameters, and the stream content. The content is either a vertical box or an extended vertical box. The stream parameters consists of the *floating_penalty*, the *split_max_depth*, and the *split_top_skip*. The parameterlist can be given explicitly or as a reference.

In the short format, the info bits *b010* indicate a normal stream content node with an explicit parameter list and the info bits *b000* a normal stream with a parameter list reference.

If the info bit *b001* is set, we have a content node of type top, first, or last. In this case, the short format has instead of the parameter list a single byte indicating the type. These types are currently not yet implemented.

Reading the long format: − − − ⟹
Writing the short format: ⟹ · · ·

⟨ symbols 2 ⟩ +≡ (315)

```
%type < info > stream
```

⟨ parsing rules 5 ⟩ +≡ (316)

```
stream: param_list list { $$ = b010; }
  | param_ref list { $$ = b000; };

content_node: start STREAM stream_ref stream END
    { hput_tags($1, TAG(stream_kind, $4)); };
```

Reading the short format: · · · ⟹
Writing the long format: ⟹ − − −

⟨ cases to get content 20 ⟩ +≡ (317)

```
case TAG(stream_kind, b000): HGET_STREAM(b000); break;
case TAG(stream_kind, b010): HGET_STREAM(b010); break;
```

When we read stream numbers, we relax the define before use policy. We just check, that the stream number is in the correct range.

⟨ get macros 19 ⟩ +≡ (318)

```
#define HGET_STREAM(I)
  { uint8_t n = HGET8; REF_RNG(stream_kind, n); hwrite_ref(n); }
  if ((I) & b010) { List l; hget_param_list(&l); hwrite_param_list(&l); }
  else HGET_REF(param_kind);
  { List l; hget_list(&l); hwrite_list(&l); }
```

7.3 Page Template Definitions

A HINT file can define multiple page templates. Not only might an index demand a different page layout than the main body of text, also the front page or the chapter headings might use their own page templates. Further, the author of a HINT file might define a two column format as an alternative to a single column format to be used if the display area is wide enough.

To help in selecting the right page template, page template definitions start with a name and an optional priority; the default priority is 1. The names might appear in a menu from which the user can select a page layout that best fits her taste. Without user interaction, the system can pick the template with the highest priority. Of course, a user interface might provide means to alter priorities. Future versions might include sophisticated feature-vectors that identify templates that are good for large or small displays, landscape or portrait mode, etc . . .

After the priority follows a glue node to specify the topskip glue and the dimension of the maximum page depth, an extended dimension to specify the page height and an extended dimension to specify the page width.

Then follows the main part of a page template definition: the template. The template consists of a list of vertical material. To construct the page, this list will be placed into a vertical box and the glue will be set. But of course before doing so, the viewer will scan the list and replace all stream insertion points by the appropriate content streams.

Let's call the vertical box obtained this way "the page". The page will fill the entire display area top to bottom and left to right. It defines not only the appearance of the main body of text but also the margins, the header, and the footer. Because the `vsize` and `hsize` variables of TEX are used for the vertical and horizontal dimension of the main body of text—they do not include the margins—the page will usually be wider than `hsize` and taller than `vsize`. The dimensions of the page are part of the page template. The viewer, knowing the actual dimensions of the display area, can derive from them the actual values of `hsize` and `vsize`.

Stream definitions are listed after the template.

The page template with number 0 is always defined and has priority 0. It will display just the main content stream. It puts a small margin of `hsize`/8 − 4.5pt all around it. Given a letter size page, 8.5 inch wide, this formula yields a margin of 1 inch, matching TEX's plain format. The margin will be positive as long as the page is wider than 1/2 inch. For narrower pages, there will be no margin at all. In general, the HINT viewer will never set `hsize` larger than the width of the page and `vsize` larger than its height.

Reading the long format: $---\Longrightarrow$
Writing the short format: $\Longrightarrow\cdots$

⟨ symbols 2 ⟩ +≡ (319)

```
%token PAGE "page"
```

⟨ scanning rules 3 ⟩ +≡ (320)

```
page            return PAGE;
```

⟨ parsing rules 5 ⟩ +≡ (321)

```
page_priority: { HPUT8(1); }
   | UNSIGNED { RNG("page priority",$1,0,255); HPUT8($1); };
stream_def_list:
   | stream_def_list stream_def_node;
page: string { hput_string($1); } page_priority glue_node dimension {
      HPUT32($5); } xdimen_node xdimen_node list stream_def_list;
```

Reading the short format: $\cdots\Longrightarrow$
Writing the long format: $\Longrightarrow---$

⟨ get functions 18 ⟩ +≡ (322)

```
void hget_page(void)
{ char *n;
  uint8_t p;
  Xdimen x;
  List l;
  HGET_STRING(n); hwrite_string(n);
  p = HGET8; if (p ≠ 1) hwritef(" %d", p);
  hget_glue_node();
  hget_dimen(TAG(dimen_kind, b001));
  hget_xdimen_node(&x); hwrite_xdimen_node(&x);      /* page height */
  hget_xdimen_node(&x); hwrite_xdimen_node(&x);      /* page width */
  hget_list(&l); hwrite_list(&l);
  while (hget_stream_def()) continue;
}
```

7.4 Page Ranges

Not every template is necessarily valid for the entire content section. A page range specifies a start position a and an end position b in the content section and the page template is valid if the start position p of the page is within that range: $a \le p < b$. If paging backward this definition might cause problems because the start position of the page is known only after the page has been build. In this case, the viewer might choose a page template based on the position at the bottom of the page. If it turns out that this “bottom template” is no longer valid when the page builder has found the start of the page, the viewer might display the page anyway with the bottom template, it might just display the page with the new "top template", or rerun the whole page building process using this time the "top template". Neither of these alternatives is guaranteed to produce a perfect result because changing the page template might change the amount of material that fits on the page. A good page template design should take this into account.

The representation of page ranges differs significantly for the short format and the long format. The short format will include a list of page ranges in the definition section which consist of a page template number, a start position, and an end position. In the long format, the start and end position of a page range is marked with a page range node switching the availability of a page template on and off. Such a page range node must be a top level node. It is an error, to switch a page template off that was not switched on, or to switch a page template on that was already switched on. It is permissible to omit switching off a page template at the very end of the content section.

While we parse a long format HINT file, we store page ranges and generate the short format after reaching the end of the content section. While we parse a short format HINT file, we check at the end of each top level node whether we should insert a page range node into the output. For the `shrink` program, it is best to store the start and end positions of all page ranges in an array sorted by the position*. To check the restrictions on the switching of page templates, we maintain for every page template an index into the range array which identifies the position where the template was switched on. A zero value instead of an index will identify templates that are currently invalid. When switching a range off again, we link the two array entries using this index. These links are useful when producing the range nodes in short format.

A range node in short format contains the template number, the start position and the end position. A zero start position is not stored, the info bit *b100* indicates a nonzero start position. An end position equal to `HINT_NO_POS` is not stored, the info bit *b010* indicates a smaller end position. The info bit *b001* indicates that positions are stored using 2 byte otherwise 4 byte are used for the positions.

⟨ hint types 1 ⟩ +≡ (323)

typedef struct { **uint8_t** *pg*; **uint32_t** *pos*; **bool** *on*; **int** *link*; } **RangePos**;

⟨ common variables 252 ⟩ +≡ (324)

RangePos ∗*range_pos*;
int *next_range* = 1, *max_range*;
int ∗*page_on*;

⟨ initialize definitions 253 ⟩ +≡ (325)

`ALLOCATE`(*page_on*, *max_ref*[*page_kind*] + 1, **int**);
`ALLOCATE`(*range_pos*, 2 ∗ (*max_ref*[*range_kind*] + 1), **RangePos**);

---

* For a HINT viewer, a data structure which allows fast retrieval of all valid page templates for a given position is needed.

⟨ hint macros 13 ⟩ +≡ (326)

```
#define ALLOCATE(R, S, T)
  ( (R) = ( T * ) calloc((S), sizeof (T)),
      (((R) ≡ NULL) ? QUIT("Out␣of␣memory␣for␣" #R) : 0) )
#define REALLOCATE(R, S, T)
  ( (R) = ( T * ) realloc((R), (S) * sizeof (T)),
      (((R) ≡ NULL) ? QUIT("Out␣of␣memory␣for␣" #R) : 0) )
```

Reading the long format: − − − ⟹

⟨ symbols 2 ⟩ +≡ (327)

```
%token RANGE "range"
```

⟨ scanning rules 3 ⟩ +≡ (328)

```
range          return RANGE;
```

⟨ parsing rules 5 ⟩ +≡ (329)

```
content_node: START RANGE REFERENCE ON END
      { REF(page_kind, $3); hput_range($3, true); }
    | START RANGE REFERENCE OFF END
      { REF(page_kind, $3); hput_range($3, false); };
```

Writing the long format: ⟹ − − −

⟨ write functions 21 ⟩ +≡ (330)

```
void hwrite_range(void)                        /* called in hwrite_end */
{ uint32_t p = hpos − hstart;
  DBG(DBGRANGE, "Range␣check␣at␣pos␣0x%x␣next␣at␣0x%x\n", p,
      range_pos[next_range].pos);
  while (next_range < max_range ∧ range_pos[next_range].pos ≤ p) {
    hwrite_start( );
    hwritef("range␣*%d␣", range_pos[next_range].pg);
    if (range_pos[next_range].on) hwritef("on");
    else hwritef("off");
    nesting −−; hwritec('>');          /* avoid a recursive call to hwrite_end */
    next_range ++;
  }
}
```

Reading the short format: ··· ⟹

⟨ get functions 18 ⟩ +≡ (331)

```
void hget_range(Info info, uint8_t pg)
{ uint32_t from, to;
  REF(page_kind, pg);
  REF(range_kind, (next_range - 1)/2);
  if (info & b100) { if (info & b001) HGET32(from); else HGET16(from); }
  else from = 0;
  if (info & b010) { if (info & b001) HGET32(to); else HGET16(to); }
  else to = HINT_NO_POS;
  range_pos[next_range].pg = pg;
  range_pos[next_range].on = true;
  range_pos[next_range].pos = from;
  DBG(DBGRANGE, "Range␣*%d␣from␣0x%x\n", pg, from);
  DBG(DBGRANGE, "Range␣*%d␣to␣0x%x\n", pg, to);
  next_range++;
  if (to ≠ HINT_NO_POS)
  { range_pos[next_range].pg = pg;
    range_pos[next_range].on = false;
    range_pos[next_range].pos = to;
    next_range++;
  }
}
```

⟨ write functions 21 ⟩ +≡ (332)

```
void hsort_ranges(void)                    /* simple insert sort by position */
{ int i;
  DBG(DBGRANGE, "Range␣sorting␣%d␣positions\n", next_range - 1);
  for (i = 3; i < next_range; i++)
  { int j = i - 1;
    if (range_pos[i].pos < range_pos[j].pos)
    { RangePos t;
      t = range_pos[i];
      do { range_pos[j + 1] = range_pos[j];
        j--;
      } while (range_pos[i].pos < range_pos[j].pos);
      range_pos[j + 1] = t;
    }
  }
  max_range = next_range; next_range = 1;      /* prepare for hwrite_range */
}
```

Writing the short format: ⟹ ···

⟨ put functions 14 ⟩ +≡ (333)

```
void hput_range(uint8_t pg, bool on)
{
  if (((next_range − 1)/2) > max_ref[range_kind])
    QUIT("Page␣range␣%d␣>␣%d", (next_range − 1)/2, max_ref[range_kind]);
  if (on ∧ page_on[pg] ≠ 0)
    QUIT("Template␣%d␣is␣switched␣on␣at␣0x%x␣and␣"SIZE_F,
        pg, range_pos[page_on[pg]].pos, hpos − hstart);
  else if (¬on ∧ page_on[pg] ≡ 0)
    QUIT("Template␣%d␣is␣switched␣off␣at␣"SIZE_F"␣but␣was␣not␣on",
        pg, hpos − hstart);
  DBG(DBGRANGE, "Range␣*%d␣%s␣at␣"SIZE_F"\n", pg, on ? "on" : "off",
      hpos − hstart);
  range_pos[next_range].pg = pg;
  range_pos[next_range].pos = hpos − hstart;
  range_pos[next_range].on = on;
  if (on) page_on[pg] = next_range;
  else
  { range_pos[next_range].link = page_on[pg];
    range_pos[page_on[pg]].link = next_range;
    page_on[pg] = 0;
  }
  next_range++;
}
void hput_range_defs(void)
{ int i;
  section_no = 1;
  hstart = dir[1].buffer;
  hend = hstart + dir[1].bsize;
  hpos = hstart + dir[1].size;
  for (i = 1; i < next_range; i++)
    if (range_pos[i].on)
    { Info info = b000;
      uint32_t p = hpos++ − hstart;
      uint32_t from, to;
      HPUT8(range_pos[i].pg);
      from = range_pos[i].pos;
      if (range_pos[i].link ≠ 0) to = range_pos[range_pos[i].link].pos;
      else to = HINT_NO_POS;
      if (from ≠ 0)
      { info = info | b100; if (from > #FFFF) info = info | b001; }
      if (to ≠ HINT_NO_POS)
      { info = info | b010; if (to > #FFFF) info = info | b001; }
```

```
      if (info & b100)
      { if (info & b001) HPUT32(from); else HPUT16(from); }
      if (info & b010)
      { if (info & b001) HPUT32(to); else HPUT16(to); }
      DBG(DBGRANGE, "Range␣*%d␣from␣0x%x␣to␣0x%x\n",
            range_pos[i].pg, from, to);
      hput_tags(p, TAG(range_kind, info));
    }
  hput_definitions_end();
}
```

8 File Structure

All HINT files start with a banner as described below. After that, they contain three mandatory sections: the directory section, the definition section, and the content section. Usually, further optional sections follow. In short format files, these contain auxiliary files (fonts, images, ...) necessary for rendering the content. In long format files, the directory section will simply list the file names of the auxiliary files.

8.1 Banner

All HINT files start with a banner. The banner contains only printable ASCII characters and spaces; its end is marked with a newline character. The first four byte are the "magic" number by which you recognize a HINT file. It consists of the four ASCII codes '`H`', '`I`', '`N`', and '`T`' in the long format and '`h`', '`i`', '`n`', and '`t`' in the short format. Then follows a space, then the version number, a dot, the sub-version number, and another space. Both numbers are encoded as decimal ASCII strings. The remainder of the banner is simply ignored but may be used to contain other useful information about the file. The maximum size of the banner is 256 byte.

⟨ hint macros 13 ⟩ +≡ (334)

```
#define MAX_BANNER 256
```

To check the banner, we have the function *hcheck_banner*; it returns *true* if successful.

⟨ common variables 252 ⟩ +≡ (335)

```
char hbanner[MAX_BANNER + 1];
int hbanner_size = 0;
```

⟨ function to check the banner 336 ⟩ ≡ (336)

```
bool hcheck_banner(char *magic)
{ int v, s;
  char *t;
  t = hbanner;
  if (strncmp(magic, hbanner, 4) ≠ 0) {
    MESSAGE("This␣is␣not␣a␣%s␣file\n", magic);
    return false;
  }
  else t += 4;
```

```
  if (hbanner[hbanner_size − 1] ≠ '\n') {
    MESSAGE("Banner␣exceeds␣maximum␣size=0x%x\n", MAX_BANNER);
    return false;
  }
  if (*t ≠ '␣') {
    MESSAGE("Space␣expected␣in␣banner␣after␣%s\n", magic);
    return false;
  }
  else t++;
  v = strtol(t, &t, 10);
  if (*t ≠ '.') {
    MESSAGE("Dot␣expected␣in␣banner␣after␣HINT␣version␣number\n");
    return false;
  }
  else t++;
  s = strtol(t, &t, 10);
  if (v ≠ HINT_VERSION) {
    MESSAGE("Wrong␣HINT␣version:␣got␣%d.%d,␣expected␣%d.%d\n", v, s,
        HINT_VERSION, HINT_MINOR_VERSION);
    return false;
  }
  if (s < HINT_MINOR_VERSION)
    { MESSAGE("Outdated␣HINT␣minor␣version:␣got␣%d.%d,\
        ␣expected␣%d.%d\n", v, s, HINT_VERSION, HINT_MINOR_VERSION);
  }
  else if (s > HINT_MINOR_VERSION)
    { MESSAGE("More␣recent␣HINT␣minor␣version:␣got␣%d.\
        %d,␣expected␣%d.%d,␣update␣your␣application\n", v, s,
        HINT_VERSION, HINT_MINOR_VERSION);
  }
  if (*t ≠ '␣' ∧ *t ≠ '\n') {
    MESSAGE("Space␣expected␣in␣banner␣after␣HINT␣minor␣version\n");
    return false;
  }
  LOG("%s␣file␣version␣"HINT_VERSION_STRING":%s", magic, t);
  DBG(DBGDIR, "banner␣size=0x%x\n", hbanner_size);
  return true;
}
```

Used in 549, 554, 555, and 557.

To read a short format file, we use the macro HGET8. It returns a single byte. We read the banner knowing that it ends with a newline character and is at most MAX_BANNER byte long. Because this is the first access to a yet unknown file, we are very careful and make sure we do not read past the end of the file. Checking the banner is a separate step.

Reading the short format: $\cdots \Longrightarrow$

⟨ get file functions 337 ⟩ ≡ (337)

```
void hget_banner(void)
{ hbanner_size = 0;
  while (hbanner_size < MAX_BANNER ∧ hpos < hend) { uint8_t c = HGET8;
    hbanner[hbanner_size++] = c;
    if (c ≡ '\n') break;
  }
  hbanner[hbanner_size] = 0;
}
```

Used in 549, 555, and 557.

To read a long format file, we use the function *fgetc*.

Reading the long format: $- - - \Longrightarrow$

⟨ read the banner 338 ⟩ ≡ (338)

```
{ hbanner_size = 0;
  while (hbanner_size < MAX_BANNER) { int c = fgetc(hin);
    if (c ≡ EOF) break;
    hbanner[hbanner_size++] = c;
    if (c ≡ '\n') break;
  }
  hbanner[hbanner_size] = 0;
}
```

Used in 554.

Writing the banner to a short format file is accomplished by calling *hput_banner* with the "magic" string `"hint"` as a first argument and a (short) comment as the second argument.

Writing the short format: $\Longrightarrow \cdots$

⟨ function to write the banner 339 ⟩ ≡ (339)

```
static size_t hput_banner(char *magic, char *str)
{ size_t s = fprintf(hout, "%s␣"HINT_VERSION_STRING"␣%s\n", magic, str);
  if (s > MAX_BANNER) QUIT("Banner␣too␣big");
  return s;
}
```

Used in 551, 554, and 555.

Writing the long format: $\Longrightarrow - - -$

Writing the banner of a long format file is essentially the same as for a short format file calling *hput_banner* with `"HINT"` as a first argument.

8.2 Long Format Files

After reading and checking the banner, reading a long format file is simply done by calling *yyparse*. The following rule gives the big picture:

Reading the long format: − − − ⟹

⟨ parsing rules 5 ⟩ +≡ (340)

hint: *directory_section definition_section content_section*;

8.3 Short Format Files

A short format file starts with the banner and continues with a list of sections. Each section has a maximum size of 2^{32} byte or 4GByte. This restriction ensures that positions inside a section can be stored as 32 bit integers, a feature that we will need only for the so called “content” section, but it is also nice for implementers to know in advance what sizes to expect. The big picture is captured by the *put_hint* function:

⟨ put functions 14 ⟩ +≡ (341)

```
static size_t hput_root(void);
static size_t hput_section(uint16_t n);
static size_t hput_optional_sections(void);

size_t hput_hint(char *str)
{ size_t s;
  DBG(DBGBASIC, "Writing␣hint␣output␣%s\n", str);
  s = hput_banner("hint", str);
  DBG(DBGDIR, "Root␣entry␣at␣"SIZE_F"\n", s);
  s += hput_root();
  DBG(DBGDIR, "Directory␣section␣at␣"SIZE_F"\n", s);
  s += hput_section(0);
  DBG(DBGDIR, "Definition␣section␣at␣"SIZE_F"\n", s);
  s += hput_section(1);
  DBG(DBGDIR, "Content␣section␣at␣"SIZE_F"\n", s);
  s += hput_section(2);
  DBG(DBGDIR, "Auxiliary␣sections␣at␣"SIZE_F"\n", s);
  s += hput_optional_sections();
  DBG(DBGDIR, "Total␣number␣of␣bytes␣written␣"SIZE_F"\n", s);
  return s;
}
```

When we work on a section, we will have the entire section in memory and use three variables to access it: *hstart* points to the first byte of the section, *hend* points to the byte after the last byte of the section, and *hpos* points to the current position inside the section. The auxiliary variable *hpos0* contains the *hpos* value of the last content node on nesting level zero.

⟨ common variables 252 ⟩ +≡ (342)

```
uint8_t *hpos = NULL, *hstart = NULL, *hend = NULL, *hpos0 = NULL;
```

There are two sets of macros that read or write binary data at the current position and advance the stream position accordingly.

Reading the short format: ··· ⟹

⟨ shared get macros 38 ⟩ +≡ (343)

```
#define HGET_ERROR
  QUIT ("HGET␣overrun␣in␣section␣%d␣at␣"SIZE_F"\n",
        section_no, hpos − hstart)
#define HEND ((hpos ≤ hend) ? 0 : (HGET_ERROR, 0))
#define HGET8 ((hpos < hend) ? *(hpos++) : (HGET_ERROR, 0))
#define HGET16(X) ((X) = (hpos[0] ≪ 8) + hpos[1], hpos += 2, HEND)
#define HGET24(X)
  ((X) = (hpos[0] ≪ 16) + (hpos[1] ≪ 8) + hpos[2], hpos += 3, HEND)
#define HGET32(X)
  ((X) = (hpos[0] ≪ 24) + (hpos[1] ≪ 16) + (hpos[2] ≪ 8) + hpos[3], hpos += 4,
      HEND)
#define HGETTAG(A) A = HGET8, DBGTAG(A, hpos − 1)
```

Writing the short format: ⟹ ···

⟨ put functions 14 ⟩ +≡ (344)

```
  void hput_error(void)
  { if (hpos < hend) return;
    QUIT("HPUT␣overrun␣section␣%d␣pos="SIZE_F"\n",
         section_no, hpos − hstart);
  }
```

⟨ put macros 345 ⟩ ≡ (345)

```
  extern void hput_error(void);
#define HPUT8(X) (hput_error( ), *(hpos++) = (X))
#define HPUT16(X) (HPUT8(((X) ≫ 8) & #FF), HPUT8((X) & #FF))
#define HPUT24(X)
  (HPUT8(((X) ≫ 16) & #FF), HPUT8(((X) ≫ 8) & #FF), HPUT8((X) & #FF))
#define HPUT32(X) (HPUT8(((X) ≫ 24) & #FF), HPUT8(((X) ≫ 16) & #FF),
      HPUT8(((X) ≫ 8) & #FF), HPUT8((X) & #FF))
```

Used in 550 and 554.

The above macros test for buffer overruns; allocating sufficient buffer space is done separately.

Before writing a node, we will insert a test and increase the buffer if necessary.

⟨ put macros 345 ⟩ +≡ (346)

```
  void hput_increase_buffer(uint32_t n);
#define HPUTX(N) (((hend − hpos) < (N)) ? hput_increase_buffer(N) : (void) 0)
#define HPUTNODE HPUTX(MAX_TAG_DISTANCE)
```

#define `HPUTTAG`(K,I)
  (`HPUTNODE`, `DBGTAG`(`TAG`(K,I), *hpos*), `HPUT8`(`TAG`(K,I)))

Fortunately the only data types that have an unbounded size are strings and texts. For these we insert specific tests. For all other cases a relatively small upper bound on the maximum distance between two tags can be determined. Currently the maximum distance between tags is 26 byte as can be determined from the *hnode_size* array described in appendix A. The definition below uses a slightly larger value leaving some room for future changes in the design of the short file format.

⟨ hint macros 13 ⟩ +≡ (347)
#define `MAX_TAG_DISTANCE` 32

8.4 Mapping a Short Format File to Memory

In the following, we implement two alternatives to map a file into memory. The first implementation, opens the file, gets its size, allocates memory, and reads the file. The second implementation uses a call to *mmap*.

Since modern computers with 64bit hardware have a huge address space, using *mmap* to map the entire file into virtual memory is the most efficient way to access a large file. "Mapping" is not the same as "reading" and it is not the same as allocating precious memory, all that is done by the operating system when needed. Mapping just reserves addresses. There is one disadvantage of mapping: it typically locks the underlying file and will not allow a separate process to modify it. This prevents using this method for previewing a HINT file while editing and recompiling it. In this case, the first implementation, which has a copy of the file in memory, is the better choice. To select the second implementation, define the macro `USE_MMAP`.

The following functions map and unmap a short format input file setting *hin_addr* to its address and *hin_size* to its size. The value *hin_addr* ≡ `NULL` indicates, that no file is open. The variable *hin_time* is set to the time when the file was last modified. It can be used to detect modifications of the file and reload it.

⟨ common variables 252 ⟩ +≡ (348)
  char ∗*hin_name* = `NULL`;
  uint64_t *hin_size* = 0;
  uint8_t ∗*hin_addr* = `NULL`;
  uint64_t *hin_time* = 0;

⟨ map functions 349 ⟩ ≡ (349)
#ifndef `USE_MMAP`
  void *hget_unmap*(**void**)
  { **if** (*hin_addr* ≠ `NULL`) *free*(*hin_addr*);
    hin_addr = `NULL`;
    hin_size = 0;
  }
  bool *hget_map*(**void**)
  { **FILE** ∗*f*;

```
    struct stat st;
    size_t s, t;
    uint64_t u;
    f = fopen(hin_name, "rb");
    if (f ≡ NULL)
    { MESSAGE("Unable␣to␣open␣file:␣%s\n", hin_name); return false; }
    if (stat(hin_name, &st) < 0) {
      MESSAGE("Unable␣to␣obtain␣file␣size:␣%s\n", hin_name);
      fclose(f);
      return false;
    }
    if (st.st_size ≡ 0) { MESSAGE("File␣%s␣is␣empty\n", hin_name);
      fclose(f);
      return false;
    }
    u = st.st_size;
    if (hin_addr ≠ NULL) hget_unmap( );
    hin_addr = malloc(u);
    if (hin_addr ≡ NULL) {
      MESSAGE("Unable␣to␣allocate␣0x%"PRIx64"␣byte␣for␣File␣%s\n", u,
          hin_name);
      fclose(f);
      return 0;
    }
    t = 0;
    do { s = fread(hin_addr + t, 1, u, f);
      if (s ≤ 0) { MESSAGE("Unable␣to␣read␣file␣%s\n", hin_name);
        fclose(f);
        free(hin_addr);
        hin_addr = NULL;
        return false;
      }
      t = t + s; u = u − s;
    } while (u > 0);
    hin_size = st.st_size;
    hin_time = st.st_mtime;
    return true;
  }
#else
#include <sys/mman.h>
  void hget_unmap(void)
  { munmap(hin_addr, hin_size);
    hin_addr = NULL;
    hin_size = 0;
  }
```

```
bool hget_map(void)
{ struct stat st;
  int fd;
  fd = open(hin_name, O_RDONLY, 0);
  if (fd < 0)
  { MESSAGE("Unable␣to␣open␣file␣%s\n", hin_name); return false; }
  if (fstat(fd, &st) < 0) { MESSAGE("Unable␣to␣get␣file␣size\n");
    close(fd);
    return false;
  }
  if (st.st_size ≡ 0) { MESSAGE("File␣%s␣is␣empty\n", hin_name);
    close(fd);
    return false;
  }
  if (hin_addr ≠ NULL) hget_unmap( );
  hin_size = st.st_size;
  hin_time = st.st_mtime;
  hin_addr = mmap(NULL, hin_size, PROT_READ, MAP_PRIVATE, fd, 0);
  if (hin_addr ≡ MAP_FAILED) { close(fd);
    hin_addr = NULL;
    hin_size = 0;
    MESSAGE("Unable␣to␣map␣file␣into␣memory\n");
    return 0;
  }
  close(fd);
  return hin_size;
}
#endif
```

Used in 549, 555, and 557.

8.5 Compression

The short file format offers the possibility to store sections in compressed form. We use the `zlib` compression library[2][1] to deflate and inflate individual sections. When one of the following functions is called, we can get the section buffer, the buffer size and the size actually used from the directory entry. If a section needs to be inflated, its size after decompression is found in the *xsize* field; if a section needs to be deflated, its size after compression will be known after deflating it.

⟨ get file functions 337 ⟩ +≡ (350)

```
static void hdecompress(uint16_t n)
{ z_stream z;                                   /* decompression stream */
  uint8_t *buffer;
  int i;
```

```
DBG(DBGCOMPRESS,
    "Decompressing␣section␣%d␣from␣0x%x␣to␣0x%x␣byte\n",
    n, dir[n].size, dir[n].xsize);
z.zalloc = (alloc_func)0; z.zfree = (free_func)0; z.opaque = (voidpf)0;
z.next_in = hstart;
z.avail_in = hend − hstart;
if (inflateInit(&z) ≠ Z_OK)
  QUIT("Unable␣to␣initialize␣decompression:␣%s", z.msg);
ALLOCATE(buffer, dir[n].xsize + MAX_TAG_DISTANCE, uint8_t);
DBG(DBGBUFFER,
    "Allocating␣output␣buffer␣size=0x%x,␣margin=0x%x\n",
    dir[n].xsize, MAX_TAG_DISTANCE);
z.next_out = buffer;
z.avail_out = dir[n].xsize + MAX_TAG_DISTANCE;
i = inflate(&z, Z_FINISH);
DBG(DBGCOMPRESS, "in:␣avail/total=0x%x/0x%lx␣"
    "out:␣avail/total=0x%x/0x%lx,␣return␣%d;\n",
    z.avail_in, z.total_in, z.avail_out, z.total_out, i);
if (i ≠ Z_STREAM_END)
  QUIT("Unable␣to␣complete␣decompression:␣%s", z.msg);
if (z.avail_in ≠ 0) QUIT("Decompression␣missed␣input␣data");
if (z.total_out ≠ dir[n].xsize)
  QUIT("Decompression␣output␣size␣mismatch␣0x%lx␣!=␣0x%x",
      z.total_out, dir[n].xsize);
if (inflateEnd(&z) ≠ Z_OK)
  QUIT("Unable␣to␣finalize␣decompression:␣%s", z.msg);
dir[n].buffer = buffer;
dir[n].bsize = dir[n].xsize;
hpos0 = hpos = hstart = buffer;
hend = hstart + dir[n].xsize;
}
```

⟨ put functions 14 ⟩ +≡ (351)

```
static void hcompress(uint16_t n)
{ z_stream z;                                  /* compression stream */
  uint8_t *buffer;
  int i;

  if (dir[n].size ≡ 0) { dir[n].xsize = 0; return; }
  DBG(DBGCOMPRESS, "Compressing␣section␣%d␣of␣size␣0x%x\n", n,
      dir[n].size);
  z.zalloc = (alloc_func)0; z.zfree = (free_func)0; z.opaque = (voidpf)0;
  if (deflateInit(&z, Z_DEFAULT_COMPRESSION) ≠ Z_OK)
    QUIT("Unable␣to␣initialize␣compression:␣%s", z.msg);
  ALLOCATE(buffer, dir[n].size + MAX_TAG_DISTANCE, uint8_t);
  z.next_out = buffer;
  z.avail_out = dir[n].size + MAX_TAG_DISTANCE;
```

```
  z.next_in = dir[n].buffer;
  z.avail_in = dir[n].size;
  i = deflate(&z, Z_FINISH);
  DBG(DBGCOMPRESS, "deflate␣in:␣avail/total=0x%x/0x%lx␣out:\
      ␣avail/total=0x%x/0x%lx,␣return␣%d;\n",
      z.avail_in, z.total_in, z.avail_out, z.total_out, i);
  if (z.avail_in ≠ 0) QUIT("Compression␣missed␣input␣data");
  if (i ≠ Z_STREAM_END) QUIT("Compression␣incomplete:␣%s", z.msg);
  if (deflateEnd(&z) ≠ Z_OK)
    QUIT("Unable␣to␣finalize␣compression:␣%s", z.msg);
  DBG(DBGCOMPRESS, "Compressed␣0x%lx␣byte␣to␣0x%lx␣byte\n",
      z.total_in, z.total_out);
  free(dir[n].buffer);
  dir[n].buffer = buffer;
  dir[n].bsize = dir[n].size + MAX_TAG_DISTANCE;
  dir[n].xsize = dir[n].size;
  dir[n].size = z.total_out;
}
```

8.6 Reading Short Format Sections

After mapping the file at address *hin_addr* access to sections of the file is provided by decompressing them if necessary and setting the three pointers *hpos*, *hstart*, and *hend*.

To read sections of a short format input file, we use the function *hget_section*.

Reading the short format: · · · ⟹

⟨ get file functions 337 ⟩ +≡ (352)

```
void hget_section(uint16_t n)
{ DBG(DBGDIR, "Reading␣section␣%d\n", n);
  RNG("Section␣number", n, 0, max_section_no);
  if (dir[n].buffer ≠ NULL ∧ dir[n].xsize > 0) {
    hpos0 = hpos = hstart = dir[n].buffer;
    hend = hstart + dir[n].xsize;
  }
  else { hpos0 = hpos = hstart = hin_addr + dir[n].pos;
    hend = hstart + dir[n].size;
    if (dir[n].xsize > 0) hdecompress(n);
  }
}
```

8.7 Writing Short Format Sections

To write a short format file, we allocate for each of the first three sections a suitable buffer, then fill these buffers, and finally write them out in sequential order.

⟨ put functions 14 ⟩ +≡ (353)

```
#define BUFFER_SIZE #400
  void new_output_buffers(void)
  { dir[0].bsize = dir[1].bsize = dir[2].bsize = BUFFER_SIZE;
    DBG(DBGBUFFER,
        "Allocating␣output␣buffer␣size=0x%x,␣margin=0x%x\n",
        BUFFER_SIZE, MAX_TAG_DISTANCE);
    ALLOCATE(dir[0].buffer, dir[0].bsize + MAX_TAG_DISTANCE, uint8_t);
    ALLOCATE(dir[1].buffer, dir[1].bsize + MAX_TAG_DISTANCE, uint8_t);
    ALLOCATE(dir[2].buffer, dir[2].bsize + MAX_TAG_DISTANCE, uint8_t);
  }
  void hput_increase_buffer(uint32_t n)
  { size_t bsize;
    uint32_t pos, pos0;
    const double buffer_factor = 1.4142136;                    /* √2 */
    pos = hpos − hstart;
    pos0 = hpos0 − hstart;
    bsize = dir[section_no].bsize ∗ buffer_factor + 0.5;
    if (bsize < pos + n) bsize = pos + n;
    if (bsize ≥ HINT_NO_POS) bsize = HINT_NO_POS;
    if (bsize < pos + n)
      QUIT("Unable␣to␣increase␣buffer␣size␣"SIZE_F"␣by␣0x%x␣byte",
          hpos − hstart, n);
    DBG(DBGBUFFER, "Reallocating␣output␣buffer␣"
        "␣for␣section␣%d␣from␣0x%x␣to␣"SIZE_F"␣byte\n", section_no,
        dir[section_no].bsize, bsize);
    REALLOCATE(dir[section_no].buffer, bsize, uint8_t);
    dir[section_no].bsize = (uint32_t) bsize;
    hstart = dir[section_no].buffer;
    hend = hstart + bsize;
    hpos0 = hstart + pos0;
    hpos = hstart + pos;
  }
  static size_t hput_data(uint16_t n, uint8_t ∗buffer, uint32_t size)
  { size_t s;
    s = fwrite(buffer, 1, size, hout);
    if (s ≠ size)
      QUIT("short␣write␣"SIZE_F"␣<␣%d␣in␣section␣%d", s, size, n);
    return s;
  }
```

```
static size_t hput_section(uint16_t n)
{ return hput_data(n, dir[n].buffer, dir[n].size);
}
```

9 Directory Section

A HINT file is subdivided in sections and each section can be identified by its section number. The first three sections, numbered 0, 1, and 2, are mandatory: directory section, definition section, and content section. The directory section, which we explain now, lists all sections that make up a HINT file.

A document will often contain not only plain text but also other media for example illustrations. Illustrations are produced with specialized tools and stored in specialized files. Because a HINT file in short format should be self contained, these special files are embedded in the HINT file as optional sections. Because a HINT file in long format should be readable, these special files are written to disk and only the file names are retained in the directory. Writing special files to disk has also the advantage that you can modify them individually before embedding them in a short format file.

9.1 Directories in Long Format

The directory section of a long format HINT file starts with the "`directory`" keyword; then follows the maximum section number used and a list of directory entries, one for each optional section numbered 3 and above. Each entry consists of the keyword "`section`" followed by the section number, followed by the file name. The section numbers must be unique and fit into 16 bit. The directory entries must be ordered with strictly increasing section numbers. Keeping section numbers consecutive is recommended because it reduces the memory footprint if directories are stored as arrays indexed by the section number as we will do below.

Reading the long format: $- - - \Longrightarrow$

⟨ symbols 2 ⟩ +≡ (354)

```
%token DIRECTORY "directory"
%token SECTION "entry"
```

⟨ scanning rules 3 ⟩ +≡ (355)

```
directory        return DIRECTORY;
section          return SECTION;
```

⟨ parsing rules 5 ⟩ +≡ (356)

```
  directory_section: START DIRECTORY UNSIGNED
        { new_directory($3 + 1); new_output_buffers(); } entry_list END;
  entry_list: | entry_list entry;
```

```
entry: START SECTION UNSIGNED string END
    { RNG("Section␣number", $3, 3, max_section_no);
      hset_entry(&(dir[$3]), $3, 0, 0, $4); };
```

We use a dynamically allocated array of directory entries to store the directory.

⟨ directory entry type 357 ⟩ ≡ (357)

```
typedef struct {
  uint64_t pos;
  uint32_t size, xsize;
  uint16_t section_no;
  char *file_name;
  uint8_t *buffer;
  uint32_t bsize;
} Entry;
```

Used in 548, 550, and 557.

The function *new_directory* allocates the directory.

⟨ directory functions 358 ⟩ ≡ (358)

```
Entry *dir = NULL;
uint16_t section_no, max_section_no;
void new_directory(uint32_t entries)
{ DBG(DBGDIR, "Creating␣directory␣with␣%d␣entries\n", entries);
  RNG("Directory␣entries", entries, 3, #10000);
  max_section_no = entries − 1; ALLOCATE(dir, entries, Entry);
  dir[0].section_no = 0; dir[1].section_no = 1; dir[2].section_no = 2;
}
```

Used in 549, 551, 554, 555, and 557.

The function *hset_entry* fills in the appropriate entry.

⟨ directory functions 358 ⟩ +≡ (359)

```
void hset_entry(Entry *e, uint16_t i, uint32_t size, uint32_t xsize,
      char *file_name)
{ e→section_no = i;
  e→size = size; e→xsize = xsize;
  if (file_name ≡ NULL ∨ *file_name ≡ 0) e→file_name = NULL;
  else e→file_name = strdup(file_name);
  DBG(DBGDIR, "Creating␣entry␣%d:␣\"%s\"␣size=0x%x␣xsize=0x%x\n",
      i, file_name, size, xsize);
}
```

Writing the auxiliary files depends on the -a, -g and -f options.

⟨ without -f skip writing an existing file 360 ⟩ ≡ (360)

```
if (¬option_force ∧ access(aux_name, F_OK) ≡ 0) {
  MESSAGE("File␣'%s'␣exists.\n"
      "To␣rewrite␣the␣file␣use␣the␣-f␣option.\n", aux_name);
  continue;
```

```
}
```

Used in 366.

The above code uses the *access* function, and we need to make sure it is defined:

⟨ make sure *access* is defined 361 ⟩ ≡ (361)

```
#ifdef WIN32
#include <io.h>
#define access(N, M) _access(N, M)
#define F_OK 0
#else
#include <unistd.h>
#endif
```

Used in 366.

With the `-g` option, filenames are considered global, and files are written to the filesystem possibly overwriting the existing files. For example a font embedded in a HINT file might replace a font of the same name in some operating systems font folder. If the HINT file is `shrink`ed on one system and `stretch`ed on another system, this is usually not the desired behavior. Without the `-g` option, the files will be written in two local directories. The names of these directories are derived from the output file name, replacing the extension "`.hint`" with "`.abs`" if the original filename contained an absolute path, and replacing it with "`.rel`" if the original filename contained a relative path. Inside these directories, the path as given in the filename is retained. When `shrink`ing a HINT file without the `-g` option, the original filenames can be reconstructed.

⟨ compute a local *aux_name* 362 ⟩ ≡ (362)

```
{ char *path = dir[i].file_name;
  int path_length = (int) strlen(path);
  ⟨ determine whether path is absolute or relative 363 ⟩
  ⟨ replace links to the parent directory 364 ⟩
  DBG(DBGDIR, "Replacing␣auxiliary␣file␣name:\n\t%s\n->\t%s\n", path,
      aux_name);
}
```

Used in 366 and 372.

⟨ determine whether *path* is absolute or relative 363 ⟩ ≡ (363)

```
int aux_length;
enum {
  absolute = 0, relative = 1
} name_type;
char *aux_ext[2] = {".abs/", ".rel/"};
int ext_length = 5;
aux_length = stem_length + ext_length + path_length;
ALLOCATE(aux_name, aux_length + 1, char);
strcpy(aux_name, stem_name);
if (path[0] ≡ '/') { name_type = absolute;
```

```
    strcpy(aux_name + stem_length, aux_ext[name_type]);
    strcpy(aux_name + stem_length + ext_length, path + 1);
  }
  else if (path_length > 3 ∧ isalpha(path[0]) ∧ path[1] ≡ ':' ∧ path[2] ≡ '/') {
    name_type = absolute;
    strcpy(aux_name + stem_length, aux_ext[name_type]);
    strcpy(aux_name + stem_length + ext_length, path);
    aux_name[stem_length + ext_length + 1] = '_';
  }
  else name_type = relative;
```

Used in 362.

When the -g is not given, auxiliar files are written into special subdirectories. To prevent them from escaping into the global file system, we replace links to the parent direcory "../" by "__/".

⟨ replace links to the parent directory 364 ⟩ ≡ (364)

```
  { int k;
    for (k = stem_length + ext_length; k < aux_length − 3; k++)
      if (aux_name[k] ≡ '.' ∧ aux_name[k + 1] ≡ '.' ∧ aux_name[k + 2] ≡ '/')
        { aux_name[k] = aux_name[k + 1] = '_';
        k = k + 2;
      }
  }
```

Used in 362.

It remains to create the directories along the path we might have constructed.

⟨ make sure the path in *aux_name* exists 365 ⟩ ≡ (365)

```
  { char *path_end;
    path_end = aux_name + 1;
    while (*path_end ≠ 0) {
      if (*path_end ≡ '/') { struct stat s;
        *path_end = 0;
        if (stat(aux_name, &s) ≡ −1) {
#ifdef WIN32
          if (mkdir(aux_name) ≠ 0)
#else
          if (mkdir(aux_name, °777) ≠ 0)
#endif
            QUIT("Unable␣to␣create␣directory␣%s", aux_name);
        DBG(DBGDIR, "Creating␣directory␣%s\n", aux_name);
      }
      else if (¬(S_IFDIR & (s.st_mode)))
        QUIT("Unable␣to␣create␣directory␣%s,␣file␣exists", aux_name);
      *path_end = '/';
    }
```

```
    path_end++;
  }
}
```

Used in 366 and 454.

Writing the long format: ⟹ – – –

⟨ write functions 21 ⟩ +≡ (366)

```
⟨ make sure access is defined 361 ⟩
extern char *stem_name;
extern int stem_length;
void hget_section(uint16_t n);
void hwrite_aux_files(void)
{ int i;
  if (¬option_aux) return;
  DBG(DBGBASIC | DBGDIR, "Writing␣%d␣aux␣files\n", max_section_no − 2);
  for (i = 3; i ≤ max_section_no; i++) { FILE *f;
    char *aux_name = NULL;
    if (option_global) aux_name = strdup(dir[i].file_name);
    else ⟨ compute a local aux_name 362 ⟩
    ⟨ without -f skip writing an existing file 360 ⟩
    ⟨ make sure the path in aux_name exists 365 ⟩
    f = fopen(aux_name, "wb");
    if (f ≡ NULL)
      QUIT("Unable␣to␣open␣file␣'%s'␣for␣writing", aux_name);
    else { size_t s;
      hget_section(i);
      DBG(DBGDIR, "Writing␣file␣%s\n", aux_name);
      s = fwrite(hstart, 1, dir[i].size, f);
      if (s ≠ dir[i].size) QUIT("writing␣file␣%s", aux_name);
      fclose(f);
    }
    free(aux_name);
  }
}
```

We write the directory, and the directory entries in long format using the following functions.

⟨ write functions 21 ⟩ +≡ (367)

```
static void hwrite_entry(int i)
{ hwrite_start();
  hwritef("section␣%u", dir[i].section_no); hwrite_string(dir[i].file_name);
  hwrite_end();
}
```

```
void hwrite_directory(void)
{ int i;
  if (dir ≡ NULL) QUIT("Directory␣not␣allocated");
  section_no = 0;
  hwritef("<directory␣%u", max_section_no);
  for (i = 3; i ≤ max_section_no; i++) hwrite_entry(i);
  hwritef("\n>\n");
}
```

9.2 Directories in Short Format

The directory section of a short format file contains entries for all sections including the directory section itself. After reading the directory section, enough information—position and size—is available to access any section directly. As usual, a directory entry starts and ends with a tag byte. The kind part of an entry's tag is not used; it is always zero. The value s of the two least significant bits of the info part indicate that sizes are stored using $s+1$ byte. The most significant bit of the info part is 1 if the section is stored in compressed form. In this case the size of the section is followed by the size of the section after decompressing it. After the tag byte follows the section number. In the short format file, section numbers must be strictly increasing and consecutive. This is redundant but helps with checking. Then follows the size—or the sizes—of the section. After the size follows the file name terminated by a zero byte. The file name might be an empty string in which case there is just the zero byte. After the zero byte follows a copy of the tag byte.

Here is the macro and function to read a directory entry:

Reading the short format: ··· ⟹

⟨shared get macros 38 ⟩ +≡ (368)

```
#define HGET_SIZE(I)
  if ((I) & b100) {
    if (((I) & b011) ≡ 0) s = HGET8, xs = HGET8;
    else if (((I) & b011) ≡ 1) HGET16(s), HGET16(xs);
    else if (((I) & b011) ≡ 2) HGET24(s), HGET24(xs);
    else if (((I) & b011) ≡ 3) HGET32(s), HGET32(xs);
  }
  else {
    if (((I) & b011) ≡ 0) s = HGET8;
    else if (((I) & b011) ≡ 1) HGET16(s);
    else if (((I) & b011) ≡ 2) HGET24(s);
    else if (((I) & b011) ≡ 3) HGET32(s);
  }
#define HGET_ENTRY(I, E)
  { uint16_t i;
    uint32_t s = 0, xs = 0;
    char *file_name;
```

```
    HGET16(i); HGET_SIZE(I); HGET_STRING(file_name);
    hset_entry(&(E), i, s, xs, file_name);
  }
```

⟨ get file functions 337 ⟩ +≡ (369)

```
  void hget_entry(Entry *e)
  { ⟨ read the start byte a 16 ⟩
    DBG(DBGDIR, "Reading␣directory␣entry\n");
    switch (a) {
    case TAG(0, b000 + 0): HGET_ENTRY(b000 + 0, *e); break;
    case TAG(0, b000 + 1): HGET_ENTRY(b000 + 1, *e); break;
    case TAG(0, b000 + 2): HGET_ENTRY(b000 + 2, *e); break;
    case TAG(0, b000 + 3): HGET_ENTRY(b000 + 3, *e); break;
    case TAG(0, b100 + 0): HGET_ENTRY(b100 + 0, *e); break;
    case TAG(0, b100 + 1): HGET_ENTRY(b100 + 1, *e); break;
    case TAG(0, b100 + 2): HGET_ENTRY(b100 + 2, *e); break;
    case TAG(0, b100 + 3): HGET_ENTRY(b100 + 3, *e); break;
    default: TAGERR(a); break;
    }
    ⟨ read and check the end byte z 17 ⟩
  }
```

Because the first entry in the directory section describes the directory section itself, we can not check its info bits in advance to determine whether it is compressed or not. Therefore the directory section starts with a root entry, which is always uncompressed. It describes the remainder of the directory which follows. There are two differences between the root entry and a normal entry: it starts with the maximum section number instead of the section number zero, and we set its position to the position of the entry for section 1 (which might already be compressed). The name of the directory section must be the empty string.

Reading the short format: ··· ⟹

⟨ get file functions 337 ⟩ +≡ (370)

```
  static void hget_root(Entry *root)
  { DBG(DBGDIR, "Root␣entry␣at␣"SIZE_F"\n", hpos − hstart);
    hget_entry(root);
    root→pos = hpos − hstart;
    max_section_no = root→section_no;
    root→section_no = 0;
    if (max_section_no < 2) QUIT("Sections␣0,␣1,␣and␣2␣are␣mandatory");
  }

  void hget_directory(void)
  { int i;
    Entry root = {0};
```

```
    hget_root(&root);
    DBG(DBGDIR, "Directory\n");
    new_directory(max_section_no + 1);
    dir[0] = root;
    DBG(DBGDIR, "Directory␣entry␣1␣at␣0x%" PRIx64 "\n", dir[0].pos);
    hget_section(0);
    for (i = 1; i ≤ max_section_no; i++)
    { hget_entry(&(dir[i])); dir[i].pos = dir[i − 1].pos + dir[i − 1].size;
      DBG(DBGDIR, "Section␣%d␣at␣0x%" PRIx64 "\n", i, dir[i].pos);
    }
  }
  void hclear_dir(void)
  { int i;
    if (dir ≡ NULL) return;
    for (i = 0; i < 3; i++)          /* currently the only compressed sections */
      if (dir[i].xsize > 0 ∧ dir[i].buffer ≠ NULL) free(dir[i].buffer);
    free(dir);
    dir = NULL;
  }
```

Armed with these preparations, we can put the directory into the HINT file.

Writing the short format: ⟹ ⋯

⟨ put functions 14 ⟩ +≡ (371)

```
  static void hput_entry(Entry *e)
  { Info b;
    if (e→size < #100 ∧ e→xsize < #100) b = b000;
    else if (e→size < #10000 ∧ e→xsize < #10000) b = b001;
    else if (e→size < #1000000 ∧ e→xsize < #1000000) b = b010;
    else b = b011;
    if (e→xsize ≠ 0) b = b | b100;
    DBG(DBGTAGS, "Directory␣entry␣no=%d␣size=0x%x␣xsize=0x%x\n",
        e→section_no, e→size, e→xsize);
    HPUTTAG(0, b);
    HPUT16(e→section_no);
    switch (b) {
    case b000: HPUT8(e→size); break;
    case b001: HPUT16(e→size); break;
    case b010: HPUT24(e→size); break;
    case b011: HPUT32(e→size); break;
    case b100: HPUT8(e→size); HPUT8(e→xsize); break;
    case b101: HPUT16(e→size); HPUT16(e→xsize); break;
    case b110: HPUT24(e→size); HPUT24(e→xsize); break;
    case b111: HPUT32(e→size); HPUT32(e→xsize); break;
    default: QUIT("Can't␣happen"); break;
```

```
    }
    hput_string(e->file_name);
    DBGTAG(TAG(0, b), hpos); HPUT8(TAG(0, b));
  }
  static void hput_directory_start(void)
  { DBG(DBGDIR, "Directory␣Section\n");
    section_no = 0;
    hpos = hstart = dir[0].buffer;
    hend = hstart + dir[0].bsize;
  }
  static void hput_directory_end(void)
  { dir[0].size = hpos - hstart;
    DBG(DBGDIR, "End␣Directory␣Section␣size=0x%x\n", dir[0].size);
  }
  static size_t hput_root(void)
  { uint8_t buffer[MAX_TAG_DISTANCE];
    size_t s;
    hpos = hstart = buffer;
    hend = hstart + MAX_TAG_DISTANCE;
    dir[0].section_no = max_section_no;
    hput_entry(&dir[0]);
    s = hput_data(0, hstart, hpos - hstart);
    DBG(DBGDIR, "Writing␣root␣size="SIZE_F"\n", s);
    return s;
  }
  extern int option_compress;
  static char **aux_names;
  void hput_directory(void)
  { int i;
    ⟨ update the file sizes of optional sections 372 ⟩
    if (option_compress) { hcompress(1); hcompress(2); }
    hput_directory_start();
    for (i = 1; i <= max_section_no; i++) {
      dir[i].pos = dir[i - 1].pos + dir[i - 1].size;
      DBG(DBGDIR, "writing␣entry␣%u␣at␣0x%" PRIx64 "\n", i, dir[i].pos);
      hput_entry(&dir[i]);
    }
    hput_directory_end();
    if (option_compress) hcompress(0);
  }
```

Now let us look at the optional sections described in the directory entries 3 and above. Where these files are found depends on the `-g` and `-a` options.

With the `-g` option given, only the file names as given in the directory entries are used. With the `-a` option given, the file names are translated to filenames in the *hin_name*`.abs` and *hin_name*`.rel` directories, as described in section 9.1. If neither the `-a` nor the `-g` option is given, `shrink` first tries the translated filename and then the global filename before it gives up.

When the `shrink` program writes the directory section in the short format, it needs to know the sizes of all the sections—including the optional sections. These sizes are not provided in the long format because it is safer and more convenient to let the machine figure out the file sizes. But before we can determine the size, we need to determine the file.

⟨ update the file sizes of optional sections 372 ⟩ ≡ (372)

```
{ int i;
  ALLOCATE(aux_names, max_section_no + 1, char *);
  for (i = 3; i ≤ max_section_no; i++) { struct stat s;
    if (¬option_global) { char *aux_name = NULL;
      ⟨ compute a local aux_name 362 ⟩
      if (stat(aux_name, &s) ≡ 0) aux_names[i] = aux_name;
      else {
        if (option_aux) QUIT("Unable␣to␣find␣file␣'%s'", aux_name);
        free(aux_name);
        aux_name = NULL;
      }
    }
    if ((aux_names[i] ≡ NULL ∧ ¬option_aux) ∨ option_global) {
      if (stat(dir[i].file_name, &s) ≠ 0)
        QUIT("Unable␣to␣find␣file␣'%s'", dir[i].file_name);
    }
    dir[i].size = s.st_size;
    dir[i].xsize = 0;
    DBG(DBGDIR, "section␣%i:␣found␣file␣%s␣size␣%u\n", i,
        aux_names[i] ? aux_names[i] : dir[i].file_name, dir[i].size);
  }
}
```

Used in 371.

⟨ rewrite the file names of optional sections 373 ⟩ ≡ (373)

```
{ int i;
  for (i = 3; i ≤ max_section_no; i++)
    if (aux_names[i] ≠ NULL) { free(dir[i].file_name);
      dir[i].file_name = aux_names[i];
      aux_names[i] = NULL;
    }
}
```

Used in 554.

The computation of the sizes of the mandatory sections will be explained later.

To conclude this section, here is the function that adds the files that are described in the directory entries 3 and above to a HINT file in short format.

Writing the short format: ⟹ ⋯

⟨ put functions 14 ⟩ +≡ (374)

```
static size_t hput_optional_sections(void)
{ int i;
  size_t s = 0;
  DBG(DBGDIR, "Optional␣Sections\n");
  for (i = 3; i ≤ max_section_no; i++)
  { FILE *f;
    size_t fsize;
    char *file_name = dir[i].file_name;
    DBG(DBGDIR, "adding␣file␣%d:␣%s\n", dir[i].section_no, file_name);
    if (dir[i].xsize ≠ 0)
      DBG(DBGDIR, "Compressing␣of␣auxiliary␣files␣currentl\
          y␣not␣supported");
    f = fopen(file_name, "rb");
    if (f ≡ NULL) QUIT("Unable␣to␣read␣section␣%d,␣file␣%s",
           dir[i].section_no, file_name);
    fsize = 0;
    while (¬feof(f))
    { size_t s, t;
      char buffer[1 ≪ 13];                                   /* 8kByte */
      s = fread(buffer, 1, 1 ≪ 13, f);
      t = fwrite(buffer, 1, s, hout);
      if (s ≠ t) QUIT("writing␣file␣%s", file_name);
      fsize = fsize + t;
    }
    fclose(f);
    if (fsize ≠ dir[i].size)
      QUIT("File␣size␣"SIZE_F"␣does␣not␣match␣section[0]␣size␣%u",
          fsize, dir[i].size);
    s = s + fsize;
  }
  return s;
}
```

10 Definition Section

In a typical HINT file, there are many things that are used over and over again. For example the interword glue of a specific font or the indentation of the first line of a paragraph. The definition section contains this information so that it can be referenced in the content section by a simple reference number. In addition there are a few parameters that guide the routines of TEX. An example is the "above display skip", which controls the amount of white space inserted above a displayed equation, or the "hyphen penalty" that tells TEX the "æsthetic cost" of ending a line with a hyphenated word. These parameters also get their values in the definition section as explained in section 11.

The most simple way to store these definitions is to store them in an array indexed by the reference numbers. To simplify the dynamic allocation of these arrays, the list of definitions will always start with the list of maximum values: a list that contains for each node type the maximum reference number used.

In the long format, the definition section starts with the keyword `definitions`, followed by the list of maximum values, followed by the definitions proper.

When writing the short format, we start by positioning the output stream at the beginning of the definition buffer and we end with recording the size of the definition section in the directory.

Reading the long format: −−− ⟹

⟨ symbols 2 ⟩ +≡ (375)

%**token** DEFINITIONS `"definitions"`

⟨ scanning rules 3 ⟩ +≡ (376)

`definitions` **return** DEFINITIONS;

⟨ parsing rules 5 ⟩ +≡ (377)

definition_section: START DEFINITIONS { *hput_definitions_start*( ); }
max_definitions *definition_list*
END { *hput_definitions_end*( ); };

definition_list: | *definition_list* *def_node*;

Writing the long format: ⟹ − − −

⟨ write functions 21 ⟩ +≡ (378)

```
void hwrite_definitions_start(void)
{ section_no = 1; hwritef("<definitions");
}

void hwrite_definitions_end(void)
{ hwritef("\n>\n");
}
```

⟨ get functions 18 ⟩ +≡ (379)

```
void hget_definition_section(void)
{ DBG(DBGBASIC | DBGDEF, "Definitions\n");
  hget_section(1);
  hwrite_definitions_start();
  DBG(DBGDEF, "List of maximum values\n");
  hget_max_definitions();
  ⟨ initialize definitions 253 ⟩
  hwrite_max_definitions();
  DBG(DBGDEF, "List of definitions\n");
  while (hpos < hend) hget_def_node();
  hwrite_definitions_end();
}
```

Writing the short format: ⟹ ⋯

⟨ put functions 14 ⟩ +≡ (380)

```
void hput_definitions_start(void)
{ DBG(DBGDEF, "Definition Section\n");
  section_no = 1;
  hpos = hstart = dir[1].buffer;
  hend = hstart + dir[1].bsize;
}

void hput_definitions_end(void)
{ dir[1].size = hpos − hstart;
  DBG(DBGDEF, "End Definition Section size=0x%x\n", dir[1].size);
}
```

10.1 Maximum Values

To help implementations allocating the right amount of memory for the definitions, the definition section starts with a list of maximum values. For each kind of node, we store the maximum valid reference number in the array *max_ref* which is indexed by the kind-values. For a reference number n and kind-value k we have $0 \le n \le max\_ref[k]$. To make sure that a hint file without any definitions will work, some definitions have default values. The initialization of default and maximum values is described in section 11. The maximum reference number that has a default value is stored in the array *max_default*. We have $-1 \le max\_default[k] \le max\_ref[k] < 2^{16}$, and for most k even $max\_ref[k] < 2^8$. Specifying maximum values that are lower than the default values is not allowed in the short format; in the long format, lower values are silently ignored. Some default values are permanently fixed; for example the zero glue with reference number *zero_skip_no* must never change. The array *max_fixed* stores the maximum reference number that has a fixed value for a given kind. Definitions with reference numbers less or equal than the corresponding $max\_fixed[k]$ number are disallowed. Usually we have $-1 \le max\_fixed[k] \le max\_default[k]$, but if for a kind-value k no definitions, and hence no maximum values are allowed, we set $max\_fixed[k] = \#\texttt{10000} > max\_default[k]$.

We use the *max_ref* array whenever we find a reference number in the input to check if it is within the proper range.

⟨ debug macros 381 ⟩ ≡ (381)

#define `REF_RNG`(K,N) **if** $((\textbf{int})(N) > max\_ref[K])$
 `QUIT("Reference␣%d␣to␣%s␣out␣of␣range␣[0␣-␣%d]",` (N), $definition\_name[K], max\_ref[K])$

Used in 545.

In the long format file, the list of maximum values starts with "<max ", then follow pairs of keywords and numbers like "<glue 57>", and it ends with ">". In the short format, we start the list of maximums with a *list_kind* tag and end it with a *list_kind* tag. Each maximum value is preceded and followed by a tag byte with the appropriate kind-value. The info value has its *b001* bit cleared if the maximum value is in the range 0 to #`FF` and fits into a single byte; the info value hast its *b001* bit set if it fits into two byte. Currently only the *label_kind* may need to use two byte.

⟨ debug macros 381 ⟩ +≡ (382)

#define `MAX_REF` (K) $((K) \equiv label\_kind\ ?\ \#\texttt{FFFF} : \#\texttt{FF})$

Other info values are reserved for future extensions. After reading the maximum values, we initialize the data structures for the definitions.

Reading the long format: − − − ⟹

⟨ symbols 2 ⟩ +≡ (383)

```
%token MAX "max"
```

⟨ scanning rules 3 ⟩ +≡ (384)

```
max                return MAX;
```

⟨ parsing rules 5 ⟩ +≡ (385)

```
max_definitions: START MAX max_list END
        { ⟨ initialize definitions 253 ⟩ hput_max_definitions( ); };
max_list: | max_list START max_value END;
max_value: FONT UNSIGNED { hset_max(font_kind, $2); }
    | INTEGER UNSIGNED { hset_max(int_kind, $2); }
    | DIMEN UNSIGNED { hset_max(dimen_kind, $2); }
    | LIGATURE UNSIGNED { hset_max(ligature_kind, $2); }
    | DISC UNSIGNED { hset_max(disc_kind, $2); }
    | GLUE UNSIGNED { hset_max(glue_kind, $2); }
    | LANGUAGE UNSIGNED { hset_max(language_kind, $2); }
    | RULE UNSIGNED { hset_max(rule_kind, $2); }
    | IMAGE UNSIGNED { hset_max(image_kind, $2); }
    | LEADERS UNSIGNED { hset_max(leaders_kind, $2); }
    | BASELINE UNSIGNED { hset_max(baseline_kind, $2); }
    | XDIMEN UNSIGNED { hset_max(xdimen_kind, $2); }
    | PARAM UNSIGNED { hset_max(param_kind, $2); }
    | STREAMDEF UNSIGNED { hset_max(stream_kind, $2); }
    | PAGE UNSIGNED { hset_max(page_kind, $2); }
    | RANGE UNSIGNED { hset_max(range_kind, $2); }
    | LABEL UNSIGNED { hset_max(label_kind, $2); }
    | COLOR UNSIGNED { hset_max(color_kind, $2); };
```

⟨ parsing functions 386 ⟩ ≡ (386)

```
void hset_max(Kind k, int n)
{ DBG(DBGDEF, "Setting␣max␣%s␣to␣%d\n", definition_name[k], n);
  RNG("Maximum", n, max_fixed[k] + 1, MAX_REF(k));
  if (n > max_ref[k]) max_ref[k] = n;
}
```

Used in 553.

Writing the long format: ⟹ − − −

⟨ write functions 21 ⟩ +≡ (387)

```
void hwrite_max_definitions(void)
{ Kind k;
  hwrite_start(); hwritef("max");
  for (k = 0; k < 32; k++)
    if (max_ref[k] > max_default[k])
    { switch (k) { ⟨ cases of writing special maximum values 248 ⟩
      default: hwrite_start();
        hwritef("%s␣%d", definition_name[k], max_ref[k]);
        hwrite_end();
        break;
      }
    }
  hwrite_end();
}
```

Reading the short format: · · · ⟹

⟨ get file functions 337 ⟩ +≡ (388)

```
void hget_max_definitions(void)
{ Kind k;
  ⟨ read the start byte a 16 ⟩
  if (a ≠ TAG(list_kind, 0)) QUIT("Start␣of␣maximum␣list␣expected");
  for (k = 0; k < 32; k++) max_ref[k] = max_default[k];
  max_outline = −1;
  while (true)
  { int n;
    if (hpos ≥ hend) QUIT("Unexpected␣end␣of␣maximum␣list");
    node_pos = hpos − hstart;
    HGETTAG(a); k = KIND(a); if (k ≡ list_kind) break;
    if (INFO(a) & b001) HGET16(n); else n = HGET8;
    switch (a) { ⟨ cases of getting special maximum values 246 ⟩
    default:
      if (max_fixed[k] > max_default[k])
        MESSAGE("Maximum␣value␣for␣kind␣%s␣not␣supported\n",
            definition_name[k]);
      else { RNG("Maximum␣number", n, max_default[k], MAX_REF(k));
        max_ref[k] = n;
        DBG(DBGDEF, "max(%s)␣=␣%d\n", definition_name[k], max_ref[k]);
      }
      break;
    }
    ⟨ read and check the end byte z 17 ⟩
  }
```

```
    if (INFO(a) ≠ 0) QUIT("End␣of␣maximum␣list␣with␣info␣%d", INFO(a));
    DBG(DBGDEF, "Getting␣Max␣Definitions␣END\n");
  }
```

Writing the short format: ⟹ ···

```
⟨ put functions 14 ⟩ +≡                                                    (389)
  void hput_max_definitions(void)
  { Kind k;
    DBG(DBGDEF, "Writing␣Max␣Definitions\n");
    HPUTTAG(list_kind, 0);
    for (k = 0; k < 32; k++)
      if (max_ref[k] > max_default[k]) { uint32_t pos = hpos++ − hstart;
        DBG(DBGDEF, "max(%s)␣=␣%d\n", definition_name[k], max_ref[k]);
        hput_tags(pos, TAG(k, hput_n(max_ref[k]) − 1));
      }
    ⟨ cases of putting special maximum values 247 ⟩
    HPUTTAG(list_kind, 0);
    DBG(DBGDEF, "Writing␣Max␣Definitions␣End\n");
  }
```

10.2 Definitions

A definition associates a reference number with a content node. Here is an example: A glue definition associates a glue number, for example 71, with a glue specification. In the long format this might look like "`<glue *71 4pt plus 5pt minus 0.5pt>`" which makes glue number 71 refer to a 4pt glue with a stretchability of 5pt and a shrinkability of 0.5pt. Such a glue definition differs from a normal glue node just by an extra byte value immediately following the keyword respectively start byte.

Whenever we need this glue in the content section, we can say "`<glue *71>`". Because we restrict the number of glue definitions to at most 256, a single byte is sufficient to store the reference number. The `shrink` and `stretch` programs will, however, not bother to store glue definitions. Instead they will write them in the new format immediately to the output.

The parser will handle definitions in any order, but the order is relevant if a definition references another definition, and of course, it never does any harm to present definitions in a systematic way.

As a rule, the definition of a reference must always precede the use of that reference. While this is always the case for references in the content section, it restricts the use of references inside the definition section.

The definitions for integers, dimensions, extended dimensions, languages, rules, ligatures, and images are "simple". They never contain references and so it is always possible to list them first. The definition of glues may contain extended dimensions, the definitions of baselines may reference glue nodes, and the definitions of parameter lists contain definitions of integers, dimensions, and glues. So these definitions should follow in this order.

The definitions of leaders and discretionary breaks allow boxes. While these boxes are usually quite simple, they may contain arbitrary references—including again references to leaders and discretionary breaks. So, at least in principle, they might impose complex (or even unsatisfiable) restrictions on the order of those definitions.

The definitions of fonts contain not only “simple” definitions but also the definitions of interword glues and hyphens introducing additional ordering restrictions. The definition of hyphens regularly contain glyphs which in turn reference a font—typically the font that just gets defined. Therefore we relax the define before use policy for glyphs: Glyphs may reference a font before the font is defined.

The definitions of page templates contain lists of arbitrary content nodes, and while the boxes inside leaders or discretionary breaks tend to be simple, the content of page templates is often quite complex. Page templates are probably the source of most ordering restrictions. Placing page templates towards the end of the list of definitions might be a good idea. A special case are stream definitions. These occur only as part of the corresponding page template definition and are listed at its end. So references to them will occur in the page template always before their definition. Finally, the definitions of page ranges always reference a page template and they should come after the page template definitions. For technical reasons explained in section 6.2, definitions of labels and outlines come last.

To avoid complex dependencies, an application can always choose not to use references in the definition section. There are only three types of nodes where references can not be avoided: fonts are referenced in glyph nodes, labels are referenced in outlines, and languages are referenced in boxes or page templates. Possible ordering restrictions can be satisfied if languages are defined early. To check the define before use policy, we use an array of bitvectors, but we limit checking to the first 256 references. We have for every reference number $N < 256$ and every kind K a single bit which is set if and only if the corresponding reference is defined.

⟨ definition checks 390 ⟩ ≡ (390)

```
uint32_t definition_bits[#100/32][32] = {{0}};
#define SET_DBIT(N, K)
  ((N) > #FF ? 1 : (definition_bits[N/32][K] |= (1 << ((N) & (32 − 1)))))
#define GET_DBIT(N, K)
  ((N) > #FF ? 1 : ((definition_bits[N/32][K] >> ((N) & (32 − 1))) & 1))
#define DEF(D, K, N) (D).k = K; (D).n = (N); SET_DBIT((D).n, (D).k);
  DBG(DBGDEF, "Defining␣%s␣%d\n", definition_name[(D).k], (D).n);
  RNG("Definition", (D).n, max_fixed[(D).k] + 1, max_ref[(D).k]);
#define REF(K, N) REF_RNG (K, N); if (¬GET_DBIT(N, K))
      QUIT("Reference␣%d␣to␣%s␣before␣definition", (N),
      definition_name[K])
```

Used in 553, 555, and 557.

⟨ initialize definitions 253 ⟩ +≡ (391)

```
definition_bits[0][list_kind] = (1 << (MAX_LIST_DEFAULT + 1)) − 1;
definition_bits[0][param_kind] = (1 << (MAX_LIST_DEFAULT + 1)) − 1;
```

$definition\_bits[0][int\_kind] = (1 \ll (\texttt{MAX\_INT\_DEFAULT} + 1)) - 1;$
$definition\_bits[0][dimen\_kind] = (1 \ll (\texttt{MAX\_DIMEN\_DEFAULT} + 1)) - 1;$
$definition\_bits[0][xdimen\_kind] = (1 \ll (\texttt{MAX\_XDIMEN\_DEFAULT} + 1)) - 1;$
$definition\_bits[0][glue\_kind] = (1 \ll (\texttt{MAX\_GLUE\_DEFAULT} + 1)) - 1;$
$definition\_bits[0][baseline\_kind] = (1 \ll (\texttt{MAX\_BASELINE\_DEFAULT} + 1)) - 1;$
$definition\_bits[0][page\_kind] = (1 \ll (\texttt{MAX\_PAGE\_DEFAULT} + 1)) - 1;$
$definition\_bits[0][stream\_kind] = (1 \ll (\texttt{MAX\_STREAM\_DEFAULT} + 1)) - 1;$
$definition\_bits[0][range\_kind] = (1 \ll (\texttt{MAX\_RANGE\_DEFAULT} + 1)) - 1;$
$definition\_bits[0][color\_kind] = (1 \ll (\texttt{MAX\_COLOR\_DEFAULT} + 1)) - 1;$

Reading the long format: − − − $\Longrightarrow$
Writing the short format: $\Longrightarrow \cdots$

⟨ symbols 2 ⟩ +≡ (392)
%**type** < *rf* > *def_node*

⟨ parsing rules 5 ⟩ +≡ (393)
def_node: *start* FONT *ref font* END
  { `DEF`($$, *font_kind*, $3); *hput_tags*($1, $4); }
| *start* INTEGER *ref integer* END
  { `DEF`($$, *int_kind*, $3); *hput_tags*($1, *hput_int*($4)); }
| *start* DIMEN *ref dimension* END
  { `DEF`($$, *dimen_kind*, $3); *hput_tags*($1, *hput_dimen*($4)); }
| *start* LANGUAGE *ref string* END
  { `DEF`($$, *language_kind*, $3); *hput_string*($4);
  hput_tags($1, `TAG`(*language_kind*, 0)); }
| *start* GLUE *ref glue* END
  { `DEF`($$, *glue_kind*, $3); *hput_tags*($1, *hput_glue*(&($4))); }
| *start* XDIMEN *ref xdimen* END
  { `DEF`($$, *xdimen_kind*, $3); *hput_tags*($1, *hput_xdimen*(&($4))); }
| *start* RULE *ref rule* END
  { `DEF`($$, *rule_kind*, $3); *hput_tags*($1, *hput_rule*(&($4))); }
| *start* LEADERS *ref leaders* END
  { `DEF`($$, *leaders_kind*, $3); *hput_tags*($1, `TAG`(*leaders_kind*, $4)); }
| *start* BASELINE *ref baseline* END
  { `DEF`($$, *baseline_kind*, $3); *hput_tags*($1, `TAG`(*baseline_kind*, $4)); }
| *start* LIGATURE *ref ligature* END
  { `DEF`($$, *ligature_kind*, $3); *hput_tags*($1, *hput_ligature*(&($4))); }
| *start* DISC *ref disc* END
  { `DEF`($$, *disc_kind*, $3); *hput_tags*($1, *hput_disc*(&($4))); }
| *start* IMAGE *ref image* END
  { `DEF`($$, *image_kind*, $3); *hput_tags*($1, `TAG`(*image_kind*, $4)); }
| *start* PARAM *ref parameters* END
  { `DEF`($$, *param_kind*, $3); *hput_tags*($1, *hput_list*($1 + 2, &($4))); }
| *start* PAGE *ref page* END
  { `DEF`($$, *page_kind*, $3); *hput_tags*($1, `TAG`(*page_kind*, 0)); };

There are a few cases where one wants to define a reference by a reference. For example, a HINT file may want to set the `parfillskip` glue to zero. While there are multiple ways to define the zero glue, the canonical way is a reference using the *zero_glue_no*. All these cases have in common that the reference to be defined is one of the default references and the defining reference is one of the fixed references. We add a few parsing rules and a testing macro for those cases where the number of default definitions is greater than the number of fixed definitions.

⟨ definition checks 390 ⟩ +≡ (394)
```
#define DEF_REF(D, K, M, N) DEF (D, K, M);
  if ((int)(M) > max_default[K])
    QUIT("Defining␣non␣default␣reference␣%d␣for␣%s", M,
        definition_name[K]);
  if ((int)(N) > max_fixed[K])
    QUIT("Defining␣reference␣%d␣for␣%s␣by␣non␣fixed␣reference␣%d", M,
        definition_name[K], N);
```

⟨ parsing rules 5 ⟩ +≡ (395)
```
  def_node: start INTEGER ref ref END
      { DEF_REF($$, int_kind, $3, $4); hput_tags($1, TAG(int_kind, 0)); }
    | start DIMEN ref ref END
      { DEF_REF($$, dimen_kind, $3, $4); hput_tags($1, TAG(dimen_kind, 0)); }
    | start GLUE ref ref END
      { DEF_REF($$, glue_kind, $3, $4); hput_tags($1, TAG(glue_kind, 0)); };
```

Reading the short format: $\cdots \Longrightarrow$
Writing the long format: $\Longrightarrow - - -$

⟨ get functions 18 ⟩ +≡ (396)
```
  void hget_definition(int n, Tag a, uint32_t node_pos)
  { switch (KIND(a)) {
    case font_kind: hget_font_def(n); break;
    case param_kind:
      { List l;
        l.t = a; HGET_LIST(INFO(a), l); hwrite_parameters(&l); break; }
    case page_kind: hget_page(); break;
    case dimen_kind: hget_dimen(a); break;
    case xdimen_kind:
      { Xdimen x; hget_xdimen(a, &x); hwrite_xdimen(&x); break; }
    case language_kind:
      if (INFO(a) ≠ b000)
        QUIT("Info␣value␣of␣language␣definition␣must␣be␣zero");
      else { char *n;
        HGET_STRING(n); hwrite_string(n);
      }
      break;
```

```
    case color_kind:
      switch (INFO(a)) { ⟨ cases to get definitions for color_kind 299 ⟩
      default:
        QUIT("Undefined␣tag␣%d␣for␣color_kind␣definition␣at␣0x%x",
            INFO(a), node_pos);
      }
      break;
    default: hget_content(a); break;
    }
  }
  void hget_def_node()
  { Kind k;
    ⟨ read the start byte a 16 ⟩
    k = KIND(a);
    if (k ≡ unknown_kind ∧ INFO(a) ≡ b100) hget_unknown_def();
    else if (k ≡ label_kind) hget_outline_or_label_def(INFO(a), node_pos);
    else { int n;
      n = HGET8;
      if (k ≠ range_kind) REF_RNG(k, n);
      SET_DBIT(n, k);
      if (k ≡ range_kind) hget_range(INFO(a), n);
      else { hwrite_start(); hwritef("%s␣*%d", definition_name[k], n);
        hget_definition(n, a, node_pos);
        hwrite_end();
      }
      if (n > max_ref[k] ∨ n ≤ max_fixed[k])
        QUIT("Definition␣%d␣for␣%s␣out␣of␣range␣[%d␣-␣%d]",
            n, definition_name[k], max_fixed[k] + 1, max_ref[k]);
      if (max_fixed[k] > max_default[k])
        QUIT("Definitions␣for␣kind␣%s␣not␣supported",
            definition_name[k]);
    }
    ⟨ read and check the end byte z 17 ⟩
  }
```

10.3 Parameter Lists

Because the content section is a "stateless" list of nodes, the definitions we see in the definition section can never change. It is however necessary to make occasionally local modifications of some of these definitions, because some definitions are parameters of the algorithms borrowed from TEX. Nodes that need such modifications, for example the paragraph nodes that are passed to TEX's line breaking algorithm, contain a list of local definitions called parameters. Typically sets of related parameters are needed. To facilitate a simple reference to such a set of parameters, we allow predefined parameter lists that can be referenced by a single number. The parameters of TEX's routines are quite basic—integers, dimensions, and glues—and all of them have default values. Therefore we restrict the definitions in parameter lists to such basic definitions.

⟨ parsing functions 386 ⟩ +≡ (397)

void *check_param_def* (*Ref* ∗ *df*)
{
  if (*df* →*k* ≠ *int_kind* ∧ *df* →*k* ≠ *dimen_kind* ∧
      df →*k* ≠ *glue_kind*)
    QUIT("Kind␣%s␣not␣allowed␣in␣parameter␣list",
      definition_name [*df* →*k*]);
  if (*df* →*n* ≤ *max_fixed* [*df* →*k*] ∨ *max_default* [*df* →*k*] < *df* →*n*)
    QUIT("Parameter␣%d␣for␣%s␣not␣allowed␣in␣parameter␣list", *df* →*n*,
      definition_name [*df* →*k*]);
}

The definitions below repeat the definitions we have seen for lists in section 4.1 with small modifications. For example we use the kind-value *param_kind*. An empty parameter list is omitted in the long format as well as in the short format.

Reading the long format: − − − ⟹
Writing the short format: ⟹ · · ·

⟨ symbols 2 ⟩ +≡ (398)

%**token** PARAM "param"
%**type** < *u* > *def_list*
%**type** < *l* > *parameters*

⟨ scanning rules 3 ⟩ +≡ (399)

param **return** PARAM;

⟨ parsing rules 5 ⟩ +≡ (400)

def_list: *position* | *def_list* *def_node* { *check_param_def* (&($2)); };

parameters: *estimate* *def_list* { $$.*p* = $2; $$.*t* = TAG(*param_kind* , *b001*);
  $$.*s* = (*hpos* − *hstart*) − $2; };

Using a parsing rule like "*param_list*: *start* PARAM *parameters* END", an empty parameter list will be written as "<param>". This looks ugly and seems like unnecessary syntax because the parser knows anyway that a parameter list will come next. Therefore the keyword can be omited except in definitions and in unknown nodes.

⟨ parsing rules 5 ⟩ +≡ (401)

named_param_list: *start* PARAM *parameters* END
  { *hput_tags* ($1, *hput_list* ($1 + 1, &($3))); };

param_list: *named_param_list*
  | *start* *parameters* END
  { *hput_tags* ($1, *hput_list* ($1 + 1, &($2))); };

Writing the long format: $\Longrightarrow - - -$

⟨ write functions 21 ⟩ +≡ (402)

```
void hwrite_parameters(List *l)
{ uint32_t h = hpos − hstart, e = hend − hstart;   /* save hpos and hend */
  hpos = l→p + hstart; hend = hpos + l→s;
  if (l→s > #FF) hwritef(" %d", l→s);
  while (hpos < hend) hget_def_node();
  hpos = hstart + h; hend = hstart + e;   /* restore hpos and hend */
}
void hwrite_param_list(List *l)
{ hwrite_start(); hwrite_parameters(l);
  hwrite_end();
}
void hwrite_named_param_list(List *l)
{ hwrite_start(); hwritef("param");
  hwrite_parameters(l);
  hwrite_end();
}
```

Reading the short format: $\cdots \Longrightarrow$

⟨ get functions 18 ⟩ +≡ (403)

```
void hget_param_list(List *l)
{ if (KIND(*hpos) ≠ param_kind)
    QUIT("Parameter list expected at 0x%x", (uint32_t)(hpos − hstart));
  else hget_list(l);
}
```

10.4 Fonts

Another definition that has no corresponding content node is the font definition. Fonts by themselves do not constitute content, instead they are used in glyph nodes. Further, fonts are never directly embedded in a content node; in a content node, a font is always specified by its font number. This limits the number of fonts that can be used in a HINT file to at most 256.

A long format font definition starts with the keyword "font" and is followed by the font number, as usual prefixed by an asterisk. Then comes the font specification with the font size, the font name, the section number of the TEX font metric file, and the section number of the file containing the glyphs for the font. The HINT format supports .pk files, the traditional font format for TEX, and the more modern PostScript Type 1 fonts, TrueType fonts, and OpenType fonts.

The format of font definitions will probably change in future versions of the HINT file format. For example, .pk files might be replaced entirely by PostScript Type 1 fonts. Also HINT needs the TEX font metric files only to obtain the sizes of characters when running TEX's line breaking algorithm. But for many TrueType fonts there are no TEX font metric files, while the necessary information about character sizes should be easy to obtain. Another information, that is currently missing from font definitions, is the fonts character encoding.

In a HINT file, text is represented as a sequence of numbers called character codes. HINT files use the UTF-8 character encoding scheme (CES) to map these numbers to their representation as byte sequences. For example the number "#`E4`" is encoded as the byte sequence "#`C3` #`A4`". The same number #`E4` now can represent different characters depending on the coded character set (CCS). For example in the common ISO-8859-1 (Latin 1) encoding the number #`E4` is the umlaut "ä" where as in the ISO-8859-7 (Latin/Greek) it is the Greek letter "δ" and in the EBCDIC encoding, used on IBM mainframes, it is the upper case letter "U".

The character encoding is irrelevant for rendering a HINT file as long as the character codes in the glyph nodes are consistent with the character codes used in the font file, but the character encoding is necessary for all programs that need to "understand" the content of the HINT file. For example programs that want to translate a HINT document to a different language, or for text-to-speech conversion.

The Internet Engineering Task Force IETF has established a character set registry[14] that defines an enumeration of all registered coded character sets[3]. The coded character set numbers are in the range 1–2999. This encoding number, as given in [4], might be one possibility for specifying the font encoding as part of a font definition.

Currently, it is only required that a font specifies an interword glue and a default discretionary break. After that comes a list of up to 12 font specific parameters.

The font size specifies the desired "at size" which might be different from the "design size" of the font as stored in the `.tfm` file.

In the short format, the font specification is given in the same order as in the long format.

Our internal representation of a font just stores the font name because in the long format we add the font name as a comment to glyph nodes.

⟨ common variables 252 ⟩ +≡ (404)

char $**$*hfont_name*; /* dynamically allocated array of font names */

⟨ hint basic types 6 ⟩ +≡ (405)

#**define** `MAX_FONT_PARAMS` 11

⟨ initialize definitions 253 ⟩ +≡ (406)

`ALLOCATE`(*hfont_name*, *max_ref*[*font_kind*] + 1, **char** $*$);

Reading the long format: $---\Longrightarrow$

⟨ symbols 2 ⟩ +≡ (407)
%**token** FONT "font"
%**type** < *info* > *font font_head*

⟨ scanning rules 3 ⟩ +≡ (408)
```
font                return FONT;
```

Note that we set the definition bit early because the definition of font f might involve glyphs that reference font f (or other fonts).

⟨ parsing rules 5 ⟩ +≡ (409)
```
font: font_head font_param_list;

font_head: string dimension UNSIGNED UNSIGNED
    { uint8_t f = $<u>0;
      SET_DBIT(f, font_kind); hfont_name[f] = strdup($1);
      $$ = hput_font_head(f, hfont_name[f], $2, $3, $4); };

font_param_list: glue_node disc_node | font_param_list font_param;

font_param:
    start PENALTY fref penalty END { hput_tags($1, hput_int($4)); }
  | start KERN fref kern END { hput_tags($1, hput_kern(&($4))); }
  | start LIGATURE fref ligature END { hput_tags($1, hput_ligature(&($4))); }
  | start DISC fref disc END { hput_tags($1, hput_disc(&($4))); }
  | start GLUE fref glue END { hput_tags($1, hput_glue(&($4))); }
  | start LANGUAGE fref string END { hput_string($4);
      hput_tags($1, TAG(language_kind, 0)); }
  | start RULE fref rule END { hput_tags($1, hput_rule(&($4))); }
  | start IMAGE fref image END { hput_tags($1, TAG(image_kind, $4)); };

fref: ref
    { RNG("Font parameter", $1, 0, MAX_FONT_PARAMS); };
```

Reading the short format: $\cdots\Longrightarrow$
Writing the long format: $\Longrightarrow---$

⟨ get functions 18 ⟩ +≡ (410)
```
static void hget_font_params(void)
{ Disc h;
  hget_glue_node();
  hget_disc_node(&(h)); hwrite_disc_node(&h);
  DBG(DBGDEF, "Start font parameters\n");
  while (KIND(*hpos) ≠ font_kind)
  { Ref df;
    ⟨ read the start byte a 16 ⟩
    df.k = KIND(a);
    df.n = HGET8;
```

```
    DBG(DBGDEF, "Reading␣font␣parameter␣%d:␣%s\n", df.n,
        definition_name[df.k]);
    if (df.k ≠ penalty_kind ∧ df.k ≠ kern_kind ∧ df.k ≠ ligature_kind ∧
          df.k ≠ disc_kind ∧ df.k ≠ glue_kind ∧ df.k ≠ language_kind ∧
          df.k ≠ rule_kind ∧ df.k ≠ image_kind)
      QUIT("Font␣parameter␣%d␣has␣invalid␣type␣%s", df.n,
          content_name[df.n]);
    RNG("Font␣parameter", df.n, 0, MAX_FONT_PARAMS);
    hwrite_start(); hwritef("%s␣*%d", content_name[KIND(a)], df.n);
    hget_definition(df.n, a, node_pos);
    hwrite_end();
    ⟨read and check the end byte z 17⟩
  }
  DBG(DBGDEF, "End␣font␣parameters\n");
}

void hget_font_def(uint8_t f)
{ char *n; Dimen s = 0; uint16_t m, y;
  HGET_STRING(n); hwrite_string(n); hfont_name[f] = strdup(n);
  HGET32(s); hwrite_dimension(s);
  DBG(DBGDEF, "Font␣%s␣size␣0x%x\n", n, s);
  HGET16(m); RNG("Font␣metrics", m, 3, max_section_no);
  HGET16(y); RNG("Font␣glyphs", y, 3, max_section_no);
  hwritef("␣%d␣%d", m, y);
  hget_font_params();
  DBG(DBGDEF, "End␣font␣definition\n");
}
```

Writing the short format: ⟹ · · ·

```
⟨put functions 14⟩ +≡                                                    (411)
  Tag hput_font_head(uint8_t f, char *n, Dimen s,
        uint16_t m, uint16_t y)
  { Info i = b000;
    DBG(DBGDEF, "Defining␣font␣%d␣(%s)␣size␣0x%x\n", f, n, s);
    hput_string(n);
    HPUT32(s); HPUT16(m); HPUT16(y);
    return TAG(font_kind, i);
  }
```

10.5 References

We have seen how to make definitions, now let's see how to reference them. In the long form, we can simply write the reference number, after the keyword like this: "`<glue *17>`". The asterisk is necessary to keep apart, for example, a penalty with value 50, written "`<penalty 50>`", from a penalty referencing the integer definition number 50, written "`<penalty *50>`".

⟨ hint types 1 ⟩ +≡ (412)

```
typedef struct { Kind k; int n; } Ref;
```

Reading the long format: − − − ⟹

Writing the short format: ⟹ · · ·

⟨ parsing rules 5 ⟩ +≡ (413)

```
xdimen_ref: ref { REF(xdimen_kind, $1); };
param_ref: ref { REF(param_kind, $1); };
stream_ref: ref { REF_RNG(stream_kind, $1); };
content_node: start PENALTY ref END
      { REF(penalty_kind, $3); hput_tags($1, TAG(penalty_kind, 0)); }
    | start KERN explicit ref END
      { REF(dimen_kind, $4); hput_tags($1, TAG(kern_kind, ($3) ? b100 : b000));
        }
    | start KERN explicit XDIMEN ref END
      { REF(xdimen_kind, $5);
        hput_tags($1, TAG(kern_kind, ($3) ? b101 : b001)); }
    | start GLUE ref END
      { REF(glue_kind, $3); hput_tags($1, TAG(glue_kind, 0)); }
    | start LIGATURE ref END
      { REF(ligature_kind, $3); hput_tags($1, TAG(ligature_kind, 0)); }
    | start DISC ref END
      { REF(disc_kind, $3); hput_tags($1, TAG(disc_kind, 0)); }
    | start RULE ref END
      { REF(rule_kind, $3); hput_tags($1, TAG(rule_kind, 0)); }
    | start IMAGE ref END
      { REF(image_kind, $3); hput_tags($1, TAG(image_kind, 0)); }
    | start LEADERS ref END
      { REF(leaders_kind, $3); hput_tags($1, TAG(leaders_kind, 0)); }
    | start BASELINE ref END
      { REF(baseline_kind, $3); hput_tags($1, TAG(baseline_kind, 0)); }
    | start LANGUAGE REFERENCE END
      { REF(language_kind, $3); hput_tags($1, hput_language($3)); };
glue_node: start GLUE ref END
      { REF(glue_kind, $3);
        if ($3 ≡ zero_skip_no) { hpos = hpos − 2; $$ = false; }
        else { hput_tags($1, TAG(glue_kind, 0)); $$ = true; }
      };
```

Reading the short format: ··· ⟹

⟨ cases to get content 20 ⟩ +≡ (414)

```
case TAG(penalty_kind, 0): HGET_REF(penalty_kind); break;
case TAG(kern_kind, b000): HGET_REF(dimen_kind); break;
case TAG(kern_kind, b100): hwritef("␣!"); HGET_REF(dimen_kind); break;
case TAG(kern_kind, b001):
  hwritef("␣xdimen"); HGET_REF(xdimen_kind); break;
case TAG(kern_kind, b101):
  hwritef("␣!␣xdimen"); HGET_REF(xdimen_kind); break;
case TAG(ligature_kind, 0): HGET_REF(ligature_kind); break;
case TAG(disc_kind, 0): HGET_REF(disc_kind); break;
case TAG(glue_kind, 0): HGET_REF(glue_kind); break;
case TAG(language_kind, b000): HGET_REF(language_kind); break;
case TAG(rule_kind, 0): HGET_REF(rule_kind); break;
case TAG(image_kind, 0): HGET_REF(image_kind); break;
case TAG(leaders_kind, 0): HGET_REF(leaders_kind); break;
case TAG(baseline_kind, 0): HGET_REF(baseline_kind); break;
```

⟨ get macros 19 ⟩ +≡ (415)

```
#define HGET_REF(K)
  { uint8_t n = HGET8; REF(K, n); hwrite_ref(n); }
```

Writing the long format: ⟹ − − −

⟨ write functions 21 ⟩ +≡ (416)

```
void hwrite_ref(int n)
{ hwritef("␣*%d", n); }

void hwrite_ref_node(Kind k, uint8_t n)
{ hwrite_start(); hwritef("%s", content_name[k]); hwrite_ref(n); hwrite_end();
}
```

11 Defaults

Several of the predefined values found in the definition section are used as parameters for the routines borrowed from TEX to display the content of a HINT file. These values must be defined, but it is inconvenient if the same standard definitions need to be placed in each and every HINT file. Therefore we specify in this chapter reasonable default values. As a consequence, even a HINT file without any definitions should produce sensible results when displayed.

The definitions that have default values are integers, dimensions, extended dimensions, glues, baselines, labels, page templates, streams, and page ranges. Each of these defaults has its own subsection below. Actually the defaults for extended dimensions, baselines, and labels are not needed by TEX's routines, but it is nice to have default values for the extended dimensions that represent `hsize`, `vsize`, a zero baseline skip, and a label for the table of content.

The array *max_default* contains for each kind-value the maximum number of the default values. The function *hset_max* is used to initialize them.

The programs `shrink` and `stretch` actually do not use the defaults, but it would be possible to suppress definitions if the defined value is the same as the default value. We start by setting $max\_default[k] \equiv -1$, meaning no defaults, and $max\_fixed[k] \equiv$ #`10000`, meaning no definitions. The following subsections will then overwrite these values for all kinds of definitions that have defaults. It remains to reset *max_fixed* to -1 for all those kinds that have no defaults but allow definitions.

⟨ take care of variables without defaults 417 ⟩ ≡ (417)

for $(k = 0;\ k < 32;\ k{+}{+})$ $max\_default[k] = -1, max\_fixed[k] =$ #`10000`;
$max\_fixed[font\_kind] = max\_fixed[ligature\_kind] = max\_fixed[disc\_kind]$
$= max\_fixed[language\_kind] = max\_fixed[rule\_kind] = max\_fixed[image\_kind]$
$= max\_fixed[leaders\_kind] = max\_fixed[param\_kind] = max\_fixed[label\_kind]$
$= -1;$

Used in 546.

11.1 Integers

Integers are very simple objects, and it might be tempting not to use predefined integers at all. But the TEX typesetting engine, which is used by HINT, uses many integer parameters to fine tune its operations. As we will see, all these integer parameters have a predefined integer number that refers to an integer definition.

Integers and penalties share the same kind-value. So a penalty node that references one of the predefined penalties, simply contains the integer number as a reference number.

The following integer numbers are predefined. The zero integer is fixed with integer number zero. The default values are taken from `plain.tex`.

⟨ default names 418 ⟩ ≡ (418)

typedef enum {
$zero\_int\_no = 0, pretolerance\_no = 1, tolerance\_no = 2, line\_penalty\_no = 3,$
$hyphen\_penalty\_no = 4, ex\_hyphen\_penalty\_no = 5, club\_penalty\_no = 6,$
$widow\_penalty\_no = 7, display\_widow\_penalty\_no = 8, broken\_penalty\_no = 9,$
$pre\_display\_penalty\_no = 10, post\_display\_penalty\_no = 11,$
$inter\_line\_penalty\_no = 12, double\_hyphen\_demerits\_no = 13,$
$final\_hyphen\_demerits\_no = 14, adj\_demerits\_no = 15, looseness\_no = 16,$
$time\_no = 17, day\_no = 18, month\_no = 19, year\_no = 20,$
$hang\_after\_no = 21, floating\_penalty\_no = 22$
} **Int_no**;
#**define** `MAX_INT_DEFAULT` *floating_penalty_no*

Used in 545.

⟨ define *int_defaults* 419 ⟩ ≡ (419)

$max\_default[int\_kind] =$ `MAX_INT_DEFAULT`;
$max\_fixed[int\_kind] = zero\_int\_no;$
$int\_defaults[zero\_int\_no] = 0;$
$int\_defaults[pretolerance\_no] = 100;$
$int\_defaults[tolerance\_no] = 200;$
$int\_defaults[line\_penalty\_no] = 10;$
$int\_defaults[hyphen\_penalty\_no] = 50;$
$int\_defaults[ex\_hyphen\_penalty\_no] = 50;$
$int\_defaults[club\_penalty\_no] = 150;$
$int\_defaults[widow\_penalty\_no] = 150;$
$int\_defaults[display\_widow\_penalty\_no] = 50;$
$int\_defaults[broken\_penalty\_no] = 100;$
$int\_defaults[pre\_display\_penalty\_no] = 10000;$
$int\_defaults[post\_display\_penalty\_no] = 0;$
$int\_defaults[inter\_line\_penalty\_no] = 0;$
$int\_defaults[double\_hyphen\_demerits\_no] = 10000;$
$int\_defaults[final\_hyphen\_demerits\_no] = 5000;$
$int\_defaults[adj\_demerits\_no] = 10000;$
$int\_defaults[looseness\_no] = 0;$
$int\_defaults[time\_no] = 720;$
$int\_defaults[day\_no] = 4;$
$int\_defaults[month\_no] = 7;$
$int\_defaults[year\_no] = 1776;$
$int\_defaults[hang\_after\_no] = 1;$
$int\_defaults[floating\_penalty\_no] = 20000;$

```
printf("int32_t␣int_defaults[MAX_INT_DEFAULT+1]={");
for (i = 0; i ≤ max_default[int_kind]; i++)
{ printf("%d", int_defaults[i]); if (i < max_default[int_kind]) printf(",␣"); }
printf("};\n\n");
```

Used in 546.

11.2 Dimensions

Notice that there are default values for the two dimensions `hsize` and `vsize`. These are the “design sizes” for the hint file. While it might not be possible to display the HINT file using these values of `hsize` and `vsize`, these are the author’s recommendation for the best “viewing experience”.

⟨ default names 418 ⟩ +≡ (420)

```
typedef enum {
    zero_dimen_no = 0, hsize_dimen_no = 1, vsize_dimen_no = 2,
    line_skip_limit_no = 3, max_depth_no = 4, split_max_depth_no = 5,
    hang_indent_no = 6, emergency_stretch_no = 7, quad_no = 8,
    math_quad_no = 9
} Dimen_no;
#define MAX_DIMEN_DEFAULT math_quad_no
```

⟨ define *dimen_defaults* 421 ⟩ ≡ (421)

```
max_default[dimen_kind] = MAX_DIMEN_DEFAULT;
max_fixed[dimen_kind] = zero_dimen_no;

dimen_defaults[zero_dimen_no] = 0;
dimen_defaults[hsize_dimen_no] = (Dimen)(6.5 * 72.27 * ONE);
dimen_defaults[vsize_dimen_no] = (Dimen)(8.9 * 72.27 * ONE);
dimen_defaults[line_skip_limit_no] = 0;
dimen_defaults[split_max_depth_no] = (Dimen)(3.5 * ONE);
dimen_defaults[hang_indent_no] = 0;
dimen_defaults[emergency_stretch_no] = 0;
dimen_defaults[quad_no] = 10 * ONE;
dimen_defaults[math_quad_no] = 10 * ONE;

printf("Dimen␣dimen_defaults[MAX_DIMEN_DEFAULT+1]={");
for (i = 0; i ≤ max_default[dimen_kind]; i++) {
    printf("0x%x", dimen_defaults[i]);
    if (i < max_default[dimen_kind]) printf(",␣");
}
printf("};\n\n");
```

Used in 546.

11.3 Extended Dimensions

Extended dimensions can be used in a variety of nodes for example kern and box nodes. We define three fixed extended dimensions: zero, hsize, and vsize. In contrast to the `hsize` and `vsize` dimensions defined in the previous section, the extended dimensions defined here are linear functions that always evaluate to the current horizontal and vertical size in the viewer.

⟨ default names 418 ⟩ +≡ (422)

typedef enum {
  zero_xdimen_no = 0, *hsize_xdimen_no* = 1, *vsize_xdimen_no* = 2
} **Xdimen_no**;
#**define** `MAX_XDIMEN_DEFAULT` *vsize_xdimen_no*

⟨ define *xdimen_defaults* 423 ⟩ ≡ (423)

max_default[*xdimen_kind*] = `MAX_XDIMEN_DEFAULT`;
max_fixed[*xdimen_kind*] = *vsize_xdimen_no*;
printf (`"Xdimen␣xdimen_defaults[MAX_XDIMEN_DEFAULT+1]={"`
`"{0x0,␣0.0,␣0.0},␣{0x0,␣1.0,␣0.0},␣{0x0,␣0.0,␣1.0}"`
`"};\n\n"`);

Used in 546.

11.4 Glue

There are predefined glue numbers that correspond to the skip parameters of TEX. The default values are taken from `plain.tex`.

⟨ default names 418 ⟩ +≡ (424)

typedef enum {
  zero_skip_no = 0, *fil_skip_no* = 1, *fill_skip_no* = 2, *line_skip_no* = 3,
  baseline_skip_no = 4, *above_display_skip_no* = 5, *below_display_skip_no* = 6,
  above_display_short_skip_no = 7, *below_display_short_skip_no* = 8,
  left_skip_no = 9, *right_skip_no* = 10, *top_skip_no* = 11, *split_top_skip_no* = 12,
  tab_skip_no = 13, *par_fill_skip_no* = 14
} **Glue_no**;
#**define** `MAX_GLUE_DEFAULT` *par_fill_skip_no*

⟨ define *glue_defaults* 425 ⟩ ≡ (425)

max_default[*glue_kind*] = `MAX_GLUE_DEFAULT`;
max_fixed[*glue_kind*] = *fill_skip_no*;
glue_defaults[*fil_skip_no*].*p*.*f* = 1.0;
glue_defaults[*fil_skip_no*].*p*.*o* = *fil_o*;
glue_defaults[*fill_skip_no*].*p*.*f* = 1.0;
glue_defaults[*fill_skip_no*].*p*.*o* = *fill_o*;
glue_defaults[*line_skip_no*].*w*.*w* = 1 ∗ `ONE`;
glue_defaults[*baseline_skip_no*].*w*.*w* = 12 ∗ `ONE`;
glue_defaults[*above_display_skip_no*].*w*.*w* = 12 ∗ `ONE`;
glue_defaults[*above_display_skip_no*].*p*.*f* = 3.0;
glue_defaults[*above_display_skip_no*].*p*.*o* = *normal_o*;

glue_defaults[*above_display_skip_no*].*m*.*f* = 9.0;
glue_defaults[*above_display_skip_no*].*m*.*o* = *normal_o*;
glue_defaults[*below_display_skip_no*].*w*.*w* = 12 ∗ `ONE`;
glue_defaults[*below_display_skip_no*].*p*.*f* = 3.0;
glue_defaults[*below_display_skip_no*].*p*.*o* = *normal_o*;
glue_defaults[*below_display_skip_no*].*m*.*f* = 9.0;
glue_defaults[*below_display_skip_no*].*m*.*o* = *normal_o*;
glue_defaults[*above_display_short_skip_no*].*p*.*f* = 3.0;
glue_defaults[*above_display_short_skip_no*].*p*.*o* = *normal_o*;
glue_defaults[*below_display_short_skip_no*].*w*.*w* = 7 ∗ `ONE`;
glue_defaults[*below_display_short_skip_no*].*p*.*f* = 3.0;
glue_defaults[*below_display_short_skip_no*].*p*.*o* = *normal_o*;
glue_defaults[*below_display_short_skip_no*].*m*.*f* = 4.0;
glue_defaults[*below_display_short_skip_no*].*m*.*o* = *normal_o*;
glue_defaults[*top_skip_no*].*w*.*w* = 10 ∗ `ONE`;
glue_defaults[*split_top_skip_no*].*w*.*w* = (**Dimen**) 8.5 ∗ `ONE`;
glue_defaults[*par_fill_skip_no*].*p*.*f* = 1.0;
glue_defaults[*par_fill_skip_no*].*p*.*o* = *fil_o*;
#**define** `PRINT_GLUE`(*G*) *printf*(`"{{0x%x,␣%f,␣%f},{%f,␣%d},{%f,␣%d}}"`, *G*.*w*.*w*, *G*.*w*.*h*, *G*.*w*.*v*, *G*.*p*.*f*, *G*.*p*.*o*, *G*.*m*.*f*, *G*.*m*.*o*)

printf(`"Glue␣glue_defaults[MAX_GLUE_DEFAULT+1]={\n"`);
for (*i* = 0; *i* ≤ *max_default*[*glue_kind*]; *i*++)
{ `PRINT_GLUE`(*glue_defaults*[*i*]); **if** (*i* < *max_default*[*int_kind*]) *printf*(`",\n"`);
}
printf(`"};\n\n"`);

Used in 546.

We fix the glue definition with number zero to be the “zero glue”: a glue with width zero and zero stretchability and shrinkability. Here is the reason: In the short format, the info bits of a glue node indicate which components of a glue are nonzero. Therefore the zero glue should have an info value of zero—which on the other hand is reserved for a reference to a glue definition. Hence, the best way to represent a zero glue is as a predefined glue.

11.5 Baseline Skips

The zero baseline which inserts no baseline skip is predefined.

⟨ default names 418 ⟩ +≡ (426)
typedef enum { *zero_baseline_no* = 0 } **Baseline_no**;
#**define** `MAX_BASELINE_DEFAULT` *zero_baseline_no*

⟨ define *baseline_defaults* 427 ⟩ ≡ (427)
max_default[*baseline_kind*] = `MAX_BASELINE_DEFAULT`;
max_fixed[*baseline_kind*] = *zero_baseline_no*;
{ **Baseline** *z* = {{{0}}};

```
  printf("Baseline␣baseline_defaults[MAX_BASELINE_DEFAULT+1]={{");
  PRINT_GLUE(z.bs); printf(",␣"); PRINT_GLUE(z.ls);
  printf(",␣0x%x}};\n\n", z.lsl);
}
```

Used in 546.

11.6 Labels

The zero label is predefined. It should point to the "home" position of the document which should be the position where a user can start reading or navigating the document. For a short document this is usually the start of the document, and hence, the default is the first position of the content section. For a larger document, the home position could point to the table of content where a reader will find links to other parts of the document.

⟨ default names 418 ⟩ +≡ (428)

```
  typedef enum { zero_label_no = 0 } Label_no;
#define MAX_LABEL_DEFAULT zero_label_no
```

⟨ define *label_defaults* 429 ⟩ ≡ (429)

```
  max_default[label_kind] = MAX_LABEL_DEFAULT;
  printf("Label␣label_defaults[MAX_LABEL_DEFAULT+1]="
      "{{0,0,LABEL_TOP,true,0,0}};\n\n");
```

Used in 546.

11.7 Streams

The zero stream is predefined for the main content.

⟨ default names 418 ⟩ +≡ (430)

```
  typedef enum { zero_stream_no = 0 } Stream_no;
#define MAX_STREAM_DEFAULT zero_stream_no
```

⟨ define stream defaults 431 ⟩ ≡ (431)

```
  max_default[stream_kind] = MAX_STREAM_DEFAULT;
  max_fixed[stream_kind] = zero_stream_no;
```

Used in 546.

11.8 Page Templates

The zero page template is a predefined, built-in page template.

⟨ default names 418 ⟩ +≡ (432)

```
  typedef enum { zero_page_no = 0 } Page_no;
#define MAX_PAGE_DEFAULT zero_page_no
```

⟨ define page defaults 433 ⟩ ≡ (433)

```
  max_default[page_kind] = MAX_PAGE_DEFAULT;
  max_fixed[page_kind] = zero_page_no;
```

Used in 546.

11.9 Page Ranges

The page range for the zero page template is the entire content section.

⟨ default names 418 ⟩ +≡ (434)

typedef enum { *zero_range_no* = 0 } **Range_no**;
#define `MAX_RANGE_DEFAULT` *zero_range_no*

⟨ define range defaults 435 ⟩ ≡ (435)

max_default[*range_kind*] = `MAX_RANGE_DEFAULT`;
max_fixed[*range_kind*] = *zero_range_no*;

Used in 546.

11.10 List, Texts, and Parameters

⟨ default names 418 ⟩ +≡ (436)

typedef enum { *empty_list_no* = 0 } **List_no**;
#define `MAX_LIST_DEFAULT` *empty_list_no*

⟨ define range defaults 435 ⟩ +≡ (437)

max_default[*list_kind*] = `MAX_LIST_DEFAULT`;
max_fixed[*list_kind*] = *empty_list_no*;
max_default[*param_kind*] = `MAX_LIST_DEFAULT`;
max_fixed[*param_kind*] = *empty_list_no*;

11.11 Colors

⟨ default names 418 ⟩ +≡ (438)

typedef enum { *zero_color_no* = 0, *link_color_no* = 1 } **Color_no**;
#define `MAX_COLOR_DEFAULT` *link_color_no*

The default colors for day mode are black on white, red on white, and green on white; the links in day mode are blue. In night mode the background becomes black, the normal text white and the other colors become slightly lighter.

We store the default color set using an byte array in RGBA format for colors; we combine a pair of colors for foreground and background in an array; we combine three pairs for normal, mark, and focus text in an array; and we define a color set as two such pairs, one for day and one for night mode to define the default colors.

⟨ define *color_defaults* 439 ⟩ ≡ (439)

```
max_default[color_kind] = MAX_COLOR_DEFAULT;
max_fixed[color_kind] = −1;
printf("ColorSet␣␣color_defaults[MAX_COLOR_DEFAULT+1]=\n"
    "{{0x000000FF,␣0xFFFFFF00,\n"        /* black on white */
  "␣␣0xEE0000FF,␣0xFFFFFF00,\n"          /* dark red */
  "␣␣0x00EE00FF,␣0xFFFFFF00,\n"          /* dark green */
  "␣␣0xFFFFFFFF,␣0x00000000,"            /* white on black */
  "␣␣0xFF1111FF,␣0x00000000,\n"          /* light red */
  "␣␣0x11FF11FF,␣0x00000000},\n"         /* light green */
  "␣{0x0000EEFF,␣0xFFFFFF00,\n"          /* dark blue on white */
```

```
"␣␣0xEE0000FF,␣0xFFFFFF00,\n"    /* dark red on white */
"␣␣0x00EE00FF,␣0xFFFFFF00,\n"    /* dark green on white */
"␣␣0x1111FFFF,␣0x00000000,\n"    /* light blue on black */
"␣␣0xFF1111FF,␣0x00000000,\n"    /* light red on black */
"␣␣0x11FF11FF,␣0x00000000\n"     /* light green on black */
"}};\n\n");
```

Used in 546.

12 Content Section

The content section is just a list of nodes. Within the `shrink` program, reading a node in long format will trigger writing the node in short format. Similarly within the `stretch` program, reading a node in short form will cause writing it in long format. As a consequence, the main task of writing the content section in long format is accomplished by calling *get_content* and writing it in the short format is accomplished by parsing the *content_list*.

Reading the Long Format: − − − ⟹

⟨ symbols 2 ⟩ +≡ (440)

```
%token CONTENT "content"
```

⟨ scanning rules 3 ⟩ +≡ (441)

```
content        return CONTENT;
```

⟨ parsing rules 5 ⟩ +≡ (442)

```
content_section: START CONTENT { hput_content_start(); }
        content_list END
    { hput_content_end(); hput_range_defs(); hput_label_defs(); };
```

Writing the Long Format: ⟹ − − −

⟨ write functions 21 ⟩ +≡ (443)

```
void hwrite_content_section(void)
{ section_no = 2;
  hwritef("<content");
  hsort_ranges();
  hsort_labels();
  hget_content_section();
  hwritef("\n>\n");
}
```

Reading the Short Format: $\cdots \Longrightarrow$

⟨ get functions 18 ⟩ +≡ (444)

```
void hget_content_section( )
{ DBG(DBGBASIC | DBGDIR, "Content\n");
  hget_section(2);
  hwrite_range( );
  hwrite_label( );
  while (hpos < hend) hget_content_node( );
}
```

Writing the Short Format: $\Longrightarrow \cdots$

⟨ put functions 14 ⟩ +≡ (445)

```
void hput_content_start(void)
{ DBG(DBGDIR, "Content␣Section\n");
  section_no = 2;
  hpos0 = hpos = hstart = dir[2].buffer;
  hend = hstart + dir[2].bsize;
}
void hput_content_end(void)
{ dir[2].size = hpos − hstart;                /* Updating the directory entry */
  DBG(DBGDIR, "End␣Content␣Section,␣size=0x%x\n", dir[2].size);
}
```

13 Processing the Command Line

The following code explains the command line parameters and options. It tells us what to expect in the rest of this section.

⟨ explain usage 446 ⟩ ≡ (446)

```
  fprintf(stdout, "Usage: %s [OPTION]... FILENAME%s\n", prog_name, in_ext);
  fprintf(stdout, DESCRIPTION);
  fprintf(stdout, "\nOptions:\n"
  "\t --help \t display this message\n"
  "\t --version\t display the HINT version\n"
  "\t -l     \t redirect stderr to a log file\n"
#if defined (STRETCH) ∨ defined (SHRINK)
  "\t -o FILE\t specify an output file name\n"
#endif
#if defined (STRETCH)
  "\t -a     \t write auxiliary files\n"
  "\t -g     \t do not use localized names (implies -a)\n"
  "\t -f     \t force overwriting existing auxiliary files\n"
  "\t -u     \t enable writing utf8 character codes\n"
  "\t -x     \t enable writing hexadecimal character codes\n"
#elif defined (SHRINK)
  "\t -a     \t use only localized names\n"
  "\t -g     \t do not use localized names\n"
  "\t -c     \t enable compression\n"
#endif
  );
#ifdef DEBUG
  fprintf(stdout, "\t -d XXXX \t set debug flag to hexadec\
      imal value XXXX.\n""\t\t\t OR together these values:\n");
  fprintf(stdout, "\t\t\t XX=%03X   basic debugging\n", DBGBASIC);
  fprintf(stdout, "\t\t\t XX=%03X   tag debugging\n", DBGTAGS);
  fprintf(stdout, "\t\t\t XX=%03X   node debugging\n", DBGNODE);
  fprintf(stdout, "\t\t\t XX=%03X   definition debugging\n", DBGDEF);
  fprintf(stdout, "\t\t\t XX=%03X   directory debugging\n", DBGDIR);
  fprintf(stdout, "\t\t\t XX=%03X   range debugging\n", DBGRANGE);
  fprintf(stdout, "\t\t\t XX=%03X   float debugging\n", DBGFLOAT);
  fprintf(stdout, "\t\t\t XX=%03X   compression debugging\n",
      DBGCOMPRESS);
```

```
  fprintf(stdout,"\t\t\t XX=%03X   buffer debugging\n",DBGBUFFER);
  fprintf(stdout,"\t\t\t XX=%03X   flex debugging\n",DBGFLEX);
  fprintf(stdout,"\t\t\t XX=%03X   bison debugging\n",DBGBISON);
  fprintf(stdout,"\t\t\t XX=%03X   TeX debugging\n",DBGTEX);
  fprintf(stdout,"\t\t\t XX=%03X   Page debugging\n",DBGPAGE);
  fprintf(stdout,"\t\t\t XX=%03X   Font debugging\n",DBGFONT);
  fprintf(stdout,"\t\t\t XX=%03X   Render debugging\n",DBGRENDER);
  fprintf(stdout,"\t\t\t XX=%03X   Label debugging\n",DBGLABEL);
#endif
```

Used in 450.

We define constants for different debug flags.

⟨ debug constants 447 ⟩ ≡ (447)

```
#define DBGNONE #0
#define DBGBASIC #1
#define DBGTAGS #2
#define DBGNODE #4
#define DBGDEF #8
#define DBGDIR #10
#define DBGRANGE #20
#define DBGFLOAT #40
#define DBGCOMPRESS #80
#define DBGBUFFER #100
#define DBGFLEX #200
#define DBGBISON #400
#define DBGTEX #800
#define DBGPAGE #1000
#define DBGFONT #2000
#define DBGRENDER #4000
#define DBGLABEL #8000
```

Used in 545.

Next we define common variables that are needed in all three programs defined here.

⟨ common variables 252 ⟩ +≡ (448)

```
  unsigned int debugflags = DBGNONE;
  int option_utf8 = false;
  int option_hex = false;
  int option_force = false;
  int option_global = false;
  int option_aux = false;
  int option_compress = false;
  char *stem_name = NULL;
  int stem_length = 0;
```

The variable *stem_name* contains the name of the input file not including the extension. The space allocated for it is large enough to append an extension with up to five characters. It can be used with the extension `.log` for the log file, with `.hint` or `.hnt` for the output file, and with `.abs` or `.rel` when writing or reading the auxiliary sections. The `stretch` program will overwrite the *stem_name* using the name of the output file if it is set with the `-o` option.

Next are the variables that are local in the *main* program.

⟨ local variables in *main* 449 ⟩ ≡ (449)

```
   char *prog_name;
   char *in_ext;
   char *out_ext;
   int option_log = false;
#ifndef SKIP
   char *file_name = NULL;
   int file_name_length = 0;
#endif
```

Used in 554, 555, and 557.

Processing the command line looks for options and then sets the input file name. For compatibility with GNU standards, the long options `--help` and `--version` are supported in addition to the short options.

⟨ process the command line 450 ⟩ ≡ (450)

```
   debugflags = DBGBASIC;
   prog_name = argv[0];
   if (argc < 2)
     { fprintf(stderr, "%s: no input file given\n" "Try '%s --help' for\
          more information\n", prog_name, prog_name);
     exit(1);
   }
   argv++;                                          /* skip the program name */
   while (*argv ≠ NULL) {
     if ((*argv)[0] ≡ '-') { char option = (*argv)[1];
       switch (option) {
       case '-':
         if (strcmp(*argv, "--version") ≡ 0) { fprintf(stderr,
               "%s version "HINT_VERSION_STRING"\n", prog_name);
           exit(0);
         }
         else if (strcmp(*argv, "--help") ≡ 0) { ⟨ explain usage 446 ⟩
           fprintf(stdout, "\nFor further information and reporting\
               bugs see https://hint.userweb.mwn.de/\n");
           exit(0);
         }
       case 'l': option_log = true; break;
#if defined (STRETCH) ∨ defined (SHRINK)
       case 'o': argv++;
         file_name_length = (int) strlen(*argv);
```

```
      ALLOCATE(file_name, file_name_length + 6, char);      /* plus extension */
      strcpy(file_name, *argv); break;
    case 'g': option_global = option_aux = true; break;
    case 'a': option_aux = true; break;
#endif
#if defined (STRETCH)
    case 'u': option_utf8 = true; break;
    case 'x': option_hex = true; break;
    case 'f': option_force = true; break;
#elif defined (SHRINK)
    case 'c': option_compress = true; break;
#endif
    case 'd':
      argv++;
      if (*argv ≡ NULL) { fprintf(stderr, "%s:␣option␣-d␣expec\
            ts␣an␣argument\n""Try␣'%s␣--help'␣for\
            ␣more␣information\n", prog_name, prog_name);
        exit(1);
      }
      debugflags = strtol(*argv, NULL, 16);
      break;
    default:
      { fprintf(stderr,
            "%s:␣unrecognized␣option␣'%s'\n""Try␣'%s␣--help'␣for\
            ␣more␣information\n", prog_name, *argv, prog_name);
        exit(1);
      }
    }
  }
  else                                                    /* the input file name */
  { int path_length = (int) strlen(*argv);
    int ext_length = (int) strlen(in_ext);

    ALLOCATE(hin_name, path_length + ext_length + 1, char);
    strcpy(hin_name, *argv);
    if (path_length < ext_length ∨ strncmp(hin_name + path_length − ext_length,
           in_ext, ext_length) ≠ 0) { strcat(hin_name, in_ext);
      path_length += ext_length;
    }
    stem_length = path_length − ext_length;
    ALLOCATE(stem_name, stem_length + 6, char);
    strncpy(stem_name, hin_name, stem_length);
    stem_name[stem_length] = 0;
    if (*(argv + 1) ≠ NULL)
      { fprintf(stderr, "%s:␣extra␣argument␣after␣input␣file␣nam\
```

```
          e:␣␣'%s'\n""Try␣'%s␣--help'␣for␣more␣information\n",
          prog_name, *(argv + 1), prog_name);
        exit(1);
      }
    }
    argv++;
  }
  if (hin_name ≡ NULL) { fprintf(stderr, "%s:␣missing␣input␣f\
      ile␣name\n""Try␣'%s␣--help'␣for␣more␣information\n",
      prog_name, prog_name);
    exit(1);
  }
```

Used in 554, 555, and 557.

After the command line has been processed, three file streams need to be opened: The input file *hin* and the output file *hout*. Further we need a log file *hlog* if debugging is enabled. For technical reasons, the scanner generated by `flex` needs an input file *yyin* which is set to *hin* and an output file *yyout* (which is not used).

⟨ common variables 252 ⟩ +≡ (451)

```
FILE *hin = NULL, *hout = NULL, *hlog = NULL;
```

The log file is opened first because this is the place where error messages should go while the other files are opened. It inherits its name from the input file name.

⟨ open the log file 452 ⟩ ≡ (452)

```
if (option_log) { strcat(stem_name, ".log");
  hlog = freopen(stem_name, "w", stderr);
  if (hlog ≡ NULL) {
    fprintf(stderr, "Unable␣to␣open␣logfile␣%s", stem_name);
    hlog = stderr;
  }
  stem_name[stem_length] = 0;
}
else hlog = stderr;
```

Used in 554, 555, and 557.

Once we have established logging, we can try to open the other files.

⟨ open the input file 453 ⟩ ≡ (453)

```
hin = fopen(hin_name, "rb");
if (hin ≡ NULL) QUIT("Unable␣to␣open␣input␣file␣%s", hin_name);
```

Used in 554.

⟨ open the output file 454 ⟩ ≡ (454)

```
if (file_name ≠ NULL) { int ext_length = (int) strlen(out_ext);
  if (file_name_length ≤ ext_length ∨ strncmp(file_name + file_name_length −
        ext_length, out_ext, ext_length) ≠ 0) { strcat(file_name, out_ext);
    file_name_length += ext_length;
  }
```

```
  }
  else { file_name_length = stem_length + (int) strlen(out_ext);
    ALLOCATE(file_name, file_name_length + 1, char);
    strcpy(file_name, stem_name); strcpy(file_name + stem_length, out_ext);
  }
  { char *aux_name = file_name;
    ⟨ make sure the path in aux_name exists 365 ⟩
    aux_name = NULL;
  }
  hout = fopen(file_name, "wb");
  if (hout ≡ NULL) QUIT("Unable␣to␣open␣output␣file␣%s", file_name);
```

Used in 554 and 555.

The `stretch` program will replace the *stem_name* using the stem of the output file.

⟨ determine the *stem_name* from the output *file_name* 455 ⟩ ≡ (455)

```
  stem_length = file_name_length − (int) strlen(out_ext);
  ALLOCATE(stem_name, stem_length + 6, char);
  strncpy(stem_name, file_name, stem_length);
  stem_name[stem_length] = 0;
```

Used in 555.

At the very end, we will close the files again.

⟨ close the input file 456 ⟩ ≡ (456)

```
  if (hin_name ≠ NULL) free(hin_name);
  if (hin ≠ NULL) fclose(hin);
```

Used in 554.

⟨ close the output file 457 ⟩ ≡ (457)

```
  if (file_name ≠ NULL) free(file_name);
  if (hout ≠ NULL) fclose(hout);
```

Used in 554 and 555.

⟨ close the log file 458 ⟩ ≡ (458)

```
  if (hlog ≠ NULL) fclose(hlog);
  if (stem_name ≠ NULL) free(stem_name);
```

Used in 554, 555, and 557.

14 Error Handling and Debugging

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

⟨ error.h 459 ⟩ ≡ (459)

```
#ifndef _ERROR_H
#define _ERROR_H
#include <stdlib.h>
#include <stdio.h>
  extern FILE *hlog;
  extern uint8_t *hpos, *hstart;
#ifndef LOG_PREFIX
#define LOG_PREFIX "HINT␣"
#endif
#define LOG(...) (fprintf(hlog, LOG_PREFIX__VA_ARGS__), fflush(hlog))
#define MESSAGE(...) (fprintf(hlog, LOG_PREFIX__VA_ARGS__), fflush(hlog))
#define QUIT(...)
  (MESSAGE("ERROR:␣"__VA_ARGS__), fprintf(hlog, "\n"), exit(1))
#endif
```

The amount of debugging depends on the debugging flags. For portability, we first define the output specifier for expressions of type **size_t**.

⟨ debug macros 381 ⟩ +≡ (460)

```
#ifdef WIN32
#define SIZE_F "0x%tx"
#else
#define SIZE_F "0x%tx"
#endif
#ifdef DEBUG
#define DBG(FLAGS, ...) ((debugflags & (FLAGS)) ? LOG(__VA_ARGS__) : 0)
#else
#define DBG(FLAGS, ...)(void) 0
#endif
#define DBGTAG(A, P) DBG(DBGTAGS, "tag␣[%s,%d]␣at␣"SIZE_F"\n",
      NAME(A), INFO(A), (P) − hstart)
#define RNG(S, N, A, Z)
  if ((int)(N) < (int)(A) ∨ (int)(N) > (int)(Z))
     QUIT(S "␣%d␣out␣of␣range␣[%d␣-␣%d]", N, A, Z)
```

```
#define TAGERR(A) QUIT("Unknown␣tag␣[%s,%d]␣at␣"SIZE_F"\n", NAME(A),
        INFO(A), hpos − hstart)
```

The `bison` generated parser will need a function *yyerror* for error reporting. We can define it now:

⟨ parsing functions 386 ⟩ +≡ (461)

```
  extern int yylineno;
  int yyerror(const char *msg)
  { QUIT("␣in␣line␣%d␣%s", yylineno, msg);
    return 0;
  }
```

To enable the generation of debugging code `bison` needs also the following:

⟨ enable bison debugging 462 ⟩ ≡ (462)

```
#ifdef DEBUG
#define YYDEBUG 1
  extern int yydebug;
#else
#define YYDEBUG 0
#endif
```

Used in 552, 553, and 554.

Appendix

A Traversing Short Format Files

For applications like searching or repositioning a file after reloading a possibly changed version of a file, it is useful to have a fast way of getting from one content node to the next. For quite some nodes, it is possible to know the size of the node from the tag. So the fastest way to get to the next node is looking up the node size in a table.

Other important nodes, for example hbox, vbox, or par nodes, end with a list node and it is possible to know the size of the node up to the final list. With that knowledge it is possible to skip the initial part of the node, then skip the list, and finally skip the tag byte. The size of the initial part can be stored in the same node size table using negated values. What works for lists, of course, will work for other kinds of nodes as well. So we use the lowest two bits of the values in the size table to store the number of embedded nodes that follow after the initial part. To combine the number of leading bytes and the number of trailing nodes into a single number that encodes both values according to this formula we use the macro `NODE_SIZE`. We can get back both values using the macros `NODE_HEAD` and `NODE_TAIL`.

⟨ hint macros 13 ⟩ +≡ (463)
#**define** `NODE_SIZE` (H, T) $((T) \equiv 0\ ?\ (\mathrm{H}) + 2 : -4 * ((\mathrm{H}) + 1) + ((T) - 1))$
#**define** `NODE_HEAD`(N) $((N) > 0\ ?\ (N) - 2 : -((N) \gg 2) - 1)$
#**define** `NODE_TAIL`(N) $((N) < 0\ ?\ ((N)\ \&\ {}^{\#}\mathtt{3}) + 1 : 0)$

For list nodes neither of these methods works and these nodes can be marked with a zero entry in the node size table.

This leads to the following code for a "fast forward" function for *hpos*:

⟨ shared skip functions 464 ⟩ ≡ (464)
  uint32_t *hff_list_pos* $= 0$, *hff_list_size* $= 0$;
  Tag *hff_tag*;

```
void hff_hpos(void)
{ signed char i, b, n;
  hff_tag = *hpos; DBGTAG(hff_tag, hpos);
  i = hnode_size[hff_tag];
  if (i > 0) { hpos = hpos + NODE_HEAD(i) + 2; return; }
  else if (i < 0) { n = NODE_TAIL(i); b = NODE_HEAD(i);
    hpos = hpos + 1 + b;                          /* skip initial part */
    while (n > 0) { hff_hpos(); n--; }            /* skip trailing nodes */
    hpos++;                                       /* skip end byte */
    return;
  }
  else if (hff_tag ≤ TAG(param_kind, 7)) ⟨ advance hpos over a list 467 ⟩
  TAGERR(hff_tag);
}
```

Used in 549 and 557.

We will put the *hnode_size* variable into the `tables.c` file using the following function. We add some comments and split negative values into their components, to make the result more readable.

⟨ print the *hnode_size* variable 465 ⟩ ≡ (465)

```
printf("signed␣char␣hnode_size[0x100]=␣{\n");
for (i = 0; i ≤ #ff; i++)
{ signed char s = hnode_size[i];
  if (s ≥ 0) printf("%d", s);
  else printf("-4*%d+%d", -(s >> 2), s & 3);
  if (i < #ff) printf(",");
  else printf("};");
  if ((i & #7) ≡ #7) printf("␣/*␣%s␣*/\n", content_name[KIND(i)]);
}
printf("\n\n");
```

Used in 546.

When dealing with unknown content nodes, it is convenient to know which nodes are known and which are not. For this purpose the *content_known* array contains one byte for each kind value and each such bytes will indicate using the seven least significant bits for which info values the corresponding nodes are known.

⟨ print the *content_known* variable 466 ⟩ ≡ (466)

```
for (k = 0; k < 32; k++)
  for (i = 0; i < 8; i++)
    if (hnode_size[TAG(k, i)] ≠ 0) content_known[k] |= (1 << i);
printf("uint8_t␣content_known[32]=␣{\n");
for (k = 0; k < 32; k++)
{ printf("0x%02X", content_known[k]);
  if (k < 31) printf(",");
  else printf("};");
  printf("␣/*␣%s␣*/\n", content_name[k]);
```

```
  }
  printf("\n");
```

Used in 546.

A.1 Lists

List don't follow the usual schema of nodes. They have a variable size that is stored in the node. We keep position and size in global variables so that the list that ends a node can be conveniently located.

⟨ advance *hpos* over a list 467 ⟩ ≡ (467)

```
switch (INFO(hff_tag) & #3) {
case 0: hff_list_pos = hpos − hstart + 1;
   hff_list_size = 0;
   hpos = hpos + 3; return;
case 1: hpos++; hff_list_size = HGET8; hff_list_pos = hpos − hstart + 1;
   hpos = hpos + 1 + hff_list_size + 1 + 1 + 1; return;
case 2: hpos++; HGET16(hff_list_size); hff_list_pos = hpos − hstart + 1;
   hpos = hpos + 1 + hff_list_size + 1 + 2 + 1; return;
case 3: hpos++; HGET32(hff_list_size); hff_list_pos = hpos − hstart + 1;
   hpos = hpos + 1 + hff_list_size + 1 + 4 + 1; return;
default: QUIT("List␣with␣unknown␣info␣[%s,%d]␣at␣"SIZE_F"\n",
         NAME(hff_tag), INFO(hff_tag), hpos − hstart);
}
```

Used in 464.

Actually list nodes never occur as content nodes in their own right but only as subnodes of content nodes.

Now let's consider the different kinds of nodes.

A.2 Glyphs

We start with the glyph nodes. All glyph nodes have a start and an end tag, one byte for the font, and depending on the info from 1 to 4 bytes for the character code.

⟨ initialize the *hnode_size* array 468 ⟩ ≡ (468)

```
hnode_size[TAG(glyph_kind, 1)] = NODE_SIZE(1 + 1, 0);
hnode_size[TAG(glyph_kind, 2)] = NODE_SIZE(1 + 2, 0);
hnode_size[TAG(glyph_kind, 3)] = NODE_SIZE(1 + 3, 0);
hnode_size[TAG(glyph_kind, 4)] = NODE_SIZE(1 + 4, 0);
```

Used in 546.

A.3 Penalties

Penalty nodes either contain a one byte reference, a one byte number, or a two byte number.

⟨ initialize the *hnode_size* array 468 ⟩ +≡ (469)
$hnode\_size[\mathtt{TAG}(penalty\_kind, 0)] = \mathtt{NODE\_SIZE}(1, 0);$
$hnode\_size[\mathtt{TAG}(penalty\_kind, 1)] = \mathtt{NODE\_SIZE}(1, 0);$
$hnode\_size[\mathtt{TAG}(penalty\_kind, 2)] = \mathtt{NODE\_SIZE}(2, 0);$
$hnode\_size[\mathtt{TAG}(penalty\_kind, 3)] = \mathtt{NODE\_SIZE}(4, 0);$

A.4 Kerns

Kern nodes can contain a reference (either to a dimension or an extended dimension) followed by either a dimension or an extended dimension node.

⟨ initialize the *hnode_size* array 468 ⟩ +≡ (470)
$hnode\_size[\mathtt{TAG}(kern\_kind, b000)] = \mathtt{NODE\_SIZE}(1, 0);$
$hnode\_size[\mathtt{TAG}(kern\_kind, b001)] = \mathtt{NODE\_SIZE}(1, 0);$
$hnode\_size[\mathtt{TAG}(kern\_kind, b010)] = \mathtt{NODE\_SIZE}(4, 0);$
$hnode\_size[\mathtt{TAG}(kern\_kind, b011)] = \mathtt{NODE\_SIZE}(0, 1);$
$hnode\_size[\mathtt{TAG}(kern\_kind, b100)] = \mathtt{NODE\_SIZE}(1, 0);$
$hnode\_size[\mathtt{TAG}(kern\_kind, b101)] = \mathtt{NODE\_SIZE}(1, 0);$
$hnode\_size[\mathtt{TAG}(kern\_kind, b110)] = \mathtt{NODE\_SIZE}(4, 0);$
$hnode\_size[\mathtt{TAG}(kern\_kind, b111)] = \mathtt{NODE\_SIZE}(0, 1);$

A.5 Extended Dimensions

Extended dimensions contain either one two or three 4 byte values depending on the info bits.

⟨ initialize the *hnode_size* array 468 ⟩ +≡ (471)
$hnode\_size[\mathtt{TAG}(xdimen\_kind, b100)] = \mathtt{NODE\_SIZE}(4, 0);$
$hnode\_size[\mathtt{TAG}(xdimen\_kind, b010)] = \mathtt{NODE\_SIZE}(4, 0);$
$hnode\_size[\mathtt{TAG}(xdimen\_kind, b001)] = \mathtt{NODE\_SIZE}(4, 0);$
$hnode\_size[\mathtt{TAG}(xdimen\_kind, b110)] = \mathtt{NODE\_SIZE}(4 + 4, 0);$
$hnode\_size[\mathtt{TAG}(xdimen\_kind, b101)] = \mathtt{NODE\_SIZE}(4 + 4, 0);$
$hnode\_size[\mathtt{TAG}(xdimen\_kind, b011)] = \mathtt{NODE\_SIZE}(4 + 4, 0);$
$hnode\_size[\mathtt{TAG}(xdimen\_kind, b111)] = \mathtt{NODE\_SIZE}(4 + 4 + 4, 0);$

A.6 Language

Language nodes either code the language in the info value or they contain a reference byte.

⟨ initialize the *hnode_size* array 468 ⟩ +≡ (472)
$hnode\_size[\mathtt{TAG}(language\_kind, b000)] = \mathtt{NODE\_SIZE}(1, 0);$
$hnode\_size[\mathtt{TAG}(language\_kind, 1)] = \mathtt{NODE\_SIZE}(0, 0);$
$hnode\_size[\mathtt{TAG}(language\_kind, 2)] = \mathtt{NODE\_SIZE}(0, 0);$
$hnode\_size[\mathtt{TAG}(language\_kind, 3)] = \mathtt{NODE\_SIZE}(0, 0);$
$hnode\_size[\mathtt{TAG}(language\_kind, 4)] = \mathtt{NODE\_SIZE}(0, 0);$

$hnode\_size[\texttt{TAG}(language\_kind, 5)] = \texttt{NODE\_SIZE}(0, 0);$
$hnode\_size[\texttt{TAG}(language\_kind, 6)] = \texttt{NODE\_SIZE}(0, 0);$
$hnode\_size[\texttt{TAG}(language\_kind, 7)] = \texttt{NODE\_SIZE}(0, 0);$

A.7 Rules

Rules usually contain a reference, otherwise they contain either one, two, or three 4 byte values depending on the info bits.

⟨ initialize the *hnode_size* array 468 ⟩ +≡ (473)
$hnode\_size[\texttt{TAG}(rule\_kind, b000)] = \texttt{NODE\_SIZE}(1, 0);$
$hnode\_size[\texttt{TAG}(rule\_kind, b100)] = \texttt{NODE\_SIZE}(4, 0);$
$hnode\_size[\texttt{TAG}(rule\_kind, b010)] = \texttt{NODE\_SIZE}(4, 0);$
$hnode\_size[\texttt{TAG}(rule\_kind, b001)] = \texttt{NODE\_SIZE}(4, 0);$
$hnode\_size[\texttt{TAG}(rule\_kind, b110)] = \texttt{NODE\_SIZE}(4 + 4, 0);$
$hnode\_size[\texttt{TAG}(rule\_kind, b101)] = \texttt{NODE\_SIZE}(4 + 4, 0);$
$hnode\_size[\texttt{TAG}(rule\_kind, b011)] = \texttt{NODE\_SIZE}(4 + 4, 0);$
$hnode\_size[\texttt{TAG}(rule\_kind, b111)] = \texttt{NODE\_SIZE}(4 + 4 + 4, 0);$

A.8 Glue

Glues usually contain a reference or they contain either one two or three 4 byte values depending on the info bits, and possibly even an extended dimension node followed by two 4 byte values.

⟨ initialize the *hnode_size* array 468 ⟩ +≡ (474)
$hnode\_size[\texttt{TAG}(glue\_kind, b000)] = \texttt{NODE\_SIZE}(1, 0);$
$hnode\_size[\texttt{TAG}(glue\_kind, b100)] = \texttt{NODE\_SIZE}(4, 0);$
$hnode\_size[\texttt{TAG}(glue\_kind, b010)] = \texttt{NODE\_SIZE}(4, 0);$
$hnode\_size[\texttt{TAG}(glue\_kind, b001)] = \texttt{NODE\_SIZE}(4, 0);$
$hnode\_size[\texttt{TAG}(glue\_kind, b110)] = \texttt{NODE\_SIZE}(4 + 4, 0);$
$hnode\_size[\texttt{TAG}(glue\_kind, b101)] = \texttt{NODE\_SIZE}(4 + 4, 0);$
$hnode\_size[\texttt{TAG}(glue\_kind, b011)] = \texttt{NODE\_SIZE}(4 + 4, 0);$
$hnode\_size[\texttt{TAG}(glue\_kind, b111)] = \texttt{NODE\_SIZE}(4 + 4, 1);$

A.9 Boxes

The layout of boxes is quite complex and explained in section 5.1. All boxes contain height and width, some contain a depth, some a shift amount, and some a glue setting together with glue sign and glue order. The last item in a box is a node list.

⟨ initialize the *hnode_size* array 468 ⟩ +≡ (475)
$hnode\_size[\texttt{TAG}(hbox\_kind, b000)] = \texttt{NODE\_SIZE}(4 + 4, 1);$ /* tag, height, width */
$hnode\_size[\texttt{TAG}(hbox\_kind, b001)] = \texttt{NODE\_SIZE}(4 + 4 + 4, 1);$ /* and depth */
$hnode\_size[\texttt{TAG}(hbox\_kind, b010)] = \texttt{NODE\_SIZE}(4 + 4 + 4, 1);$ /* or shift */
$hnode\_size[\texttt{TAG}(hbox\_kind, b011)] = \texttt{NODE\_SIZE}(4 + 4 + 4 + 4, 1);$ /* or both */
$hnode\_size[\texttt{TAG}(hbox\_kind, b100)] = \texttt{NODE\_SIZE}(4 + 4 + 5, 1);$ /* and glue setting */

$hnode\_size$[TAG($hbox\_kind$, $b101$)] = NODE_SIZE($4+4+4+5, 1$); /* and depth */
$hnode\_size$[TAG($hbox\_kind$, $b110$)] = NODE_SIZE($4+4+4+5, 1$); /* or shift */
$hnode\_size$[TAG($hbox\_kind$, $b111$)] = NODE_SIZE($4+4+4+4+5, 1$); /* or both */
$hnode\_size$[TAG($vbox\_kind$, $b000$)] = NODE_SIZE($4+4, 1$); /* same for vbox */
$hnode\_size$[TAG($vbox\_kind$, $b001$)] = NODE_SIZE($4+4+4, 1$);
$hnode\_size$[TAG($vbox\_kind$, $b010$)] = NODE_SIZE($4+4+4, 1$);
$hnode\_size$[TAG($vbox\_kind$, $b011$)] = NODE_SIZE($4+4+4+4, 1$);
$hnode\_size$[TAG($vbox\_kind$, $b100$)] = NODE_SIZE($4+4+5, 1$);
$hnode\_size$[TAG($vbox\_kind$, $b101$)] = NODE_SIZE($4+4+4+5, 1$);
$hnode\_size$[TAG($vbox\_kind$, $b110$)] = NODE_SIZE($4+4+4+5, 1$);
$hnode\_size$[TAG($vbox\_kind$, $b111$)] = NODE_SIZE($4+4+4+4+5, 1$);

A.10 Extended Boxes

Extended boxes start with height, width, depth, stretch, or shrink components. Then follows an extended dimension either as a reference or a node. The node ends with a list.

⟨ initialize the $hnode\_size$ array 468 ⟩ +≡ (476)
$hnode\_size$[TAG($hset\_kind$, $b000$)] = NODE_SIZE($4+4+4+4+1, 1$);
$hnode\_size$[TAG($hset\_kind$, $b001$)] = NODE_SIZE($4+4+4+4+4+1, 1$);
$hnode\_size$[TAG($hset\_kind$, $b010$)] = NODE_SIZE($4+4+4+4+4+1, 1$);
$hnode\_size$[TAG($hset\_kind$, $b011$)] = NODE_SIZE($4+4+4+4+4+4+1, 1$);
$hnode\_size$[TAG($vset\_kind$, $b000$)] = NODE_SIZE($4+4+4+4+1, 1$);
$hnode\_size$[TAG($vset\_kind$, $b001$)] = NODE_SIZE($4+4+4+4+4+1, 1$);
$hnode\_size$[TAG($vset\_kind$, $b010$)] = NODE_SIZE($4+4+4+4+4+1, 1$);
$hnode\_size$[TAG($vset\_kind$, $b011$)] = NODE_SIZE($4+4+4+4+4+4+1, 1$);
$hnode\_size$[TAG($hset\_kind$, $b100$)] = NODE_SIZE($4+4+4+4, 2$);
$hnode\_size$[TAG($hset\_kind$, $b101$)] = NODE_SIZE($4+4+4+4+4, 2$);
$hnode\_size$[TAG($hset\_kind$, $b110$)] = NODE_SIZE($4+4+4+4+4, 2$);
$hnode\_size$[TAG($hset\_kind$, $b111$)] = NODE_SIZE($4+4+4+4+4+4, 2$);
$hnode\_size$[TAG($vset\_kind$, $b100$)] = NODE_SIZE($4+4+4+4, 2$);
$hnode\_size$[TAG($vset\_kind$, $b101$)] = NODE_SIZE($4+4+4+4+4, 2$);
$hnode\_size$[TAG($vset\_kind$, $b110$)] = NODE_SIZE($4+4+4+4+4, 2$);
$hnode\_size$[TAG($vset\_kind$, $b111$)] = NODE_SIZE($4+4+4+4+4+4, 2$);

The hpack and vpack nodes start with a shift amount and in case of vpack a depth. Then again an extended dimension and a list.

⟨ initialize the $hnode\_size$ array 468 ⟩ +≡ (477)
$hnode\_size$[TAG($hpack\_kind$, $b000$)] = NODE_SIZE($1, 1$);
$hnode\_size$[TAG($hpack\_kind$, $b001$)] = NODE_SIZE($1, 1$);
$hnode\_size$[TAG($hpack\_kind$, $b010$)] = NODE_SIZE($4+1, 1$);
$hnode\_size$[TAG($hpack\_kind$, $b011$)] = NODE_SIZE($4+1, 1$);
$hnode\_size$[TAG($vpack\_kind$, $b000$)] = NODE_SIZE($4+1, 1$);
$hnode\_size$[TAG($vpack\_kind$, $b001$)] = NODE_SIZE($4+1, 1$);

$hnode\_size$[TAG($vpack\_kind$, $b010$)] = NODE_SIZE($4 + 4 + 1, 1$);
$hnode\_size$[TAG($vpack\_kind$, $b011$)] = NODE_SIZE($4 + 4 + 1, 1$);
$hnode\_size$[TAG($hpack\_kind$, $b100$)] = NODE_SIZE($0, 2$);
$hnode\_size$[TAG($hpack\_kind$, $b101$)] = NODE_SIZE($0, 2$);
$hnode\_size$[TAG($hpack\_kind$, $b110$)] = NODE_SIZE($4, 2$);
$hnode\_size$[TAG($hpack\_kind$, $b111$)] = NODE_SIZE($4, 2$);
$hnode\_size$[TAG($vpack\_kind$, $b100$)] = NODE_SIZE($4, 2$);
$hnode\_size$[TAG($vpack\_kind$, $b101$)] = NODE_SIZE($4, 2$);
$hnode\_size$[TAG($vpack\_kind$, $b110$)] = NODE_SIZE($4 + 4, 2$);
$hnode\_size$[TAG($vpack\_kind$, $b111$)] = NODE_SIZE($4 + 4, 2$);

A.11 Leaders

Most leader nodes will use a reference. Otherwise they contain a glue node followed by a box or rule node.

⟨ initialize the *hnode_size* array 468 ⟩ +≡ (478)
$hnode\_size$[TAG($leaders\_kind$, $b000$)] = NODE_SIZE($1, 0$);
$hnode\_size$[TAG($leaders\_kind$, 1)] = NODE_SIZE($0, 1$);
$hnode\_size$[TAG($leaders\_kind$, 2)] = NODE_SIZE($0, 1$);
$hnode\_size$[TAG($leaders\_kind$, 3)] = NODE_SIZE($0, 1$);
$hnode\_size$[TAG($leaders\_kind$, $b100 \mid 1$)] = NODE_SIZE($0, 2$);
$hnode\_size$[TAG($leaders\_kind$, $b100 \mid 2$)] = NODE_SIZE($0, 2$);
$hnode\_size$[TAG($leaders\_kind$, $b100 \mid 3$)] = NODE_SIZE($0, 2$);

A.12 Baseline Skips

Here we expect either a reference or two optional glue nodes followed by an optional dimension.

⟨ initialize the *hnode_size* array 468 ⟩ +≡ (479)
$hnode\_size$[TAG($baseline\_kind$, $b000$)] = NODE_SIZE($1, 0$);
$hnode\_size$[TAG($baseline\_kind$, $b001$)] = NODE_SIZE($4, 0$);
$hnode\_size$[TAG($baseline\_kind$, $b010$)] = NODE_SIZE($0, 1$);
$hnode\_size$[TAG($baseline\_kind$, $b100$)] = NODE_SIZE($0, 1$);
$hnode\_size$[TAG($baseline\_kind$, $b110$)] = NODE_SIZE($0, 2$);
$hnode\_size$[TAG($baseline\_kind$, $b011$)] = NODE_SIZE($4, 1$);
$hnode\_size$[TAG($baseline\_kind$, $b101$)] = NODE_SIZE($4, 1$);
$hnode\_size$[TAG($baseline\_kind$, $b111$)] = NODE_SIZE($4, 2$);

A.13 Ligatures

As usual a reference is possible, otherwise the font is followed by character bytes as given by the info. Only if the info value is 7, the number of character bytes is stored separately.

⟨ initialize the *hnode_size* array 468 ⟩ +≡ (480)
$hnode\_size$[TAG($ligature\_kind$, $b000$)] = NODE_SIZE($1, 0$);
$hnode\_size$[TAG($ligature\_kind$, 1)] = NODE_SIZE($1 + 1, 0$);

$hnode\_size$[TAG($ligature\_kind$, 2)] = NODE_SIZE(1 + 2, 0);
$hnode\_size$[TAG($ligature\_kind$, 3)] = NODE_SIZE(1 + 3, 0);
$hnode\_size$[TAG($ligature\_kind$, 4)] = NODE_SIZE(1 + 4, 0);
$hnode\_size$[TAG($ligature\_kind$, 5)] = NODE_SIZE(1 + 5, 0);
$hnode\_size$[TAG($ligature\_kind$, 6)] = NODE_SIZE(1 + 6, 0);
$hnode\_size$[TAG($ligature\_kind$, 7)] = NODE_SIZE(1, 1);

A.14 Discretionary breaks

The simple cases here are references, discretionary breaks with empty pre- and post-list, or with a zero line skip limit Otherwise one or two lists are followed by an optional replace count.

⟨ initialize the *hnode_size* array 468 ⟩ +≡ (481)
$hnode\_size$[TAG($disc\_kind$, $b000$)] = NODE_SIZE(1, 0);
$hnode\_size$[TAG($disc\_kind$, $b010$)] = NODE_SIZE(0, 1);
$hnode\_size$[TAG($disc\_kind$, $b011$)] = NODE_SIZE(0, 2);
$hnode\_size$[TAG($disc\_kind$, $b100$)] = NODE_SIZE(1, 0);
$hnode\_size$[TAG($disc\_kind$, $b110$)] = NODE_SIZE(1, 1);
$hnode\_size$[TAG($disc\_kind$, $b111$)] = NODE_SIZE(1, 2);

A.15 Paragraphs

Paragraph nodes contain an extended dimension, an parameter list and a list. The first two can be given as a reference.

⟨ initialize the *hnode_size* array 468 ⟩ +≡ (482)
$hnode\_size$[TAG(par_kind, $b000$)] = NODE_SIZE(1 + 1, 1);
$hnode\_size$[TAG(par_kind, $b010$)] = NODE_SIZE(1, 2);
$hnode\_size$[TAG(par_kind, $b110$)] = NODE_SIZE(0, 3);
$hnode\_size$[TAG(par_kind, $b100$)] = NODE_SIZE(1, 2);

A.16 Mathematics

Displayed math needs a parameter list, either as list or as reference followed by an optional left or right equation number and a list. Text math is simpler: the only information is in the info value.

⟨ initialize the *hnode_size* array 468 ⟩ +≡ (483)
$hnode\_size$[TAG($math\_kind$, $b000$)] = NODE_SIZE(1, 1);
$hnode\_size$[TAG($math\_kind$, $b001$)] = NODE_SIZE(1, 2);
$hnode\_size$[TAG($math\_kind$, $b010$)] = NODE_SIZE(1, 2);
$hnode\_size$[TAG($math\_kind$, $b100$)] = NODE_SIZE(0, 2);
$hnode\_size$[TAG($math\_kind$, $b101$)] = NODE_SIZE(0, 3);
$hnode\_size$[TAG($math\_kind$, $b110$)] = NODE_SIZE(0, 3);
$hnode\_size$[TAG($math\_kind$, $b111$)] = NODE_SIZE(0, 0);
$hnode\_size$[TAG($math\_kind$, $b011$)] = NODE_SIZE(0, 0);

A.17 Adjustments

⟨ initialize the *hnode_size* array 468 ⟩ +≡ (484)

$hnode\_size[\texttt{TAG}(adjust\_kind, 1)] = \texttt{NODE\_SIZE}(0, 1);$

A.18 Tables

Tables have an extended dimension either as a node or as a reference followed by two lists.

⟨ initialize the *hnode_size* array 468 ⟩ +≡ (485)

$hnode\_size[\texttt{TAG}(table\_kind, b000)] = \texttt{NODE\_SIZE}(1, 2);$
$hnode\_size[\texttt{TAG}(table\_kind, b001)] = \texttt{NODE\_SIZE}(1, 2);$
$hnode\_size[\texttt{TAG}(table\_kind, b010)] = \texttt{NODE\_SIZE}(1, 2);$
$hnode\_size[\texttt{TAG}(table\_kind, b011)] = \texttt{NODE\_SIZE}(1, 2);$
$hnode\_size[\texttt{TAG}(table\_kind, b100)] = \texttt{NODE\_SIZE}(0, 3);$
$hnode\_size[\texttt{TAG}(table\_kind, b101)] = \texttt{NODE\_SIZE}(0, 3);$
$hnode\_size[\texttt{TAG}(table\_kind, b110)] = \texttt{NODE\_SIZE}(0, 3);$
$hnode\_size[\texttt{TAG}(table\_kind, b111)] = \texttt{NODE\_SIZE}(0, 3);$

Outer item nodes are lists of inner item nodes, inner item nodes are box nodes followed by an optional span count.

⟨ initialize the *hnode_size* array 468 ⟩ +≡ (486)

$hnode\_size[\texttt{TAG}(item\_kind, b000)] = \texttt{NODE\_SIZE}(0, 1);$ /* outer */
$hnode\_size[\texttt{TAG}(item\_kind, 1)] = \texttt{NODE\_SIZE}(0, 1);$ /* inner */
$hnode\_size[\texttt{TAG}(item\_kind, 2)] = \texttt{NODE\_SIZE}(0, 1);$
$hnode\_size[\texttt{TAG}(item\_kind, 3)] = \texttt{NODE\_SIZE}(0, 1);$
$hnode\_size[\texttt{TAG}(item\_kind, 4)] = \texttt{NODE\_SIZE}(0, 1);$
$hnode\_size[\texttt{TAG}(item\_kind, 5)] = \texttt{NODE\_SIZE}(0, 1);$
$hnode\_size[\texttt{TAG}(item\_kind, 6)] = \texttt{NODE\_SIZE}(0, 1);$
$hnode\_size[\texttt{TAG}(item\_kind, 7)] = \texttt{NODE\_SIZE}(1, 1);$

A.19 Images

If not given by a reference, images contain a section reference and optional dimensions and a descriptive list.

⟨ initialize the *hnode_size* array 468 ⟩ +≡ (487)

$hnode\_size[\texttt{TAG}(image\_kind, b000)] = \texttt{NODE\_SIZE}(1, 0);$
$hnode\_size[\texttt{TAG}(image\_kind, b001)] = \texttt{NODE\_SIZE}(2 + 4 + 4, 1);$
$hnode\_size[\texttt{TAG}(image\_kind, b010)] = \texttt{NODE\_SIZE}(2 + 4 + 4, 1);$
$hnode\_size[\texttt{TAG}(image\_kind, b011)] = \texttt{NODE\_SIZE}(2 + 4 + 4, 1);$
$hnode\_size[\texttt{TAG}(image\_kind, b100)] = \texttt{NODE\_SIZE}(2 + 4 + 1 + 1, 1);$
$hnode\_size[\texttt{TAG}(image\_kind, b101)] = \texttt{NODE\_SIZE}(2 + 4 + 1, 2);$
$hnode\_size[\texttt{TAG}(image\_kind, b110)] = \texttt{NODE\_SIZE}(2 + 4 + 1, 2);$
$hnode\_size[\texttt{TAG}(image\_kind, b111)] = \texttt{NODE\_SIZE}(2 + 4, 3);$

A.20 Links

Links contain either a 2 byte or a 1 byte reference and possibly a color reference.

⟨ initialize the *hnode_size* array 468 ⟩ +≡ (488)

hnode_size[`TAG`(*link_kind*, *b000*)] = `NODE_SIZE`(1, 0);
hnode_size[`TAG`(*link_kind*, *b001*)] = `NODE_SIZE`(2, 0);
hnode_size[`TAG`(*link_kind*, *b010*)] = `NODE_SIZE`(1, 0);
hnode_size[`TAG`(*link_kind*, *b011*)] = `NODE_SIZE`(2, 0);
hnode_size[`TAG`(*link_kind*, *b100*)] = `NODE_SIZE`(2, 0);
hnode_size[`TAG`(*link_kind*, *b101*)] = `NODE_SIZE`(3, 0);
hnode_size[`TAG`(*link_kind*, *b110*)] = `NODE_SIZE`(2, 0);
hnode_size[`TAG`(*link_kind*, *b111*)] = `NODE_SIZE`(3, 0);

A.21 Streams

After the stream reference follows a parameter list, either as reference or as a list, and then a content list.

⟨ initialize the *hnode_size* array 468 ⟩ +≡ (489)

hnode_size[`TAG`(*stream_kind*, *b000*)] = `NODE_SIZE`(1 + 1, 1);
hnode_size[`TAG`(*stream_kind*, *b010*)] = `NODE_SIZE`(1, 2);
hnode_size[`TAG`(*stream_kind*, *b100*)] = `NODE_SIZE`(1, 0);

A.22 Colors

⟨ initialize the *hnode_size* array 468 ⟩ +≡ (490)

hnode_size[`TAG`(*color_kind*, *b000*)] = `NODE_SIZE`(1, 0);

B Reading Short Format Files Backwards

This section is not really part of the file format definition, but it illustrates an important property of the content section in short format files: it can be read in both directions. This is important because we want to be able to start at an arbitrary point in the content and from there move pagewise backward.

The program `skip` described in this section does just that. As wee see in appendix C.12, its *main* program is almost the same as the *main* program of the program `stretch` in appendix C.11. The major difference is the removal of an output file and the replacement of the call to *hwrite_content_section* by a call to *hteg_content_section*.

⟨ skip functions 491 ⟩ ≡ (491)

```
static void hteg_content_section(void)
{ hget_section(2);
  hpos = hend;
  while (hpos > hstart) hteg_content_node();
}
```

Used in 557.

The functions *hteg_content_section* and *hteg_content_node* above are reverse versions of the functions *hget_content_section* and *hget_content_node*. Many such "reverse functions" will follow now and we will consistently use the same naming scheme: replacing "*get*" by "*teg*" or "GET" by "TEG". The `skip` program does not do much input checking; it will just extract enough information from a content node to skip a node and "advance" or better "retreat" to the previous node.

⟨ skip functions 491 ⟩ +≡ (492)

```
static void hteg_content_node(void)
{ ⟨ skip the end byte z 493 ⟩
  hteg_content(z);
  ⟨ skip and check the start byte a 494 ⟩
}
static void hteg_content(Tag z)
{ switch (z)
  { ⟨ cases to skip content 501 ⟩
    default:
      if (¬hteg_unknown(z)) TAGERR(z);
      break;
  }
```

```
  }
```

The code to skip the end byte z and to check the start byte a is used repeatedly.

```
⟨ skip the end byte z 493 ⟩ ≡                                        (493)
  Tag a, z;                               /* the start and the end byte */
  uint32_t node_pos = hpos − hstart;
  if (hpos ≤ hstart) return;
  HTEGTAG(z);
```

Used in 492, 498, 509, 512, 515, and 535.

```
⟨ skip and check the start byte a 494 ⟩ ≡                            (494)
  HTEGTAG(a);
  if (a ≠ z)
    QUIT("Tag␣mismatch␣[%s,%d]!=[%s,%d]␣at␣"SIZE_F"␣to␣0x%x\n",
        NAME(a), INFO(a), NAME(z), INFO(z),
        hpos − hstart, node_pos − 1);
```

Used in 492, 498, 509, 512, 515, and 535.

We replace the "GET" macros by the following "TEG" macros:

```
⟨ shared get macros 38 ⟩ +≡                                          (495)
#define HBACK(X)
  ((hpos − (X) < hstart) ? (QUIT("HTEG␣underflow\n"), NULL) : (hpos −= (X)))
#define HTEG8 (HBACK(1), hpos[0])
#define HTEG16(X) (HBACK(2), (X) = (hpos[0] ≪ 8) + hpos[1])
#define HTEG24(X) (HBACK(3), (X) = (hpos[0] ≪ 16) + (hpos[1] ≪ 8) + hpos[2])
#define HTEG32(X)
  (HBACK(4), (X) = (hpos[0] ≪ 24) + (hpos[1] ≪ 16) + (hpos[2] ≪ 8) + hpos[3])
#define HTEGTAG(X) X = HTEG8, DBGTAG(X, hpos)
```

Now we review step by step the different kinds of nodes.

B.1 Floating Point Numbers

```
⟨ shared skip functions 464 ⟩ +≡                                     (496)
  float32_t hteg_float32(void)
  { union { float32_t d; uint32_t bits; } u;
    HTEG32(u.bits);
    return u.d;
  }
```

B.2 Extended Dimensions

⟨ skip macros 497 ⟩ ≡ (497)

```
#define HTEG_XDIMEN(I, X)
  if ((I) & b001) HTEG32((X).v);
  if ((I) & b010) HTEG32((X).h);
  if ((I) & b100) HTEG32((X).w);
```

Used in 546 and 557.

⟨ skip functions 491 ⟩ +≡ (498)

```
  static void hteg_xdimen_node(Xdimen *x)
  { ⟨ skip the end byte z 493 ⟩
    switch (z) {
#if 0                        /* currently the info value 0 is not supported */
    case TAG(xdimen_kind, b000):                       /* see section 10.5 */
      { uint8_t n; n = HTEG8;
      } break;
#endif
    case TAG(xdimen_kind, b001): HTEG_XDIMEN(b001, *x); break;
    case TAG(xdimen_kind, b010): HTEG_XDIMEN(b010, *x); break;
    case TAG(xdimen_kind, b011): HTEG_XDIMEN(b011, *x); break;
    case TAG(xdimen_kind, b100): HTEG_XDIMEN(b100, *x); break;
    case TAG(xdimen_kind, b101): HTEG_XDIMEN(b101, *x); break;
    case TAG(xdimen_kind, b110): HTEG_XDIMEN(b110, *x); break;
    case TAG(xdimen_kind, b111): HTEG_XDIMEN(b111, *x); break;
    default: QUIT("Extent␣expected␣at␣0x%x␣got␣%s", node_pos, NAME(z));
      break;
    }
    ⟨ skip and check the start byte a 494 ⟩
  }
```

B.3 Stretch and Shrink

⟨ skip macros 497 ⟩ +≡ (499)

```
#define HTEG_STRETCH(S)
  { Stch st; HTEG32(st.u); S.o = st.u & 3; st.u &= ~3; S.f = st.f; }
```

B.4 Glyphs

⟨ skip macros 497 ⟩ +≡ (500)

```
#define HTEG_GLYPH (I, G) (G).f = HTEG8;
  if (I ≡ 1) (G).c = HTEG8;
  else if (I ≡ 2) HTEG16((G).c);
  else if (I ≡ 3) HTEG24((G).c);
  else if (I ≡ 4) HTEG32((G).c);
```

⟨ cases to skip content 501 ⟩ ≡ (501)

case TAG(*glyph_kind*, 1): { **Glyph** g; HTEG_GLYPH(1, g); } **break**;
case TAG(*glyph_kind*, 2): { **Glyph** g; HTEG_GLYPH(2, g); } **break**;
case TAG(*glyph_kind*, 3): { **Glyph** g; HTEG_GLYPH(3, g); } **break**;
case TAG(*glyph_kind*, 4): { **Glyph** g; HTEG_GLYPH(4, g); } **break**;

Used in 492.

B.5 Penalties

⟨ skip macros 497 ⟩ +≡ (502)

#**define** HTEG_PENALTY(I, P)
if $(I \equiv 1)$ { **int8_t** n; $n =$ HTEG8; $P = n$; }
else if $(I \equiv 2)$ { **int16_t** n; HTEG16(n); $P = n$; }
else if $(I \equiv 3)$ { **int32_t** n; HTEG32(n); $P = n$; }

⟨ cases to skip content 501 ⟩ +≡ (503)

case TAG(*penalty_kind*, 1): { **int32_t** p; HTEG_PENALTY(1, p); } **break**;
case TAG(*penalty_kind*, 2): { **int32_t** p; HTEG_PENALTY(2, p); } **break**;
case TAG(*penalty_kind*, 3): { **int32_t** p; HTEG_PENALTY(2, p); } **break**;

B.6 Kerns

⟨ skip macros 497 ⟩ +≡ (504)

#**define** HTEG_KERN(I, X)
if $(((I)$ & *b011*$) \equiv 2)$ HTEG32($X.w$);
else if $(((I)$ & *b011*$) \equiv 3)$*hteg_xdimen_node*(&(X))

⟨ cases to skip content 501 ⟩ +≡ (505)

case TAG(*kern_kind*, *b010*): { **Xdimen** x; HTEG_KERN(*b010*, x); } **break**;
case TAG(*kern_kind*, *b011*): { **Xdimen** x; HTEG_KERN(*b011*, x); } **break**;
case TAG(*kern_kind*, *b110*): { **Xdimen** x; HTEG_KERN(*b110*, x); } **break**;
case TAG(*kern_kind*, *b111*): { **Xdimen** x; HTEG_KERN(*b111*, x); } **break**;

B.7 Language

⟨ cases to skip content 501 ⟩ +≡ (506)

case TAG(*language_kind*, 1): **case** TAG(*language_kind*, 2):
case TAG(*language_kind*, 3): **case** TAG(*language_kind*, 4):
case TAG(*language_kind*, 5): **case** TAG(*language_kind*, 6):
case TAG(*language_kind*, 7): **break**;

B.8 Rules

⟨ skip macros 497 ⟩ +≡ (507)

```
#define HTEG_RULE(I, R)
  if ((I) & b001) HTEG32((R).w); else (R).w = RUNNING_DIMEN;
  if ((I) & b010) HTEG32((R).d); else (R).d = RUNNING_DIMEN;
  if ((I) & b100) HTEG32((R).h); else (R).h = RUNNING_DIMEN;
```

⟨ cases to skip content 501 ⟩ +≡ (508)

```
  case TAG(rule_kind, b011): { Rule r; HTEG_RULE(b011, r); } break;
  case TAG(rule_kind, b101): { Rule r; HTEG_RULE(b101, r); } break;
  case TAG(rule_kind, b001): { Rule r; HTEG_RULE(b001, r); } break;
  case TAG(rule_kind, b110): { Rule r; HTEG_RULE(b110, r); } break;
  case TAG(rule_kind, b111): { Rule r; HTEG_RULE(b111, r); } break;
```

⟨ skip functions 491 ⟩ +≡ (509)

```
  static void hteg_rule_node(void)
  { ⟨ skip the end byte z 493 ⟩
    if (KIND(z) ≡ rule_kind) { Rule r; HTEG_RULE(INFO(z), r); }
    else QUIT("Rule␣expected␣at␣0x%x␣got␣%s", node_pos, NAME(z));
    ⟨ skip and check the start byte a 494 ⟩
  }
```

B.9 Glue

⟨ skip macros 497 ⟩ +≡ (510)

```
#define HTEG_GLUE(I, G)
  if (I ≡ b111) hteg_xdimen_node(&((G).w));
  else (G).w.h = (G).w.v = 0.0;
  if ((I) & b001) HTEG_STRETCH((G).m) else (G).m.f = 0.0, (G).m.o = 0;
  if ((I) & b010) HTEG_STRETCH((G).p) else (G).p.f = 0.0, (G).p.o = 0;
  if ((I) ≠ b111) { if ((I) & b100) HTEG32((G).w.w); else (G).w.w = 0; }
```

⟨ cases to skip content 501 ⟩ +≡ (511)

```
  case TAG(glue_kind, b001): { Glue g; HTEG_GLUE(b001, g); } break;
  case TAG(glue_kind, b010): { Glue g; HTEG_GLUE(b010, g); } break;
  case TAG(glue_kind, b011): { Glue g; HTEG_GLUE(b011, g); } break;
  case TAG(glue_kind, b100): { Glue g; HTEG_GLUE(b100, g); } break;
  case TAG(glue_kind, b101): { Glue g; HTEG_GLUE(b101, g); } break;
  case TAG(glue_kind, b110): { Glue g; HTEG_GLUE(b110, g); } break;
  case TAG(glue_kind, b111): { Glue g; HTEG_GLUE(b111, g); } break;
```

⟨ skip functions 491 ⟩ +≡ (512)

```
  static void hteg_glue_node(void)
  { ⟨ skip the end byte z 493 ⟩
    if (INFO(z) ≡ b000) HTEG_REF(glue_kind);
    else { Glue g; HTEG_GLUE(INFO(z), g); }
```

```
    ⟨ skip and check the start byte a 494 ⟩
  }
```

B.10 Boxes

```
⟨ skip macros 497 ⟩ +≡                                                    (513)
#define HTEG_BOX(I, B) hteg_list (&(B.l));
  if ((I) & b100)
  { B.s = HTEG8; B.r = hteg_float32(); B.o = B.s & #F; B.s = B.s ≫ 4; }
  else { B.r = 0.0; B.o = B.s = 0; }
  if ((I) & b010) HTEG32(B.a); else B.a = 0;
  HTEG32(B.w);
  if ((I) & b001) HTEG32(B.d); else B.d = 0;
  HTEG32(B.h);
```

```
⟨ cases to skip content 501 ⟩ +≡                                          (514)
  case TAG(hbox_kind, b000): { Box b; HTEG_BOX(b000, b); } break;
  case TAG(hbox_kind, b001): { Box b; HTEG_BOX(b001, b); } break;
  case TAG(hbox_kind, b010): { Box b; HTEG_BOX(b010, b); } break;
  case TAG(hbox_kind, b011): { Box b; HTEG_BOX(b011, b); } break;
  case TAG(hbox_kind, b100): { Box b; HTEG_BOX(b100, b); } break;
  case TAG(hbox_kind, b101): { Box b; HTEG_BOX(b101, b); } break;
  case TAG(hbox_kind, b110): { Box b; HTEG_BOX(b110, b); } break;
  case TAG(hbox_kind, b111): { Box b; HTEG_BOX(b111, b); } break;
  case TAG(vbox_kind, b000): { Box b; HTEG_BOX(b000, b); } break;
  case TAG(vbox_kind, b001): { Box b; HTEG_BOX(b001, b); } break;
  case TAG(vbox_kind, b010): { Box b; HTEG_BOX(b010, b); } break;
  case TAG(vbox_kind, b011): { Box b; HTEG_BOX(b011, b); } break;
  case TAG(vbox_kind, b100): { Box b; HTEG_BOX(b100, b); } break;
  case TAG(vbox_kind, b101): { Box b; HTEG_BOX(b101, b); } break;
  case TAG(vbox_kind, b110): { Box b; HTEG_BOX(b110, b); } break;
  case TAG(vbox_kind, b111): { Box b; HTEG_BOX(b111, b); } break;
```

```
⟨ skip functions 491 ⟩ +≡                                                 (515)
  static void hteg_hbox_node(void)
  { Box b;
    ⟨ skip the end byte z 493 ⟩
    if (KIND(z) ≠ hbox_kind)
      QUIT("Hbox␣expected␣at␣0x%x␣got␣%s", node_pos, NAME(z));
    HTEG_BOX(INFO(z), b);
    ⟨ skip and check the start byte a 494 ⟩
  }
  static void hteg_vbox_node(void)
  { Box b;
```

```
    ⟨ skip the end byte z 493 ⟩
    if (KIND(z) ≠ vbox_kind)
      QUIT("Vbox␣expected␣at␣0x%x␣got␣%s", node_pos, NAME(z));
    HTEG_BOX(INFO(z), b);
    ⟨ skip and check the start byte a 494 ⟩
  }
```

B.11 Extended Boxes

⟨ skip macros 497 ⟩ +≡ (516)

```
#define HTEG_SET(I)
  { List l; hteg_list(&l); }
  if ((I) & b100) { Xdimen x; hteg_xdimen_node(&x); }
  else HTEG_REF(xdimen_kind);
  { Stretch m; HTEG_STRETCH(m); }
  { Stretch p; HTEG_STRETCH(p); }
  if ((I) & b010) { Dimen a; HTEG32(a); }
  { Dimen w; HTEG32(w); }
  { Dimen d; if ((I) & b001) HTEG32(d); else d = 0; }
  { Dimen h; HTEG32(h); }
#define HTEG_PACK(K, I)
  { List l; hteg_list(&l); }
  if ((I) & b100) { Xdimen x;
    hteg_xdimen_node(&x); } else HTEG_REF(xdimen_kind);
  if ((I) & b010) { Dimen d; HTEG32(d); }
  if (K ≡ vpack_kind) { Dimen d; HTEG32(d); }
```

⟨ cases to skip content 501 ⟩ +≡ (517)

```
  case TAG(hset_kind, b000): HTEG_SET(b000); break;
  case TAG(hset_kind, b001): HTEG_SET(b001); break;
  case TAG(hset_kind, b010): HTEG_SET(b010); break;
  case TAG(hset_kind, b011): HTEG_SET(b011); break;
  case TAG(hset_kind, b100): HTEG_SET(b100); break;
  case TAG(hset_kind, b101): HTEG_SET(b101); break;
  case TAG(hset_kind, b110): HTEG_SET(b110); break;
  case TAG(hset_kind, b111): HTEG_SET(b111); break;
  case TAG(vset_kind, b000): HTEG_SET(b000); break;
  case TAG(vset_kind, b001): HTEG_SET(b001); break;
  case TAG(vset_kind, b010): HTEG_SET(b010); break;
  case TAG(vset_kind, b011): HTEG_SET(b011); break;
  case TAG(vset_kind, b100): HTEG_SET(b100); break;
  case TAG(vset_kind, b101): HTEG_SET(b101); break;
  case TAG(vset_kind, b110): HTEG_SET(b110); break;
  case TAG(vset_kind, b111): HTEG_SET(b111); break;
```

```
case TAG(hpack_kind, b000): HTEG_PACK(hpack_kind, b000); break;
case TAG(hpack_kind, b001): HTEG_PACK(hpack_kind, b001); break;
case TAG(hpack_kind, b010): HTEG_PACK(hpack_kind, b010); break;
case TAG(hpack_kind, b011): HTEG_PACK(hpack_kind, b011); break;
case TAG(hpack_kind, b100): HTEG_PACK(hpack_kind, b100); break;
case TAG(hpack_kind, b101): HTEG_PACK(hpack_kind, b101); break;
case TAG(hpack_kind, b110): HTEG_PACK(hpack_kind, b110); break;
case TAG(hpack_kind, b111): HTEG_PACK(hpack_kind, b111); break;
case TAG(vpack_kind, b000): HTEG_PACK(vpack_kind, b000); break;
case TAG(vpack_kind, b001): HTEG_PACK(vpack_kind, b001); break;
case TAG(vpack_kind, b010): HTEG_PACK(vpack_kind, b010); break;
case TAG(vpack_kind, b011): HTEG_PACK(vpack_kind, b011); break;
case TAG(vpack_kind, b100): HTEG_PACK(vpack_kind, b100); break;
case TAG(vpack_kind, b101): HTEG_PACK(vpack_kind, b101); break;
case TAG(vpack_kind, b110): HTEG_PACK(vpack_kind, b110); break;
case TAG(vpack_kind, b111): HTEG_PACK(vpack_kind, b111); break;
```

B.12 Leaders

⟨ skip macros 497 ⟩ +≡ (518)

```
#define HTEG_LEADERS(I)
  if (KIND(hpos[−1]) ≡ rule_kind) hteg_rule_node();
  else if (KIND(hpos[−1]) ≡ hbox_kind) hteg_hbox_node();
  else hteg_vbox_node();
  if ((I) & b100) hteg_glue_node();
```

⟨ cases to skip content 501 ⟩ +≡ (519)

```
case TAG(leaders_kind, 1): HTEG_LEADERS(1); break;
case TAG(leaders_kind, 2): HTEG_LEADERS(2); break;
case TAG(leaders_kind, 3): HTEG_LEADERS(3); break;
case TAG(leaders_kind, b100 | 1): HTEG_LEADERS(b100 | 1); break;
case TAG(leaders_kind, b100 | 2): HTEG_LEADERS(b100 | 2); break;
case TAG(leaders_kind, b100 | 3): HTEG_LEADERS(b100 | 3); break;
```

B.13 Baseline Skips

⟨ skip macros 497 ⟩ +≡ (520)

```
#define HTEG_BASELINE(I, B)
  if ((I) & b010) hteg_glue_node();
  else { B.ls.p.o = B.ls.m.o = B.ls.w.w = 0;
    B.ls.w.h = B.ls.w.v = B.ls.p.f = B.ls.m.f = 0.0; }
  if ((I) & b100) hteg_glue_node();
  else { B.bs.p.o = B.bs.m.o = B.bs.w.w = 0;
    B.bs.w.h = B.bs.w.v = B.bs.p.f = B.bs.m.f = 0.0; }
  if ((I) & b001) HTEG32((B).lsl); else B.lsl = 0;
```

⟨ cases to skip content 501 ⟩ +≡ (521)

case TAG(*baseline_kind*, *b001*): { **Baseline** *b*; HTEG_BASELINE(*b001*, *b*); } **break**;
case TAG(*baseline_kind*, *b010*): { **Baseline** *b*; HTEG_BASELINE(*b010*, *b*); } **break**;
case TAG(*baseline_kind*, *b011*): { **Baseline** *b*; HTEG_BASELINE(*b011*, *b*); } **break**;
case TAG(*baseline_kind*, *b100*): { **Baseline** *b*; HTEG_BASELINE(*b100*, *b*); } **break**;
case TAG(*baseline_kind*, *b101*): { **Baseline** *b*; HTEG_BASELINE(*b101*, *b*); } **break**;
case TAG(*baseline_kind*, *b110*): { **Baseline** *b*; HTEG_BASELINE(*b110*, *b*); } **break**;
case TAG(*baseline_kind*, *b111*): { **Baseline** *b*; HTEG_BASELINE(*b111*, *b*); } **break**;

B.14 Ligatures

⟨ skip macros 497 ⟩ +≡ (522)

#define HTEG_LIG(I, L)
if $((I) \equiv 7)$ *hteg_list*(&((*L*).*l*));
else { $(L).l.s = (I)$; $hpos$ −= $(L).l.s$; $(L).l.p = hpos - hstart$; }
$(L).f =$ HTEG8;

⟨ cases to skip content 501 ⟩ +≡ (523)

case TAG(*ligature_kind*, 1): { **Lig** *l*; HTEG_LIG(1, *l*); } **break**;
case TAG(*ligature_kind*, 2): { **Lig** *l*; HTEG_LIG(2, *l*); } **break**;
case TAG(*ligature_kind*, 3): { **Lig** *l*; HTEG_LIG(3, *l*); } **break**;
case TAG(*ligature_kind*, 4): { **Lig** *l*; HTEG_LIG(4, *l*); } **break**;
case TAG(*ligature_kind*, 5): { **Lig** *l*; HTEG_LIG(5, *l*); } **break**;
case TAG(*ligature_kind*, 6): { **Lig** *l*; HTEG_LIG(6, *l*); } **break**;
case TAG(*ligature_kind*, 7): { **Lig** *l*; HTEG_LIG(7, *l*); } **break**;

B.15 Discretionary breaks

⟨ skip macros 497 ⟩ +≡ (524)

#define HTEG_DISC(I, H)
if ((*I*) & *b001*) *hteg_list*(&((H).*q*));
else { (H).$q.p = hpos - hstart$; (H).$q.s = 0$; (H).$q.t =$ TAG(*list_kind*, *b000*); }
if ((*I*) & *b010*) *hteg_list*(&((H).*p*));
else { (H).$p.p = hpos - hstart$; (H).$p.s = 0$; (H).$p.t =$ TAG(*list_kind*, *b000*); }
if ((*I*) & *b100*) (H).$r =$ HTEG8; **else** (H).$r = 0$;

⟨ cases to skip content 501 ⟩ +≡ (525)

```
case TAG(disc_kind, b001): { Disc h; HTEG_DISC(b001, h); } break;
case TAG(disc_kind, b010): { Disc h; HTEG_DISC(b010, h); } break;
case TAG(disc_kind, b011): { Disc h; HTEG_DISC(b011, h); } break;
case TAG(disc_kind, b100): { Disc h; HTEG_DISC(b100, h); } break;
case TAG(disc_kind, b101): { Disc h; HTEG_DISC(b101, h); } break;
case TAG(disc_kind, b110): { Disc h; HTEG_DISC(b110, h); } break;
case TAG(disc_kind, b111): { Disc h; HTEG_DISC(b111, h); } break;
```

B.16 Paragraphs

⟨ skip macros 497 ⟩ +≡ (526)

```
#define HTEG_PAR(I)
  { List l; hteg_list(&l); }
  if ((I) & b010) { List l; hteg_param_list(&l); }
  else if ((I) ≠ b100) HTEG_REF(param_kind);
  if ((I) & b100) { Xdimen x; hteg_xdimen_node(&x); }
  else HTEG_REF(xdimen_kind);
  if ((I) ≡ b100) HTEG_REF(param_kind);
```

⟨ cases to skip content 501 ⟩ +≡ (527)

```
case TAG(par_kind, b000): HTEG_PAR(b000); break;
case TAG(par_kind, b010): HTEG_PAR(b010); break;
case TAG(par_kind, b100): HTEG_PAR(b100); break;
case TAG(par_kind, b110): HTEG_PAR(b110); break;
```

B.17 Mathematics

⟨ skip macros 497 ⟩ +≡ (528)

```
#define HTEG_MATH(I)
  if ((I) & b001) hteg_hbox_node();
  { List l; hteg_list(&l); }
  if ((I) & b010) hteg_hbox_node();
  if ((I) & b100) { List l; hteg_param_list(&l); } else HTEG_REF(param_kind);
```

⟨ cases to skip content 501 ⟩ +≡ (529)

```
case TAG(math_kind, b000): HTEG_MATH(b000); break;
case TAG(math_kind, b001): HTEG_MATH(b001); break;
case TAG(math_kind, b010): HTEG_MATH(b010); break;
case TAG(math_kind, b100): HTEG_MATH(b100); break;
case TAG(math_kind, b101): HTEG_MATH(b101); break;
case TAG(math_kind, b110): HTEG_MATH(b110); break;
case TAG(math_kind, b011): case TAG(math_kind, b111): break;
```

B.18 Images

⟨ skip macros 497 ⟩ +≡ (530)

```
#define HTEG_IMAGE(I)
  { Image x = {0};
    List d;

    hteg_list(&d);
    if ((I) & b100) {
      if ((I) ≡ b111) { hteg_xdimen_node(&x.h);
        hteg_xdimen_node(&x.w);
      }
      else if ((I) ≡ b110) { hteg_xdimen_node(&x.w);
        x.hr = HTEG8;
      }
      else if ((I) ≡ b101) { hteg_xdimen_node(&x.h);
        x.wr = HTEG8;
      }
      else { x.hr = HTEG8;
        x.wr = HTEG8;
      }
      x.a = hteg_float32();
    }
    else if ((I) ≡ b011) { HTEG32(x.h.w);
      HTEG32(x.w.w);
    }
    else if ((I) ≡ b010) { HTEG32(x.w.w);
      x.a = hteg_float32();
    }
    else if ((I) ≡ b001) { HTEG32(x.h.w);
      x.a = hteg_float32();
    }
    HTEG16(x.n);
  }
```

⟨ cases to skip content 501 ⟩ +≡ (531)

```
  case TAG(image_kind, b001): HTEG_IMAGE(b001); break;
  case TAG(image_kind, b010): HTEG_IMAGE(b010); break;
  case TAG(image_kind, b011): HTEG_IMAGE(b011); break;
  case TAG(image_kind, b100): HTEG_IMAGE(b100); break;
  case TAG(image_kind, b101): HTEG_IMAGE(b101); break;
  case TAG(image_kind, b110): HTEG_IMAGE(b110); break;
  case TAG(image_kind, b111): HTEG_IMAGE(b111); break;
```

B.19 Links and Labels

⟨ skip macros 497 ⟩ +≡ (532)

```
#define HTEG_LINK(I)
  { uint16_t n;
    if (I & b100) n = HTEG8;
    if (I & b001) HTEG16(n); else n = HTEG8; }
```

⟨ cases to skip content 501 ⟩ +≡ (533)

```
  case TAG(link_kind, b000): HTEG_LINK(b000); break;
  case TAG(link_kind, b001): HTEG_LINK(b001); break;
  case TAG(link_kind, b010): HTEG_LINK(b010); break;
  case TAG(link_kind, b011): HTEG_LINK(b011); break;
  case TAG(link_kind, b100): HTEG_LINK(b100); break;
  case TAG(link_kind, b101): HTEG_LINK(b101); break;
  case TAG(link_kind, b110): HTEG_LINK(b110); break;
  case TAG(link_kind, b111): HTEG_LINK(b111); break;
```

B.20 Colors

⟨ cases to skip content 501 ⟩ +≡ (534)

```
  case TAG(color_kind, b000): (void) HTEG8; break;
```

B.21 Plain Lists, Texts, and Parameter Lists

⟨ shared skip functions 464 ⟩ +≡ (535)

```
  void hteg_size_boundary(Info info)
  { uint32_t n;
    info = info & #3;
    if (info ≡ 0) return;
    n = HTEG8;
    if (n ≠ #100 − info) QUIT("List␣size␣boundary␣byte␣0x%x␣does␣not␣m\
          atch␣info␣value␣%d␣at␣"SIZE_F, n, info, hpos − hstart);
  }
  uint32_t hteg_list_size(Info info)
  { uint32_t n = 0;
    info = info & #3;
    if (info ≡ 0) return 0;
    else if (info ≡ 1) n = HTEG8;
    else if (info ≡ 2) HTEG16(n);
    else if (info ≡ 3) HTEG32(n);
    else QUIT("List␣info␣%d␣must␣be␣0,␣1,␣2,␣or␣3", info);
    return n;
  }
  void hteg_list(List *l){ ⟨ skip the end byte z 493 ⟩
        if (KIND(z) ≠ list_kind ∧ KIND(z) ≠ param_kind)
```

```
    QUIT("List␣expected␣at␣0x%x", (uint32_t)(hpos − hstart));
  else if ((INFO(z) & #3) ≡ 0) { HBACK(1);
    l→s = 0; }
  else { uint32_t s;
    l→t = z;
    l→s = hteg_list_size(INFO(z));
    hteg_size_boundary(INFO(z));
    hpos = hpos − l→s;
    l→p = hpos − hstart;
    hteg_size_boundary(INFO(z));
    s = hteg_list_size(INFO(z));
    if (s ≠ l→s) QUIT("List␣sizes␣at␣"SIZE_F"␣and␣0x%x␣do␣not␣ma\
          tch␣0x%x␣!=␣0x%x", hpos − hstart, node_pos − 1, s, l→s);
  }
  ⟨ skip and check the start byte a 494 ⟩
  }
  void hteg_param_list(List *l)
  { if (KIND(*(hpos − 1)) ≠ param_kind) return;
    hteg_list(l);
  }
```

B.22 Adjustments

⟨ cases to skip content 501 ⟩ +≡ (536)

```
  case TAG(adjust_kind, b001): { List l; hteg_list(&l); } break;
```

B.23 Tables

⟨ skip macros 497 ⟩ +≡ (537)

```
#define HTEG_TABLE(I)
  { List l; hteg_list(&l); }
  { List l; hteg_list(&l); }
  if ((I) & b100) { Xdimen x; hteg_xdimen_node(&x); }
  else HTEG_REF(xdimen_kind)
```

⟨ cases to skip content 501 ⟩ +≡ (538)

```
  case TAG(table_kind, b000): HTEG_TABLE(b000); break;
  case TAG(table_kind, b001): HTEG_TABLE(b001); break;
  case TAG(table_kind, b010): HTEG_TABLE(b010); break;
  case TAG(table_kind, b011): HTEG_TABLE(b011); break;
  case TAG(table_kind, b100): HTEG_TABLE(b100); break;
  case TAG(table_kind, b101): HTEG_TABLE(b101); break;
  case TAG(table_kind, b110): HTEG_TABLE(b110); break;
  case TAG(table_kind, b111): HTEG_TABLE(b111); break;
```

case TAG(*item_kind*, *b000*): { **List** *l*; *hteg_list*(&*l*); } **break**;
case TAG(*item_kind*, *b001*): *hteg_content_node*(); **break**;
case TAG(*item_kind*, *b010*): *hteg_content_node*(); **break**;
case TAG(*item_kind*, *b011*): *hteg_content_node*(); **break**;
case TAG(*item_kind*, *b100*): *hteg_content_node*(); **break**;
case TAG(*item_kind*, *b101*): *hteg_content_node*(); **break**;
case TAG(*item_kind*, *b110*): *hteg_content_node*(); **break**;
case TAG(*item_kind*, *b111*): *hteg_content_node*(); { **uint8_t** *n*; *n* = HTEG8; }
  break;

B.24 Stream Nodes

⟨ skip macros 497 ⟩ +≡ (539)
#**define** HTEG_STREAM(*I*)
  { **List** *l*; *hteg_list*(&*l*); }
  if ((*I*) & *b010*) { **List** *l*; *hteg_param_list*(&*l*); } **else** HTEG_REF(*param_kind*);
  HTEG_REF(*stream_kind*);

⟨ cases to skip content 501 ⟩ +≡ (540)
case TAG(*stream_kind*, *b000*): HTEG_STREAM(*b000*); **break**;
case TAG(*stream_kind*, *b010*): HTEG_STREAM(*b010*); **break**;

B.25 References

⟨ skip macros 497 ⟩ +≡ (541)
#**define** HTEG_REF(*K*) **do** { **uint8_t** *n*; *n* = HTEG8; } **while** (*false*)

⟨ cases to skip content 501 ⟩ +≡ (542)
case TAG(*penalty_kind*, 0): HTEG_REF(*penalty_kind*); **break**;
case TAG(*kern_kind*, *b000*): HTEG_REF(*dimen_kind*); **break**;
case TAG(*kern_kind*, *b100*): HTEG_REF(*dimen_kind*); **break**;
case TAG(*kern_kind*, *b001*): HTEG_REF(*xdimen_kind*); **break**;
case TAG(*kern_kind*, *b101*): HTEG_REF(*xdimen_kind*); **break**;
case TAG(*ligature_kind*, 0): HTEG_REF(*ligature_kind*); **break**;
case TAG(*disc_kind*, 0): HTEG_REF(*disc_kind*); **break**;
case TAG(*glue_kind*, 0): HTEG_REF(*glue_kind*); **break**;
case TAG(*language_kind*, 0): HTEG_REF(*language_kind*); **break**;
case TAG(*rule_kind*, 0): HTEG_REF(*rule_kind*); **break**;
case TAG(*image_kind*, 0): HTEG_REF(*image_kind*); **break**;
case TAG(*leaders_kind*, 0): HTEG_REF(*leaders_kind*); **break**;
case TAG(*baseline_kind*, 0): HTEG_REF(*baseline_kind*); **break**;

B.26 Unknown nodes

⟨ skip functions 491 ⟩ +≡ (543)

```
static int hteg_unknown(Tag z)
{ int b, n;
  int8_t s;
  s = hnode_size[z];
  DBG(DBGTAGS, "Trying␣unknown␣tag␣0x%x␣at␣0x%x\n", z,
      (uint32_t)(hpos − hstart − 1));
  if (s ≡ 0) return 0;
  b = NODE_HEAD(s);
  n = NODE_TAIL(s);
  DBG(DBGTAGS, "Trying␣unknown␣node␣size␣%d␣%d\n", b, n);
  while (n > 0) { z = ∗(hpos − 1);
    if (KIND(z) ≡ xdimen_kind) { Xdimen x;
      hteg_xdimen_node(&x);
    }
    else if (KIND(z) ≡ param_kind) { List l; hteg_param_list(&l); }
    else if (KIND(z) ≤ list_kind) { List l; hteg_list(&l); }
    else hteg_content_node( );
    n−−;
  }
  while (b > 0) { z = HTEG8;
    b−−;
  }
  return 1;
}
```

C Code and Header Files

C.1 basetypes.h

To define basic types in a portable way, we create an include file. The macro _MSC_VER (Microsoft Visual C Version) is defined only if using the respective compiler.

⟨basetypes.h 544 ⟩ ≡ (544)

```
#ifndef __BASETYPES_H__
#define __BASETYPES_H__
#include <stdlib.h>
#include <stdio.h>
#ifndef _STDLIB_H
#define _STDLIB_H
#endif
#ifdef _MSC_VER
#include <windows.h>
#define uint8_t UINT8
#define uint16_t UINT16
#define uint32_t UINT32
#define uint64_t UINT64
#define int8_t INT8
#define int16_t INT16
#define int32_t INT32
#define bool BOOL
#define true (0 ≡ 0)
#define false (¬true)
#define __SIZEOF_FLOAT__ 4
#define __SIZEOF_DOUBLE__ 8
#define PRIx64 "I64x"
#pragma warning(disable:4244 4996 4127)
#else
#include <stdint.h>
#include <stdbool.h>
#include <inttypes.h>
#include <unistd.h>
#ifdef WIN32
```

```
#include <io.h>
#endif
#endif
  typedef float float32_t;
  typedef double float64_t;
#if __SIZEOF_FLOAT__ ≠ 4
#error float32␣type␣must␣have␣size␣4
#endif
#if __SIZEOF_DOUBLE__ ≠ 8
#error float64␣type␣must␣have␣size␣8
#endif
#define HINT_VERSION 2
#define HINT_MINOR_VERSION 1
#define AS_STR (X)#X
#define VERSION_AS_STR (X,Y) AS_STR (X) "." AS_STR (Y)
#define HINT_VERSION_STRING VERSION_AS_STR
  (HINT_VERSION,HINT_MINOR_VERSION)
#endif
```

C.2 format.h

The format.h file contains definitions of types, macros, variables and functions that are needed in other compilation units.

```
⟨ format.h 545 ⟩ ≡                                                    (545)
#ifndef _HFORMAT_H_
#define _HFORMAT_H_
  ⟨ debug macros 381 ⟩
  ⟨ debug constants 447 ⟩
  ⟨ hint macros 13 ⟩
  ⟨ hint basic types 6 ⟩
  ⟨ default names 418 ⟩
  extern const char *content_name[32];
  extern const char *definition_name[32];
  extern unsigned int debugflags;
  extern FILE *hlog;
  extern int max_fixed[32], max_default[32], max_ref[32], max_outline;
  extern int32_t int_defaults[MAX_INT_DEFAULT + 1];
  extern Dimen dimen_defaults[MAX_DIMEN_DEFAULT + 1];
  extern Xdimen xdimen_defaults[MAX_XDIMEN_DEFAULT + 1];
  extern Glue glue_defaults[MAX_GLUE_DEFAULT + 1];
  extern Baseline baseline_defaults[MAX_BASELINE_DEFAULT + 1];
  extern Label label_defaults[MAX_LABEL_DEFAULT + 1];
  extern ColorSet color_defaults[MAX_COLOR_DEFAULT + 1];
  extern signed char hnode_size[#100];
  extern uint8_t content_known[32];
#endif
```

C.3 tables.c

For maximum flexibility and efficiency, the file `tables.c` is generated by a C program. Here is the *main* program of `mktables`:

```
⟨ mktables.c 546 ⟩ ≡                                                    (546)
#include "basetypes.h"
#include "format.h"
  ⟨ skip macros 497 ⟩
  int max_fixed[32], max_default[32];
  int32_t int_defaults[MAX_INT_DEFAULT + 1] = {0};
  Dimen dimen_defaults[MAX_DIMEN_DEFAULT + 1] = {0};
  Xdimen xdimen_defaults[MAX_XDIMEN_DEFAULT + 1] = {{0}};
  Glue glue_defaults[MAX_GLUE_DEFAULT + 1] = {{{0}}};
  Baseline baseline_defaults[MAX_BASELINE_DEFAULT + 1] = {{{{0}}}};
  signed char hnode_size[#100] = {0};
  uint8_t content_known[32] = {0};
  ⟨ define content_name and definition_name 7 ⟩
  int main(void)
  { Kind k;
    int i;
    printf("#include␣\"basetypes.h\"\n"
    "#include␣\"format.h\"\n\n");
    ⟨ print content_name and definition_name 8 ⟩
    printf("int␣max_outline=-1;\n\n");
    ⟨ take care of variables without defaults 417 ⟩
    ⟨ define int_defaults 419 ⟩
    ⟨ define dimen_defaults 421 ⟩
    ⟨ define glue_defaults 425 ⟩
    ⟨ define xdimen_defaults 423 ⟩
    ⟨ define baseline_defaults 427 ⟩
    ⟨ define page defaults 433 ⟩
    ⟨ define stream defaults 431 ⟩
    ⟨ define range defaults 435 ⟩
    ⟨ define label_defaults 429 ⟩
    ⟨ define color_defaults 439 ⟩
    ⟨ print defaults 547 ⟩
    ⟨ initialize the hnode_size array 468 ⟩
    ⟨ print the hnode_size variable 465 ⟩
    ⟨ print the content_known variable 466 ⟩
    return 0;
  }
```

The following code prints the arrays containing the default values.

```
⟨ print defaults 547 ⟩ ≡                                                    (547)
  printf("int␣max_fixed[32]=␣{");
  for (k = 0; k < 32; k++)
  { printf("%d", max_fixed[k]); if (k < 31) printf(",␣"); }
  printf("};\n\n");
  printf("int␣max_default[32]=␣{");
  for (k = 0; k < 32; k++)
  { printf("%d", max_default[k]); if (k < 31) printf(",␣"); }
  printf("};\n\n");
  printf("int␣max_ref[32]=␣{");
  for (k = 0; k < 32; k++)
  { printf("%d", max_default[k]); if (k < 31) printf(",␣"); }
  printf("};\n\n");
```

Used in 546.

C.4 get.h

The `get.h` file contains function prototypes for all the functions that read the short format.

```
⟨ get.h 548 ⟩ ≡                                                             (548)
  ⟨ hint types 1 ⟩
  ⟨ directory entry type 357 ⟩
  ⟨ shared get macros 38 ⟩
  extern Entry *dir;
  extern uint16_t section_no, max_section_no;
  extern uint8_t *hpos, *hstart, *hend, *hpos0;
  extern uint64_t hin_size, hin_time;
  extern uint8_t *hin_addr;
  extern Label *labels;
  extern char *hin_name;
  extern bool hget_map(void);
  extern void hget_unmap(void);
  extern void new_directory(uint32_t entries);
  extern void hset_entry(Entry *e, uint16_t i,
      uint32_t size, uint32_t xsize, char *file_name);
  extern void hget_banner(void);
  extern void hget_section(uint16_t n);
  extern void hget_entry(Entry *e);
  extern void hget_directory(void);
  extern void hclear_dir(void);
  extern bool hcheck_banner(char *magic);
  extern int max_range;
  extern void hget_max_definitions(void);
  extern uint32_t hget_utf8(void);
  extern void hget_size_boundary(Info info);
```

```
extern uint32_t hget_list_size(Info info);
extern void hget_list(List *l);
extern float32_t hget_float32(void);
extern void hff_hpos(void);
extern uint32_t hff_list_pos, hff_list_size;
extern Tag hff_tag;
extern float32_t hteg_float32(void);
extern uint32_t hteg_list_size(Info info);    /* seems like these are declared
    static */
#if 0
extern void hteg_list(List *l);
extern void hteg_size_boundary(Info info);
#endif
```

C.5 get.c

```
⟨get.c 549 ⟩ ≡                                                           (549)
#include "basetypes.h"
#include <string.h>
#include <math.h>
#include <zlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "error.h"
#include "format.h"
#include "get.h"
  ⟨common variables 252 ⟩
  ⟨map functions 349 ⟩
  ⟨function to check the banner 336 ⟩
  ⟨directory functions 358 ⟩
  ⟨get file functions 337 ⟩
  ⟨shared get functions 53 ⟩
  ⟨shared skip functions 464 ⟩
```

C.6 put.h

The put.h file contains function prototypes for all the functions that write the short format.

```
⟨put.h 550 ⟩ ≡                                                           (550)
  ⟨put macros 345 ⟩
  ⟨hint macros 13 ⟩
  ⟨hint types 1 ⟩
  ⟨directory entry type 357 ⟩
```

```
extern Entry *dir;
extern uint16_t section_no, max_section_no;
extern uint8_t *hpos, *hstart, *hend, *hpos0;
extern int next_range;
extern RangePos *range_pos;
extern int next_range, max_range;
extern int *page_on;
extern Label *labels;
extern int first_label;
extern int max_outline;
extern Outline *outlines;
extern FILE *hout;
extern void new_directory(uint32_t entries);
extern void new_output_buffers(void);        /* declarations for the parser */
extern void hput_definitions_start(void);
extern void hput_definitions_end(void);
extern void hput_content_start(void);
extern void hput_content_end(void);
extern void hset_label(int n, int w);
extern Tag hput_link(int n, int c, int on);
extern void hset_outline(int m, int r, int d, uint32_t p);
extern void hput_label_defs(void);
extern void hput_tags(uint32_t pos, Tag tag);
extern Tag hput_glyph(Glyph *g);
extern Tag hput_xdimen(Xdimen *x);
extern Tag hput_int(int32_t p);
extern Tag hput_language(uint8_t n);
extern Tag hput_rule(Rule *r);
extern Tag hput_glue(Glue *g);
extern Tag hput_list(uint32_t size_pos, List *y);
extern uint8_t hsize_bytes(uint32_t n);
extern void hput_txt_cc(uint32_t c);
extern void hput_txt_font(uint8_t f);
extern void hput_txt_global(Ref *d);
extern void hput_txt_local(uint8_t n);
extern Info hput_box_dimen(Dimen h, Dimen d, Dimen w);
extern Info hput_box_shift(Dimen a);
extern Info hput_box_glue_set(int8_t s, float32_t r, Order o);
extern void hput_stretch(Stretch *s);
extern Tag hput_kern(Kern *k);
extern void hput_utf8(uint32_t c);
extern Tag hput_ligature(Lig *l);
extern Tag hput_disc(Disc *h);
extern Info hput_span_count(uint32_t n);
extern void hextract_image_dimens(int n, double *a, Dimen *w, Dimen
    *h);
```

```
extern Info hput_image_spec(uint32_t n, float32_t a, uint32_t wr, Xdimen
    *w, uint32_t hr, Xdimen *h);
extern int colors_i;
extern ColorSet colors_0, colors_n;
extern void color_init(void);
extern void hput_color_def(uint32_t pos, int n);
extern void hput_string(char *str);
extern void hput_range(uint8_t pg, bool on);
extern void hput_max_definitions(void);
extern Tag hput_dimen(Dimen d);
extern Tag hput_font_head(uint8_t f, char *n, Dimen s,
    uint16_t m, uint16_t y);
extern void hput_range_defs(void);
extern void hput_xdimen_node(Xdimen *x);
extern void hput_directory(void);
extern size_t hput_hint(char *str);
extern void hput_list_size(uint32_t n, int i);
extern uint32_t hput_unknown_def(uint32_t t, uint32_t b, uint32_t n);
extern Tag hput_unknown(uint32_t pos, uint32_t t, uint32_t b, uint32_t
    n);
extern int hcompress_depth(int n, int c);
```

C.7 put.c

```
⟨ put.c 551 ⟩ ≡                                                    (551)
#include "basetypes.h"
#include <string.h>
#include <ctype.h>
#include <math.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <zlib.h>
#include "error.h"
#include "format.h"
#include "put.h"
  ⟨ common variables 252 ⟩
  ⟨ shared put variables 273 ⟩
  ⟨ directory functions 358 ⟩
  ⟨ function to write the banner 339 ⟩
  ⟨ put functions 14 ⟩
```

C.8 lexer.l

The definitions for lex are collected in the file lexer.l

```
⟨ lexer.l 552 ⟩ ≡                                                          (552)
%{
#include "basetypes.h"
#include "error.h"
#include "format.h"
#include "put.h"
  ⟨ enable bison debugging 462 ⟩
#include "parser.h"
  ⟨ scanning macros 23 ⟩ ⟨ scanning functions 62 ⟩
  int yywrap(void) { return 1; }
#ifdef _MSC_VER
#pragma warning ( disable: 4267 )
#endif
%}
%option yylineno stack batch never − interactive
%option debug
%option nounistd nounput noinput noyy_top_state
  ⟨ scanning definitions 24 ⟩
%%
  ⟨ scanning rules 3 ⟩
[a-z]+        QUIT("Unexpected␣keyword␣'%s'␣in␣line␣%d",
                  yytext, yylineno);
.             QUIT("Unexpected␣character␣'%c'␣(0x%02X)␣in␣line␣%d",
                  yytext[0] > '␣' ? yytext[0] : '␣', yytext[0], yylineno);
%%
```

C.9 parser.y

The grammar rules for bison are collected in the file parser.y.

```
⟨ parser.y 553 ⟩ ≡                                                         (553)
%{
#include "basetypes.h"
#include <string.h>
#include <math.h>
#include "error.h"
#include "format.h"
#include "put.h"
  extern char **hfont_name;                          /* in common variables */
  ⟨ definition checks 390 ⟩
  extern void hset_entry(Entry *e, uint16_t i,
       uint32_t size, uint32_t xsize, char *file_name);
  ⟨ enable bison debugging 462 ⟩
```

extern int *yylex*(**void**);

⟨ parsing functions 386 ⟩
% }

%**union** { **uint32_t** *u*; **int32_t** *i*; **char** ∗*s*; **float64_t** *f*; **Glyph** *c*; **Dimen** *d*; **Stretch** *st*; **Xdimen** *xd*; **Kern** *kt*; **Rule** *r*; **Glue** *g*; **Image** *x*; **List** *l*; **Box** *h*; **Disc** *dc*; **Lig** *lg*; **Ref** *rf*; **Info** *info*; **Order** *o*;
bool *b*; }
%**error_verbose**
%**start** *hint*
⟨ symbols 2 ⟩
%%
⟨ parsing rules 5 ⟩
%%

C.10 shrink.c

`shrink` is a C program translating a HINT file in long format into a HINT file in short format.

⟨ shrink.c 554 ⟩ ≡ (554)

```
#include "basetypes.h"
#include <math.h>
#include <string.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/stat.h>
#ifdef WIN32
#include <direct.h>
#endif
#include <zlib.h>
#include "error.h"
#include "format.h"
#include "put.h"
  ⟨ enable bison debugging 462 ⟩
#include "parser.h"
  extern void yyset_debug(int lex_debug);
  extern int yylineno;
  extern FILE *yyin, *yyout;
  extern int yyparse(void);
  ⟨ put macros 345 ⟩
  ⟨ common variables 252 ⟩
  ⟨ shared put variables 273 ⟩
  ⟨ function to check the banner 336 ⟩
  ⟨ directory functions 358 ⟩
  ⟨ function to write the banner 339 ⟩
```

```
  ⟨ put functions 14 ⟩
#define SHRINK
#define DESCRIPTION "\nConvert␣a␣‘long’␣ASCII␣HINT␣file␣into\
      ␣a␣‘short’␣binary␣HINT␣file.\n"
  int main(int argc, char *argv[])
  { ⟨ local variables in main 449 ⟩
    in_ext = ".hint";
    out_ext = ".hnt";
    ⟨ process the command line 450 ⟩
    if (debugflags & DBGFLEX) yyset_debug(1);
    else yyset_debug(0);
#if YYDEBUG
    if (debugflags & DBGBISON) yydebug = 1;
    else yydebug = 0;
#endif
    ⟨ open the log file 452 ⟩
    ⟨ open the input file 453 ⟩
    ⟨ open the output file 454 ⟩
    yyin = hin;
    yyout = hlog;
    ⟨ read the banner 338 ⟩
    if (¬hcheck_banner("HINT")) QUIT("Invalid␣banner");
    yylineno++;
    DBG(DBGBISON | DBGFLEX, "Parsing␣Input\n");
    yyparse();
    hput_directory();
    ⟨ rewrite the file names of optional sections 373 ⟩
    hput_hint("created␣by␣shrink");
    ⟨ close the output file 457 ⟩
    ⟨ close the input file 456 ⟩
    ⟨ close the log file 458 ⟩
    return 0;
  }
```

C.11 stretch.c

stretch is a C program translating a HINT file in short format into a HINT file in long format.

```
⟨ stretch.c 555 ⟩ ≡                                                    (555)
#include "basetypes.h"
#include <math.h>
#include <string.h>
#include <ctype.h>
#include <zlib.h>
#include <sys/types.h>
#include <sys/stat.h>
```

```
#ifdef WIN32
#include <direct.h>
#endif
#include <fcntl.h>
#include "error.h"
#include "format.h"
#include "get.h"
  ⟨ get macros 19 ⟩
  ⟨ write macros 22 ⟩
  ⟨ common variables 252 ⟩
  ⟨ shared put variables 273 ⟩
  ⟨ map functions 349 ⟩
  ⟨ function to check the banner 336 ⟩
  ⟨ function to write the banner 339 ⟩
  ⟨ directory functions 358 ⟩
  ⟨ definition checks 390 ⟩
  ⟨ get function declarations 556 ⟩
  ⟨ write functions 21 ⟩
  ⟨ get file functions 337 ⟩
  ⟨ shared get functions 53 ⟩
  ⟨ get functions 18 ⟩
#define STRETCH
#define DESCRIPTION "\nConvert␣a␣‘short’␣binary␣HINT␣file␣in\
     to␣a␣‘long’␣ASCII␣HINT␣file.\n"
  int main(int argc, char *argv[])
  { ⟨ local variables in main 449 ⟩
    in_ext = ".hnt";
    out_ext = ".hint";
    ⟨ process the command line 450 ⟩
    ⟨ open the log file 452 ⟩
    ⟨ open the output file 454 ⟩
    ⟨ determine the stem_name from the output file_name 455 ⟩
    if (¬hget_map()) QUIT("Unable␣to␣map␣the␣input␣file");
    hpos = hstart = hin_addr;
    hend = hstart + hin_size;
    hget_banner();
    if (¬hcheck_banner("hint")) QUIT("Invalid␣banner");
    hput_banner("HINT", "created␣by␣stretch");
    hget_directory();
    hwrite_directory();
    hget_definition_section();
    hwrite_content_section();
    hwrite_aux_files();
    hget_unmap();
    ⟨ close the output file 457 ⟩
```

```
    DBG(DBGBASIC, "End␣of␣Program\n");
    ⟨ close the log file 458 ⟩
    return 0;
  }
```

In the above program, the get functions call the write functions and the write functions call some get functions. This requires function declarations to satisfy the define before use requirement of C. Some of the necessary function declarations are already contained in `get.h`. The remaining declarations are these:

```
⟨ get function declarations 556 ⟩ ≡                                    (556)
  extern void hget_xdimen_node(Xdimen *x);
  extern void hget_def_node(void);
  extern void hget_font_def(uint8_t f);
  extern void hget_content_section(void);
  extern Tag hget_content_node(void);
  extern void hget_glue_node(void);
  extern void hget_rule_node(void);
  extern void hget_hbox_node(void);
  extern void hget_vbox_node(void);
  extern void hget_param_list(List *l);
  extern int hget_txt(void);
  extern int hget_unknown(Tag a);
```

Used in 555 and 557.

C.12 `skip.c`

`skip` is a C program reading the content section of a HINT file in short format backwards.

```
⟨ skip.c 557 ⟩ ≡                                                       (557)
  #include "basetypes.h"
  #include <math.h>
  #include <string.h>
  #include <ctype.h>
  #include <zlib.h>
  #include <sys/types.h>
  #include <sys/stat.h>
  #include <fcntl.h>
  #include "error.h"
  #include "format.h"
  #if 1
  #include "get.h"
  #else
    ⟨ hint types 1 ⟩
    ⟨ directory entry type 357 ⟩
    ⟨ shared get macros 38 ⟩
  #endif
```

```
⟨ get macros 19 ⟩
⟨ write macros 22 ⟩
⟨ common variables 252 ⟩
⟨ shared put variables 273 ⟩
⟨ map functions 349 ⟩
⟨ function to check the banner 336 ⟩
⟨ directory functions 358 ⟩
⟨ shared get macros 38 ⟩
⟨ get file functions 337 ⟩
⟨ skip macros 497 ⟩
⟨ skip function declarations 558 ⟩
⟨ shared skip functions 464 ⟩
⟨ skip functions 491 ⟩
⟨ definition checks 390 ⟩
⟨ get function declarations 556 ⟩
⟨ write functions 21 ⟩
⟨ shared get functions 53 ⟩
⟨ get functions 18 ⟩
#define SKIP
#define DESCRIPTION "\n␣This␣program␣tests␣parsing␣a␣binary␣\
    HINT␣file␣in␣reverse␣direction.\n"
  int main(int argc, char *argv[])
  { ⟨ local variables in main 449 ⟩
    in_ext = ".hnt";
    out_ext = ".bak";
    ⟨ process the command line 450 ⟩
    ⟨ open the log file 452 ⟩
    hout = NULL;
    if (¬hget_map()) QUIT("Unable␣to␣map␣the␣input␣file");
    hpos = hstart = hin_addr;
    hend = hstart + hin_size;
    hget_banner();
    if (¬hcheck_banner("hint")) QUIT("Invalid␣banner");
    hget_directory();
    hget_definition_section();
    DBG(DBGBASIC, "Skipping␣Content␣Section\n");
    hteg_content_section();
    DBG(DBGBASIC, "Fast␣forward␣Content␣Section\n");
    hpos = hstart;
    while (hpos < hend) { hff_hpos();
      if (KIND(*(hpos − 1)) ≡ par_kind ∧ KIND(hff_tag) ≡ list_kind ∧ hff_list_size >
          0 ∧ ¬(INFO(hff_tag) & b100)) { uint8_t *p = hpos, *q;
        DBG(DBGTAGS, "Fast␣forward␣list␣at␣0x%x,␣size␣%d\n", hff_list_pos,
            hff_list_size);
        hpos = hstart + hff_list_pos;
```

```
        q = hpos + hff_list_size;
        while (hpos < q) hff_hpos( );
        DBG(DBGTAGS, "Fast␣forward␣list␣end␣at␣0x%x\n",
            (uint32_t)(hpos − hstart));
        hpos = p;
        DBG(DBGTAGS, "Continue␣at␣0x%x\n", (uint32_t)(hpos − hstart − 1));
      }
    }
    hget_unmap( );
    ⟨ close the log file 458 ⟩
    return 0;
  }
```

As we have seen already in the `stretch` program, a few function declarations are necessary to satisfy the define before use requirement of C.

```
⟨ skip function declarations 558 ⟩ ≡                                    (558)
  static void hteg_content_node(void);
  static void hteg_content(Tag z);
  static void hteg_xdimen_node(Xdimen *x);
  static void hteg_list(List *l);
  static void hteg_param_list(List *l);
  static void hteg_rule_node(void);
  static void hteg_hbox_node(void);
  static void hteg_vbox_node(void);
  static void hteg_glue_node(void);
  static int hteg_unknown(Tag z);
```

Used in 557.

D Format Definitions

D.1 Reading the Long Format

D.2 Writing the Long Format

D.3 Reading the Short Format

D.4 Writing the Short Format

Crossreference of Code

References

[1] Peter Deutsch and Jaen-Loup Gailly. *RFC1950, ZLIB Compressed Data Format Specification Version 3.3*. RFC Editor, United States, 1996.

[2] Jaen-loup Gailly and Mark Adler. zlib. `http://zlib.net/`.

[3] IANA Internet Assigned Numbers Authority, Los Angeles, CA. *IANA Charset MIB*, May 2014.

[4] IANA Internet Assigned Numbers Authority, Los Angeles, CA. *Character Sets Registry*, December 2018.

[5] IANA Internet Assigned Numbers Authority, Los Angeles, CA. *Language Tags*, April 2020.

[6] IEEE Computer Society Standards Committee. Working group of the Microprocessor Standards Subcommittee and American National Standards Institute. *IEEE standard for binary floating-point arithmetic*. IEEE Computer Society Press, 1985.

[7] IEEE Computer Society Standards Committee. Working group of the Microprocessor Standards Subcommittee and American National Standards Institute. IEEE Standard 754-2008. Technical report, August 2008.

[8] Donald E. Knuth. *The TEX book*. Computers & Typesetting, Volume A. Addison-Wesley Publishing Company, 1984.

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

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

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

[12] John R. Levine. *flex* & *bison*. O'Reilly Media, 2009. ISBN 978-0-596-15597-1.

[13] John R. Levine, Tony Mason, and Doug Brown. *lex* & *yacc*. O'Reilly Media, 2012.

[14] Ira McDonald. *IANA Charset MIB*. IETF Internet Engineering Task Force, rfc 3808 edition, June 2004.

[15] Addison Phillips and Mark Davis. Tags for Identifying Languages. RFC 5646, September 2009.

[16] Martin Ruckert. Computer Modern Roman fonts for ebooks. *TUGboat*, 37(3):277–280, 2016.

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

[18] Martin Ruckert. `web2w` *package on CTAN*. `https://ctan.org/pkg/web2w`, August 2017.

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

[20] Martin Ruckert. *WEB to cweb*. 2 edition, 2021. ISBN 979-8-54989510-2. `https://www.amazon.de/dp/B09BY85KG9`.

Index

C

D

E

F

G

H

X

Y

Z