HINT:
The File Format

HINT: The File Format

Reflowable
Output
for TEX

Für meine Mutter

Version 1.0

MARTIN RUCKERT *Munich University of Applied Sciences*

`Ruckert, Martin.`
`HINT: The File Format`
`Includes index.`
`ISBN 978-1079481594`

Internet page `https://www.cs.hm.edu/~ruckert` may contain current information about this book, downloadable software, and news.

`ruckert@cs.hm.edu`

ISBN-13: 978-1079481594

First printing: August 2019

Revision: 1873, Date: Thu, 05 Mar 2020

Preface

Late in summer 2017, with my new C based `cweb` implementation of TEX[6] in hand[13][12], 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[11], 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[14].

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 war 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 have 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[7] 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[8] 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 inter-word 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 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[5] to produce output in this format. Both are not described here; a survey describing them can be found in [14].

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 a trailing . . . `_t`.

⟨ hint types $_1$ ⟩ ≡ (1)

typedef struct { **uint32_t** *c*; **uint8_t** *f*; } **glyph_t**; Used in 430, 432, 434, 435, and 437.

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 extract a sequence of tokens. This is called “scanning”. We generate the procedure to do the scanning using the program `flex`[9] which is the GNU version of the common UNIX tool `lex`[10].

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 “−−− ⟹”. 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 177.

Reading the Long Format: − − − ⟹

⟨ symbols $_2$ ⟩ ≡ (2)

%token START "<"
%token END ">"
%token GLYPH "glyph"
%token < *u* > UNSIGNED
%token < *u* > REFERENCE Used in 433.

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 432.

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[9].

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`[9] which is the GNU version of the common UNIX tool `yacc`[10].

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 433.

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 172) where we find **uint32_t** *u* and **glyph_t** *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 8 ⟩ , ⟨ alternative kind names 9 ⟩ } **kind_t**;
#undef `DEF_KIND` Used in 333.

⟨ define *content_name* and *definition_name* 7 ⟩ ≡ (7)
#define `DEF_KIND` (C, D, N) $\#C$
  const char **content_name*[32] = { ⟨ kinds 8 ⟩ } ;
#undef `DEF_KIND`

```
  printf("const␣char␣*content_name[32]={");
  for (k = 0; k ≤ 31; k++) { printf("\"%s\"", content_name[k]);
    if (k < 31) printf(",␣");
  }
  printf("};\n\n");
#define DEF_KIND (C, D, N) #D
  const char *definition_name[#20] = { ⟨ kinds 8 ⟩ } ;
#undef DEF_KIND
  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 334.

```
⟨ kinds 8 ⟩ ≡                                                    (8)
  DEF_KIND(text, text, 0),
  DEF_KIND(list, list, 1),
  DEF_KIND(param, param, 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(hyphen, hyphen, 9),
  DEF_KIND(math, math, 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(display, display, 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(range, range, 27),
  DEF_KIND(undefined1, undefined1, 28),
  DEF_KIND(undefined2, undefined2, 29),
  DEF_KIND(undefined3, undefined3, 30),
```

DEF_KIND(*penalty*, *int*, 31) Used in 6 and 7.

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

⟨ alternative kind names 9 ⟩ ≡ (9)
font_kind = *glyph_kind*, *int_kind* = *penalty_kind*, *dimen_kind* = *kern_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 ⟩ +≡ (10)

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

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. The following macros are used to assemble and disassemble tags:

⟨ hint macros 11 ⟩ ≡ (11)

#**define** KIND(*T*) $(((T) \gg 3)$ & #1F)
#**define** NAME(*T*) *content_name*[KIND(*T*)]
#**define** INFO(*T*) ((*T*) & #7)
#**define** TAG(*K*, *I*) $(((K) \ll 3) \mid (I))$ Used in 333 and 430.

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: $\Longrightarrow\cdots$

⟨ put functions 12 ⟩ ≡ (12)

```
uint8_t hput_glyph(glyph_t *g)
{ info_t info;
  if (g→c ≤ #FF) { HPUT8(g→c); info = 1; }
  else if (g→c ≤ #FFFF) { HPUT16(g→c); info = 2; }
  else if (g→c ≤ #FFFFFF) { HPUT24(g→c); info = 3; }
  else { HPUT32(g→c); info = 4; }
  HPUT8(g→f);
  return TAG(glyph_kind, info);
}
```

Used in 431 and 434.

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 12 ⟩ +≡ (13)

```
void hput_tags(uint32_t pos, uint8_t 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* 14 ⟩ ≡ (14)

```
uint8_t 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 16, 90, 121, 131, 138, 150, 160, 203, 238, 291, 308, 314, and 327.

⟨ read and check the end byte *z* 15 ⟩ ≡ (15)

```
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 16, 90, 121, 138, 156, 160, 203, 238, 291, 308, 311, and 327.

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 nodes' keyword found in the *content_name* array. Then it calls *hget_content* to read the nodes 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.

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: ⋯ ⟹

⟨ get functions 16 ⟩ ≡ (16)

```
void hget_content_node(void)
{ ⟨ read the start byte a 14 ⟩ hwrite_start();
  hwritef("%s", content_name[KIND(a)]); hget_content(a);
  ⟨ read and check the end byte z 15 ⟩ hwrite_end();
}
void hget_content(uint8_t a)
{ uint32_t node_pos = hpos − hstart;
  switch (a)
  { ⟨ cases to get content 18 ⟩
    default: TAGERR(a); break;
  }
}
```

Used in 435.

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_t** 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: ··· ⟹

⟨ get macros 17 ⟩ ≡ (17)

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

Used in 435.

⟨ cases to get content 18 ⟩ ≡ (18)

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

Used in 16.

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* function used in *hwrite_end* is discussed in section 7.4.

Writing the Long Format: ⟹ − − −

⟨ write functions 19 ⟩ ≡ (19)

```
  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_end(void)
{ nesting--; hwritec('>');
  if (nesting ≡ 0 ∧ section_no ≡ 2) hwrite_range();
}
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_glyph(glyph_t *g)
{ char *n = hfont_name[g→f];
  hwrite_charcode(g→c); hwritef("␣*%d", g→f);
  if (n ≠ NULL) hwrite_comment(n);
}
```

Used in 435.

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: − − − ⟹

⟨ scanning macros 20 ⟩ ≡ (20)

```
#define SCAN_UDEC(S) yylval.u = strtoul(S, NULL, 10)
```

Used in 432.

Unsigned integers can be given in hexadecimal notation as well.

⟨ scanning definitions 21 ⟩ ≡ (21)

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

Used in 432.

⟨ scanning rules 3 ⟩ +≡ (22)

```
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 20 ⟩ +≡ (23)

```
#define SCAN_HEX(S) yylval.u = strtoul(S, NULL, 16)
```

Last not least, we add rules for signed integers.

⟨ symbols 2 ⟩ +≡ (24)

```
%token <i> SIGNED
%type <i> integer
```

⟨ scanning rules 3 ⟩ +≡ (25)

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

⟨ scanning macros 20 ⟩ +≡ (26)

```
#define SCAN_DEC(S) yylval.i = strtol(S, NULL, 10)
```

⟨ parsing rules 5 ⟩ +≡ (27)

```
  integer: SIGNED | UNSIGNED { RNG("number", $1, 0, INT32_MAX); };
```

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: $\Longrightarrow$ − − −

⟨ write functions 19 ⟩ +≡ (28)

```
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 20 ⟩ +≡ (29)

```
#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 21 ⟩ +≡ (30)
```
%x STR
```

⟨ symbols 2 ⟩ +≡ (31)
```
%token <s> STRING
```

⟨ scanning rules 3 ⟩ +≡ (32)
```
'                STR_START; BEGIN(STR);

<STR>{
'                STR_END; SCAN_STR; BEGIN(INITIAL); return STRING;
''               STR_ADD('\'');
[\x20-\x7E]      STR_ADD(yytext[0]);
.|\n             RNG("String␣character", yytext[0], #20, #7E);
  }
```

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

Writing the long format: ⟹ − − −

⟨ write functions 19 ⟩ +≡ (33)
```
void hwrite_string(char *str)
{ hwritec('␣');
  if (str ≡ NULL) hwritef("''");
  else
  { hwritec('\'');
    while (*str ≠ 0)
    { if (*str ≡ '\'') hwritec('\'');
      hwritec(*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 12 ⟩ +≡ (34)

```
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: ··· ⟹

⟨ get file macros 35 ⟩ ≡ (35)

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

Used in 428, 429, 435, and 437.

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 ⟩ +≡ (36)

```
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 21 ⟩ +≡ (37)
```
UTF8_1           [\x00-\x7F]
```

⟨ scanning macros 20 ⟩ +≡ (38)
#define `SCAN_UTF8_1`(S) $yylval.u = ((S)[0]\ \&\ {}^{\#}\mathtt{7F})$

- 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 21 ⟩ +≡ (39)
```
UTF8_2           [\xC0-\xDF][\x80-\xBF]
```

⟨ scanning macros 20 ⟩ +≡ (40)
#define `SCAN_UTF8_2`(S) $yylval.u = (((S)[0]\ \&\ {}^{\#}\mathtt{1F}) \ll 6) + ((S)[1]\ \&\ {}^{\#}\mathtt{3F})$

- 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 21 ⟩ +≡ (41)
```
UTF8_3           [\xE0-\xEF][\x80-\xBF][\x80-\xBF]
```

⟨ scanning macros 20 ⟩ +≡ (42)
#define `SCAN_UTF8_3`(S)
$yylval.u = (((S)[0]\ \&\ {}^{\#}\mathtt{0F}) \ll 12) + (((S)[1]\ \&\ {}^{\#}\mathtt{3F}) \ll 6) + ((S)[2]\ \&\ {}^{\#}\mathtt{3F})$

- 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 21 ⟩ +≡ (43)
```
UTF8_4           [\xF0-\xF7][\x80-\xBF][\x80-\xBF][\x80-\xBF]
```

⟨ scanning macros 20 ⟩ +≡ (44)
#define `SCAN_UTF8_4`(S)
$yylval.u = (((S)[0]\ \&\ {}^{\#}\mathtt{03}) \ll 18) + (((S)[1]\ \&\ {}^{\#}\mathtt{3F}) \ll 12) +$
$(((S)[2]\ \&\ {}^{\#}\mathtt{3F}) \ll 6) + ((S)[3]\ \&\ {}^{\#}\mathtt{3F})$

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 ⟩ +≡ (45)

```
%token <u> CHARCODE
```

⟨ scanning rules 3 ⟩ +≡ (46)

```
'''              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 ⟩ +≡ (47)

```
%type <s> string
```

⟨ parsing rules 5 ⟩ +≡ (48)

```
string: STRING | CHARCODE { static char s[2];
    RNG("String element", $1, 0x20, 0x7E); 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 19 ⟩ +≡ (49)

```
int hwrite_utf8(uint32_t c)
{ if (c < 0x80) { hwritec(c);
    return 1;
  }
  else if (c < 0x800) { hwritec(0xC0 | (c >> 6)); hwritec(0x80 | (c & 0x3F));
    return 2;
  }
  else if (c < 0x10000)
  { hwritec(0xE0 | (c >> 12));
    hwritec(0x80 | ((c >> 6) & 0x3F)); hwritec(0x80 | (c & 0x3F));
    return 3;
  }
  else if (c < 0x200000)
  { hwritec(0xF0 | (c >> 18)); hwritec(0x80 | ((c >> 12) & 0x3F));
    hwritec(0x80 | ((c >> 6) & 0x3F)); hwritec(0x80 | (c & 0x3F));
    return 4;
  }
  else RNG("character code", c, 0, 0x1FFFFF);
  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: ··· ⟹

⟨ get functions 16 ⟩ +≡ (50)

```
#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");
    }
  }
```

Writing the short format: ⟹ ⋯

⟨ put functions 12 ⟩ +≡ (51)

```
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 ⟩ +≡ (52)

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

⟨ scanning rules 3 ⟩ +≡ (53)

```
[+-]?[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[3][4]. We use the following definitions:

⟨ hint basic types 6 ⟩ +≡ (54)

```
#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
```

We expect a variable of type *float64_t* to have a binary representation using 64 bit. The most significant bit is the sign bit, then follow DBL_E_BITS = 11 bits for the exponent, and DBL_M_BITS = 52 bits for the mantissa. The sign bit is 1 for a negative number and 0 for a positive number. A floating point number is stored in normalized form which means that the mantissa is shifted such that it has exactly 52+1 bit not counting leading zeros. The leading bit is then always 1 and there is no need to store it. So 52 bits suffice. To store the exponent, the excess $q = 1023$ is added and the result is stored as an unsigned 11 bit number. For example if we regard the exponent bits and the mantissa bits as unsigned binary numbers e and m then the absolute value of such a floating point number can be expressed as $(1 + m * 2^{-52}) \cdot 2^{e-1023}$. We make similar assumptions about variables of type *float32_t* using the constants as defined above.

To convert the decimal representation of a floating point number to binary values of type *float64_t*, we use a C library function.

⟨ scanning macros 20 ⟩ +≡ (55)

```
#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 ⟩ +≡ (56)

```
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 #10 hexadecimal digits to the left. Here are the exact rules:

⟨ scanning rules 3 ⟩ +≡ (57)

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

⟨ scanning macros 20 ⟩ +≡ (58)

```
#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 59 ⟩ ≡ (59)

```
float64_t xtof(char *x)
{ int sign, digits, exp;
  uint64_t mantissa = 0;

  DBG(DBGFLOAT, "converting␣%s:\n", x);
  ⟨ read the optional sign 60 ⟩
  x = x + 2;                                /* skip "0x" */
  ⟨ read the mantissa 61 ⟩
```

```
  ⟨ normalize the mantissa 62 ⟩
  ⟨ read the optional exponent 63 ⟩
  ⟨ return the binary representation 64 ⟩
}                                                              Used in 432.
```

Now the pieces:

```
⟨ read the optional sign 60 ⟩ ≡                                       (60)
  if (*x ≡ '-') { sign = −1; x++; }
  else if (*x ≡ '+') { sign = +1; x++; }
  else sign = +1;
  DBG(DBGFLOAT, "\tsign=%d\n", sign);                          Used in 59.
```

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 61 ⟩ ≡                                            (61)
  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 59.
```

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 62 ⟩ ≡                                       (62)
  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 59.

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 63 ⟩ ≡ (63)

```
  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 59.

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

⟨ return the binary representation 64 ⟩ ≡ (64)

```
  { 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 59.

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`

Writing the long format: ⟹ − − −

```
⟨ write functions 19 ⟩ +≡                                                    (65)
#define MAX_HEX_DIGITS 12
  void hwrite_float64(float64_t d)
  { uint64_t bits, mantissa;
    int exp, digits;
    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 66 ⟩
    if (exp > MAX_HEX_DIGITS) ⟨ write large numbers 69 ⟩
    else if (exp ≥ 0) ⟨ write medium numbers 70 ⟩
    else ⟨ write small numbers 71 ⟩
  }
```

The extraction just reverses the creation of the binary representation.

```
⟨ extract mantissa and exponent 66 ⟩ ≡                                       (66)
  { 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 65.
```

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

```
⟨ extract mantissa and exponent 66 ⟩ +≡                                      (67)
  { 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 66 ⟩ +≡ (68)

```
  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 69 ⟩ ≡ (69)

```
  { 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 65.

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 70 ⟩ ≡ (70)

```
  { 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 65.

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 71 ⟩ ≡ (71)

```
  { 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 65.

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 12 ⟩ +≡ (72)

```
void hput_float32(float32_t d)
{ union { float32_t d; uint32_t bits; } u;
  u.d = d; HPUT32(u.bits);
}
```

⟨ get functions 16 ⟩ +≡ (73)

```
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_t** 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 ⟩ +≡ (74)

```
  typedef int32_t scaled_t;
#define ONE ((scaled_t)(1 << 16))
```

⟨ hint macros 11 ⟩ +≡ (75)

```
#define ROUND(X) ((int)((X) >= 0.0 ? floor((X) + 0.5) : ceil((X) - 0.5)))
```

Writing the long format: ⟹ − − −

⟨ write functions 19 ⟩ +≡ (76)

```
void hwrite_scaled(scaled_t x)
{ hwrite_float64(x/(float64_t)ONE);
}
```

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 ⟩ +≡ (77)

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

⟨ scanning rules 3 ⟩ +≡ (78)

```
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[5].

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 ⟩ +≡ (79)

```
  typedef scaled_t dimen_t;
#define MAX_DIMEN ((dimen_t)(#3FFFFFFF))
```

⟨ parsing rules 5 ⟩ +≡ (80)

```
  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: ⟹ − − −

⟨ write functions 19 ⟩ +≡ (81)

```
void hwrite_dimension(dimen_t 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: · · · ⟹

⟨ get functions 16 ⟩ +≡ (82)

```
void hget_dimen(void)
{ uint32_t d;
  HGET32(d);
  hwrite_dimension(d);
}
```

Writing the short format: ⟹ · · ·

⟨ put functions 12 ⟩ +≡ (83)

```
uint8_t hput_dimen(dimen_t 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 ⟩ +≡ (84)

```
typedef struct { dimen_t w; float32_t h, v; } xdimen_t;
```

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 ⟩ +≡ (85)

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

⟨ scanning rules 3 ⟩ +≡ (86)

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

⟨ parsing rules 5 ⟩ +≡ (87)

```
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 19 ⟩ +≡ (88)

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

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

Reading the short format: ··· ⟹

```
⟨ get macros 17 ⟩ +≡                                                        (89)
#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 16 ⟩ +≡                                                     (90)
  void hget_xdimen(uint8_t a, xdimen_t *x)
  {
    switch (a) {
#if 0                         /* currently the info value 0 is not supported */
    case TAG(xdimen_kind, b000):                        /* see section 10.5 */
      { REF(xdimen_kind, HGET8); } break;
#endif
    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));
    }
  }

  void hget_xdimen_node(xdimen_t *x)
  { ⟨ read the start byte a 14 ⟩
    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 15 ⟩
  }
```

Writing the short format: ⟹ ···

```
⟨ put functions 12 ⟩ +≡                                                     (91)
  uint8_t hput_xdimen(xdimen_t *x)
  { info_t info = b000;
    if (x→w ≡ 0 ∧ x→h ≡ 0.0 ∧ x→v ≡ 0.0) HPUT8(zero_xdimen_no);
    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_t *x)
{ uint32_t p = hpos++ - hstart;
  hput_tags(p, hput_xdimen(x));
}
```

2.8 Stretch and Shrink

In section 3.4, 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_t**. 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_t** to simplify conversion.

⟨ hint basic types 6 ⟩ +≡ (92)

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

Writing the short format: ⟹ ···

⟨ put functions 12 ⟩ +≡ (93)

```
void hput_stretch(stretch_t *s)
{ uint32_t mantissa, lowbits, sign, exponent;
  stch_t 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 ≥ ((uint32_t) 1 ≪ FLT_M_BITS))
      { exponent++;                                          /* adjust exponent */
        RNG("Float32␣exponent", exponent, 1, 2 ∗ FLT_EXCESS);
        mantissa = mantissa ≫ 1;
      }
      break;
    }
    DBG(DBGFLOAT, "␣round␣s=%d␣e=0x%x␣m=0x%x", sign, exponent, mantissa);
    st.u = sign | (exponent ≪ FLT_M_BITS) | mantissa | s→o;
    DBG(DBGFLOAT, "float␣%f␣hex␣0x%x\n", st.f, st.u);
    HPUT32(st.u);
  }
```

Reading the short format: · · · ⟹

⟨ get macros 17 ⟩ +≡ (94)

```
#define HGET_STRETCH(S)
  { stch_t st; HGET32(st.u); S.o = st.u & 3;
    st.u &= ∼3;
    S.f = st.f; }
```

Reading the long format: − − − ⟹

⟨ symbols 2 ⟩ +≡ (95)

```
%token FIL "fil"
%token FILL "fill"
%token FILLL "filll"
%type < st > stretch
%type < o > order
```

⟨ scanning rules 3 ⟩ +≡ (96)

```
fil           return FIL;
fill          return FILL;
filll         return FILLL;
```

⟨ parsing rules 5 ⟩ +≡ (97)

```
  order: PT { $$ = normal_o; }
    | FIL { $$ = fil_o; } | FILL { $$ = fill_o; } | FILLL { $$ = filll_o; };
  stretch: number order { $$.f = $1; $$.o = $2; };
```

Writing the long format: ⟹ − − −

⟨ write functions 19 ⟩ +≡ (98)

```
void hwrite_order(order_t 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_t *s)
{ hwrite_float64(s→f);
  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 4 can be used for general integers (see section 10) that need four byte of storage.

Reading the long format: − − − ⟹

⟨ symbols 2 ⟩ +≡ (99)

```
%token PENALTY "penalty"
%token INTEGER "int"
%type <i> penalty
```

⟨ scanning rules 3 ⟩ +≡ (100)

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

⟨ parsing rules 5 ⟩ +≡ (101)

```
penalty: integer { RNG("Penalty", $1, −20000, +20000); $$ = $1; };
content_node: start PENALTY penalty END { hput_tags($1, hput_int($3)); };
```

Reading the short format: · · · ⟹

⟨ cases to get content 18 ⟩ +≡ (102)

```
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;
```

⟨ get macros 17 ⟩ +≡ (103)

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

Writing the short format: ⟹ ⋯

⟨ put functions 12 ⟩ +≡ (104)

```
uint8_t hput_int(int32_t n)
{ info_t info;
  if (n ≥ 0)
  { if (n < #80) { HPUT8(n); info = 1; }
    else if (n < #8000) { HPUT16(n); info = 2; }
    else { HPUT32(n); info = 4; }
  }
  else
  { if (n ≥ −#80) { HPUT8(n); info = 1; }
    else if (n ≥ −#8000) { HPUT16(n); info = 2; }
    else { HPUT32(n); info = 4; }
  }
  return TAG(int_kind, info);
}
```

3.2 Mathematics

Being able to handle mathematics nicely is one of the primary features of TEX and so you should expect the same from HINT. A math node occurs before (*info* = *b100*) and after (*info* = *b010*) a mathematical formula in a horizontal list. A math node also features a width which represents the amount of surrounding space. A zero width can be omitted. In the short format, the *b001* bit is set if the width is present.

⟨ hint types 1 ⟩ +≡ (105)

```
typedef struct { bool on; dimen_t w; } math_t;
```

Reading the long format: − − − ⟹

⟨ symbols 2 ⟩ +≡ (106)

```
%token MATH "math"
%token ON "on"
%token OFF "off"
%type <m> math
```

⟨ scanning rules 3 ⟩ +≡ (107)

```
math        return MATH;
on          return ON;
off         return OFF;
```

⟨ parsing rules 5 ⟩ +≡ (108)

```
math: ON dimension { $$.on = true; $$.w = $2; };
math: OFF dimension { $$.on = false; $$.w = $2; };
math: ON { $$.on = true; $$.w = 0; };
math: OFF { $$.on = false; $$.w = 0; };
content_node: start MATH math END { hput_tags($1, hput_math(&($3))); };
```

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

⟨ write functions 19 ⟩ +≡ (109)

```
void hwrite_math(math_t *m)
{ if (m→on) hwritef("␣on"); else hwritef("␣off");
  if (m→w ≠ 0) hwrite_dimension(m→w);
}
```

Reading the short format: · · · $\Longrightarrow$

⟨ cases to get content 18 ⟩ +≡ (110)

```
case TAG(math_kind, b100):
  { math_t m; HGET_MATH(b100, m); hwrite_math(&m); } break;
case TAG(math_kind, b010):
  { math_t m; HGET_MATH(b010, m); hwrite_math(&m); } break;
case TAG(math_kind, b101):
  { math_t m; HGET_MATH(b101, m); hwrite_math(&m); } break;
case TAG(math_kind, b011):
  { math_t m; HGET_MATH(b011, m); hwrite_math(&m); } break;
```

⟨ get macros 17 ⟩ +≡ (111)

```
#define HGET_MATH(I, M)
  if ((I) & b001) HGET32(M.w); else M.w = 0;
  if ((I) & b100) M.on = true;
  if ((I) & b010) M.on = false;
```

Writing the short format: $\Longrightarrow$ · · ·

⟨ put functions 12 ⟩ +≡ (112)

```
uint8_t hput_math(math_t *m)
{ info_t info;
  if (m→on) info = b100; else info = b010;
  if (m→w ≠ 0) { HPUT32(m→w); info |= b001; }
  return TAG(math_kind, info);
}
```

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 $-2^{30} = {}^{\#}\texttt{C0000000}$.

⟨ hint macros 11 ⟩ +≡ (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 5.4) 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_t** (the writing is done with a separate routine), parsing a *rule_node* will always include writing it.

⟨ hint types 1 ⟩ +≡ (114)

```
typedef struct { dimen_t h, d, w; } rule_t;
```

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 19 ⟩ +≡                                          (118)
  static void hwrite_rule_dimension(dimen_t d, char c)
  { if (d ≡ RUNNING_DIMEN) hwritef(" %c", c);
    else hwrite_dimension(d);
  }
  void hwrite_rule(rule_t *r)
  { hwrite_rule_dimension(r→h, '|');
    hwrite_rule_dimension(r→d, '|');
    hwrite_rule_dimension(r→w, '_');
  }
```

Reading the short format: · · · ⟹

```
⟨ cases to get content 18 ⟩ +≡                                      (119)
case TAG(rule_kind, b011): { rule_t r; HGET_RULE(b011, r); hwrite_rule(&(r)); }
  break;
case TAG(rule_kind, b101): { rule_t r; HGET_RULE(b101, r); hwrite_rule(&(r)); }
  break;
case TAG(rule_kind, b001): { rule_t r; HGET_RULE(b001, r); hwrite_rule(&(r)); }
  break;
case TAG(rule_kind, b110): { rule_t r; HGET_RULE(b110, r); hwrite_rule(&(r)); }
  break;
case TAG(rule_kind, b111): { rule_t r; HGET_RULE(b111, r); hwrite_rule(&(r)); }
  break;
```

```
⟨ get macros 17 ⟩ +≡                                                (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 16 ⟩ +≡                                             (121)
  void hget_rule_node(void)
  { ⟨ read the start byte a 14 ⟩
    if (KIND(a) ≡ rule_kind)
    { rule_t 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 15 ⟩
  }
```

Writing the short format: $\Longrightarrow\cdots$

⟨ put functions 12 ⟩ +≡ (122)

```
uint8_t hput_rule(rule_t *r)
{ info_t 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 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, and we could equally well list them in section 5 as composite 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 inter word 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 ⟩ +≡ (123)

```
typedef struct { xdimen_t w; stretch_t p, m; } glue_t;
```

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

⟨ hint macros 11 ⟩ +≡ (124)

```
#define ZERO_GLUE(G)
  ((G).w.w ≡ 0 ∧ (G).w.h ≡ 0.0 ∧ (G).w.v ≡ 0.0 ∧ (G).p.f ≡ 0.0 ∧ (G).m.f ≡ 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 a node for the zero glue might be omitted entirely.

Reading the long format: − − − ⟹

⟨ symbols 2 ⟩ +≡ (125)

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

⟨ scanning rules 3 ⟩ +≡ (126)

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

⟨ parsing rules 5 ⟩ +≡ (127)

```
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 19 ⟩ +≡ (128)

```
void hwrite_plus(stretch_t *p)
{ if (p→f ≠ 0.0) { hwritef("␣plus"); hwrite_stretch(p); }
}
void hwrite_minus(stretch_t *m)
{ if (m→f ≠ 0.0) { hwritef("␣minus"); hwrite_stretch(m); }
}
void hwrite_glue(glue_t *g)
{ hwrite_xdimen(&(g→w)); hwrite_plus(&g→p); hwrite_minus(&g→m);
}
void hwrite_glue_node(glue_t *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 18 ⟩ +≡ (129)

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

⟨ get macros 17 ⟩ +≡ (130)

```
#define HGET_GLUE(I, G){
        if (I ≡ b111) hget_xdimen_node(&((G).w));
        else
        { (G).w.h = 0.0; (G).w.v = 0.0;
          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; }
```

```
⟨ get functions 16 ⟩ +≡                                                   (131)
  void hget_glue_node(void)
  { ⟨ read the start byte a 14 ⟩
    if (KIND(a) ≠ glue_kind) { hpos−−; return; }
    if (INFO(a) ≡ b000) { hwrite_ref_node(glue_kind, HGET8); }
    else { glue_t g; HGET_GLUE(INFO(a), g); hwrite_glue_node(&g); }
    ⟨ read and check the end byte z 15 ⟩
  }
```

Writing the short format: ⟹ · · ·

```
⟨ put functions 12 ⟩ +≡                                                   (132)
  uint8_t hput_glue(glue_t *g)
  { info_t info = b000;
    if (ZERO_GLUE(*g)) { HPUT32(g→w.w); info = b100;
    }                                          /* this is not a reference */
    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_xdimen_node(&(g→w));
      hput_stretch(&g→p); hput_stretch(&g→m);
      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 four different types of lists: plain lists that use the kind value *list_kind*, text lists that use the kind value *text_kind*, adjustments that use the kind value *adjust_kind*, and parameter lists that use the kind value *param_kind*. A description of the first two types of lists follows here. Adjustments are just plain lists of vertical material described in section 5.10, and 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 tradeoffs 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 *info* value to indicate the number of byte used to store the list size: A list with $0 < \mathit{info} \leq 5$ uses $\mathit{info} - 1$ byte to store the size. The info value zero is reserved for references to predefined lists (which are currently not implemented).

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`, #`FD`, and #`FC` which can not be confused with valid tag bytes and indicate that the size is stored using 1, 2, 3, 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 $7 \geq i \geq 4$ and can not be confused with valid penalty nodes which use only the info values 0, 1, and 2.

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 ⟩ +≡ (133)

typedef struct { **kind_t** *k*; **uint32_t** *p*; **uint32_t** *s*; } **list_t**;

The major drawback of his 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* or *adjust_kind*.

Not uncommon are empty lists; these are the only lists that can be stored using *info* = 1; such a list has zero bytes of size information, and implicitly its size is zero. The *info* value 0 is not used since we do not use predefined plain lists.

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: − − − ⟹

⟨ symbols 2 ⟩ +≡ (134)
%**type** $< l >$ *list*
%**type** $< u >$ *position content_list*

⟨ parsing rules 5 ⟩ +≡ (135)
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
{ `$$`.*k* = *list_kind*; `$$`.*p* = `$3`; `$$`.*s* = (*hpos* − *hstart*) − `$3`;
hput_tags(`$1`, *hput_list*(`$1` + 1, &(`$$`))); };

Writing the long format: ⟹ − − −

⟨ write functions 19 ⟩ +≡ (136)

```
void hwrite_list(list_t *l)
{ uint32_t h = hpos − hstart, e = hend − hstart;   /* save hpos and hend */
  hpos = l→p + hstart; hend = hpos + l→s;
  if (l→k ≡ list_kind) ⟨ write a list 137 ⟩
  else if (l→k ≡ text_kind) ⟨ write a text 147 ⟩
  else QUIT("List␣expected␣got␣%s", content_name[l→k]);
  hpos = hstart + h; hend = hstart + e;   /* restore hpos and hend */
}
```

⟨ write a list 137 ⟩ ≡ (137)

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

Used in 136.

Reading the short format: · · · ⟹

⟨ get functions 16 ⟩ +≡ (138)

```
void hget_size_boundary(info_t info)
{ uint32_t n;
  if (info < 2) return;
  n = HGET8;
  if (n − 1 ≠ #100 − info)
    QUIT("Size␣boundary␣byte␣0x%x␣with␣info␣value␣%d␣at␣"SIZE_F, n,
        info, hpos − hstart − 1);
}
uint32_t hget_list_size(info_t info)
{ uint32_t n;
  if (info ≡ 1) return 0;
  else if (info ≡ 2) n = HGET8;
  else if (info ≡ 3) HGET16(n);
  else if (info ≡ 4) HGET24(n);
  else if (info ≡ 5) HGET32(n);
  else QUIT("List␣info␣%d␣must␣be␣1,␣2,␣3,␣4,␣or␣5", info);
  return n;
}
void hget_list(list_t *l)
{ if (KIND(*hpos) ≠ list_kind ∧ KIND(*hpos) ≠ adjust_kind ∧
```

```
    KIND(*hpos) ≠ text_kind ∧
         KIND(*hpos) ≠ param_kind)
    { l→p = hpos − hstart; l→s = 0; l→k = list_kind; }
    else { ⟨ read the start byte a 14 ⟩
       l→k = KIND(a);
       HGET_LIST(INFO(a), *l);
       ⟨ read and check the end byte z 15 ⟩
       DBG(DBGNODE, "Get␣list␣at␣0x%x␣size=%u\n", l→p, l→s);
    }
  }
```

⟨ get macros 17 ⟩ +≡ (139)

```
#define HGET_LIST(I, L) (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: ⟹ · · ·

⟨ put functions 12 ⟩ +≡ (140)

```
  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 if (n < #1000000) return 3;
    else return 4;
  }
  void hput_list_size(uint32_t n, int i)
  { if (i ≡ 0) ;
    else if (i ≡ 1) HPUT8(n);
    else if (i ≡ 2) HPUT16(n);
    else if (i ≡ 3) HPUT24(n);
    else HPUT32(n);
  }
  uint8_t hput_list(uint32_t start_pos, list_t *l)
  { if (l→s ≡ 0) { hpos = hstart + start_pos;
       return TAG(l→k, 1); }
    else
    { uint32_t list_end = hpos − hstart;
       info_t i = l→p − start_pos − 1;    /* number of byte allocated for size */
       info_t j = hsize_bytes(l→s);         /* number of byte needed for size */
```

```
    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);
      memmove(hstart + l→p + d, hstart + l→p, l→s);
      l→p = l→p + d; list_end = list_end + d;
    }
    hpos = hstart + start_pos; hput_list_size(l→s, j); HPUT8(#100 − j);
    hpos = hstart + list_end; HPUT8(#100 − j); hput_list_size(l→s, j);
    return TAG(l→k, j + 1);
  }
}
```

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 *text_kind* $\equiv 0$, *list_kind* $\equiv 1$, *param_kind* $\equiv 2$, *xdimen_kind* $\equiv 3$, and *adjust_kind* $\equiv 4$ were chosen on page 5. Texts, lists, parameter lists, and extended dimensions occur only *inside* of content nodes, but are not content nodes in their own right; 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* $\equiv 4$) 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 four #`FC`, #`FD`, #`FE`, and #`FF` are used as boundary marks for the text size and therefore we use only #`FB` as control code.

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 using the kind value *text_kind*.
- 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 popping and pushing 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 inter word 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 hyphen 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, #0E, #0E, #0F, and #10 are also followed by a second parameter byte. They are used to reference the global definitions of penalty, kern, ligature, hyphen, glue, math, 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), “\H*n*\” (hyphen), “\G*n*\” (glue), “\M*n*\” (math), “\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 inter-word 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 “\@”.

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

⟨ hint types 1 ⟩ +≡ (141)

```
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_t;
```

Reading the long format: − − − ⟹

⟨ scanning definitions 21 ⟩ +≡ (142)

```
%x TXT
```

⟨ symbols 2 ⟩ +≡ (143)

```
%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 ⟩ +≡ (144)

```
\"                 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;
\\"@"              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;
\\H[0-9]+\\        SCAN_REF(hyphen_kind); return TXT_GLOBAL;
\\G[0-9]+\\        SCAN_REF(glue_kind); return TXT_GLOBAL;
\\M[0-9]+\\        SCAN_REF(math_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 20 ⟩ +≡ (145)

```
#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); scan_level++;
#define SCAN_END
  if (scan_level−−) yy_pop_state();
  elseQUIT("Too␣many␣'>'␣in␣line␣%d", yylineno)
#define SCAN_TXT_START BEGIN(TXT)
#define SCAN_TXT_END BEGIN(INITIAL)
```

⟨ parsing rules 5 ⟩ +≡ (146)

```
  list: TXT_START position
        { hpos += 4;        /* start byte, two size byte, and boundary byte */
        } text TXT_END
        { $$.k = text_kind; $$.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 147 ⟩ ≡ (147)

```
{ 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 136.

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

⟨ write functions 19 ⟩ +≡ (148)

```
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("\\-");
    default:
      if (option_utf8) return hwrite_utf8(c);
      else return hwritef("\\C%d\\", c);
  }
}
```

Reading the short format: $\cdots \Longrightarrow$

⟨ get macros 17 ⟩ +≡ (149)

```
#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 16 ⟩ +≡ (150)

```
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(hyphen_kind, "H");
    case txt_global + 5: HGET_GREF(glue_kind, "G");
    case txt_global + 6: HGET_GREF(math_kind, "M");
    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 14 ⟩
          i = hwritef("<%s", content_name[KIND(a)]); hget_content(a);
          ⟨ read and check the end byte z 15 ⟩
          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 12 ⟩ +≡ (151)

```
  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_t *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 hyphen_kind: HPUT8(txt_global + 4); break;
```

```
    case glue_kind: HPUT8(txt_global + 5); break;
    case math_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);
  }
```

⟨ hint types 1 ⟩ +≡ (152)

```
  typedef struct { kind_t k; uint8_t n; } ref_t;
```

5 Composite Nodes

The nodes that we consider in this section contain other nodes for example a glue node or a list of node. When we implement the parsing routines for composite nodes in the long format, we have to take into account that parsing such a glue node or list node will already write the glue or 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.

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 elements inside the box.

⟨ hint types $_1$ ⟩ +≡ (153)

typedef struct
{ **dimen_t** *h*, *d*, *w*, *a*; *float32_t* *r*; **int8_t** *s*, *o*; **list_t** *l*; } **box_t**;

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 elements horizontally, the shift amount a shifts the box down; a vertical box aligns the reference points vertically, the 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: $---\Longrightarrow$

⟨ symbols $_2$ ⟩ +≡ (154)

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

⟨ scanning rules $_3$ ⟩ +≡ (155)

```
hbox                return HBOX;

vbox                return VBOX;
```

⟨ parsing rules $_5$ ⟩ +≡ (156)

```
box_dimen: dimension dimension dimension
      { $$ = hput_box_dimen($1, $2, $3); };
box_shift: { $$ = b000; } | dimension { $$ = hput_box_shift($1); };
box_glue_set: { $$ = 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: $\Longrightarrow---$

⟨ write functions $_{19}$ ⟩ +≡ (157)

```
void hwrite_box(box_t *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); hwrite_order(b→o);
  }
  hwrite_list(&(b→l));
}
```

Reading the short format: · · · ⟹

⟨ cases to get content 18 ⟩ +≡ (158)

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

⟨ get macros 17 ⟩ +≡ (159)

```
#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 16 ⟩ +≡ (160)

```
  void hget_hbox_node(void)
```

```
{ box_t b;
  ⟨ read the start byte a 14 ⟩
  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 15 ⟩
  hwrite_start(); hwritef("hbox"); hwrite_box(&b); hwrite_end();
}
void hget_vbox_node(void)
{ box_t b;
  ⟨ read the start byte a 14 ⟩
  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 15 ⟩
  hwrite_start(); hwritef("vbox"); hwrite_box(&b); hwrite_end();
}
```

Writing the short format: ⟹ ⋯

⟨ put functions 12 ⟩ +≡ (161)

```
info_t hput_box_dimen(dimen_t h, dimen_t d, dimen_t w)
{ info_t i; HPUT32(h);
  if (d ≠ 0) { HPUT32(d); i = b001; } else i = b000;
  HPUT32(w);
  return i;
}
info_t hput_box_shift(dimen_t a)
{ if (a ≠ 0) { HPUT32(a); return b010; } else return b000;
}
info_t hput_box_glue_set(int8_t s, float32_t r, order_t 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 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 its number of lines which, in turn, depends on `hsize`. It is not possible to merge this 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 of type *hset_kind* or *vset_kind*, info bit *b100* indicates if set, a complete extended dimension, and if unset, a reference to a predefined extended dimension for the target size; info bit *b010* indicates a nonzero shift amount, and info bit *b001* indicates a nonzero depth.

In the short format of a box of type *hpack_kind* or *vset_kind*, info bit *b100* indicates if set, a complete extended dimension, and if unset, a reference to a predefined extended dimension for the target size; info bit *b010* indicates if set, an additional target size, and if unset an exact target size; and info bit *b001* indicates a maximum depth less than `MAX_DIMEN` (this is used only for vertical lists).

Reading the long format: − − − ⟹

⟨ symbols $_2$ ⟩ +≡ (162)
%**token** HPACK "hpack"
%**token** HSET "hset"
%**token** VPACK "vpack"
%**token** VSET "vset"
%**token** ADD "add"
%**token** TO "to"
%**type** < *info* > *xbox box_goal hpack vpack*

⟨ scanning rules $_3$ ⟩ +≡ (163)

`hpack`	**return** HPACK;
`hset`	**return** HSET;
`vpack`	**return** VPACK;
`vset`	**return** VSET;
`add`	**return** ADD;
`to`	**return** TO;

⟨ parsing rules $_5$ ⟩ +≡ (164)
box_flex: *plus minus* { *hput_stretch*(&($1)); *hput_stretch*(&($2)); };
xbox: *xdimen_ref box_dimen box_shift box_flex list* { $$ = $2 | $3; }
| *xdimen_node box_dimen box_shift box_flex list* { $$ = $2 | $3 | *b100*; };
box_goal: TO *xdimen_ref* { $$ = *b000*; }
| ADD *xdimen_ref* { $$ = *b010*; }

```
    | TO xdimen_node { $$ = b100; }
    | ADD xdimen_node { $$ = b110; };
  hpack: box_goal list;
  vpack: box_goal list
    | box_goal dimension { if ($2 ≠ MAX_DIMEN) HPUT32($2); } list
      { if ($2 ≠ MAX_DIMEN) $$ = $1 | b001; else $$ = $1; };
  content_node: start HSET xbox END { hput_tags($1, TAG(hset_kind, $3)); }
    | start HPACK hpack END { hput_tags($1, TAG(hpack_kind, $3)); }
    | start VSET xbox END { hput_tags($1, TAG(vset_kind, $3)); }
    | start VPACK vpack END { hput_tags($1, TAG(vpack_kind, $3)); };
```

Reading the short format: · · · ⟹

⟨ cases to get content 18 ⟩ +≡ (165)

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

⟨ get macros 17 ⟩ +≡ (166)

```
#define HGET_SET(I)
  if ((I) & b100) { xdimen_t x;
```

```
    hget_xdimen_node(&x); hwrite_xdimen_node(&x); }
  else HGET_REF(xdimen_kind)
  { dimen_t h; HGET32(h); hwrite_dimension(h); }
  { dimen_t d; if ((I) & b001) HGET32(d); else d = 0; hwrite_dimension(d); }
  { dimen_t w; HGET32(w); hwrite_dimension(w); }
  if ((I) & b010) { dimen_t a; HGET32(a); hwrite_dimension(a); }
  { stretch_t p; HGET_STRETCH(p); hwrite_plus(&p); }
  { stretch_t m; HGET_STRETCH(m); hwrite_minus(&m); }
  { list_t l; hget_list(&l); hwrite_list(&l); }

#define HGET_PACK(I)
  if ((I) & b010) hwritef("␣add"); else hwritef("␣to");
  if ((I) & b100) { xdimen_t x;
    hget_xdimen_node(&x); hwrite_xdimen_node(&x); } else
    HGET_REF(xdimen_kind);
  if ((I) & b001) { dimen_t d; HGET32(d); hwrite_dimension(d); }
  { list_t l; hget_list(&l); hwrite_list(&l); }
```

5.3 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 ⟩ +≡ (167)

```
  typedef struct { bool x; xdimen_t d; } kern_t;
```

Reading the long format: − − − ⟹

⟨ symbols 2 ⟩ +≡ (168)
%**token** KERN "kern"
%**token** EXPLICIT "!"
%**type** < *b* > *explicit*
%**type** < *kt* > *kern*

⟨ scanning rules 3 ⟩ +≡ (169)
kern **return** KERN;

! **return** EXPLICIT;

⟨ parsing rules 5 ⟩ +≡ (170)
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: ⟹ − − −

⟨ write functions 19 ⟩ +≡ (171)
void *hwrite_explicit*(**bool** *x*)
{ **if** (*x*) *hwritef*("␣!"); }

void *hwrite_kern*(**kern_t** ∗*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: · · · ⟹

⟨ cases to get content 18 ⟩ +≡ (172)
case TAG(*kern_kind*, *b010*): { **kern_t** *k*; HGET_KERN(*b010*, *k*); } **break**;
case TAG(*kern_kind*, *b011*): { **kern_t** *k*; HGET_KERN(*b011*, *k*); } **break**;
case TAG(*kern_kind*, *b110*): { **kern_t** *k*; HGET_KERN(*b110*, *k*); } **break**;
case TAG(*kern_kind*, *b111*): { **kern_t** *k*; HGET_KERN(*b111*, *k*); } **break**;

⟨ get macros 17 ⟩ +≡ (173)
#**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: $\Longrightarrow \cdots$

⟨ put functions 12 ⟩ +≡ (174)

```
uint8_t hput_kern(kern_t *k)
{ info_t info;
  if (k→x) info = b100; else info = b000;
  if (k→d.h ≡ 0.0 ∧ k→d.v ≡ 0.0) {
    if (k→d.w ≡ 0) HPUT8(zero_dimen_no);
    else { HPUT32(k→d.w); info = info | 2; }
  }
  else { hput_xdimen_node(&(k→d)); info = info | 3; }
  return TAG(kern_kind, info);
}
```

5.4 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.

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

⟨ symbols 2 ⟩ +≡ (175)

```
%token LEADERS "leaders"
%token ALIGN "align"
%token CENTER "center"
%token EXPAND "expand"
%type < info > leaders
%type < info > ltype
```

⟨ scanning rules 3 ⟩ +≡ (176)

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

⟨ parsing rules 5 ⟩ +≡ (177)

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

leaders: *glue_node* *ltype* *rule_node* { $$ = $2; }
  | *glue_node* *ltype* *hbox_node* { $$ = $2; }
  | *glue_node* *ltype* *vbox_node* { $$ = $2; };

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

Writing the long format: ⟹ − − −

⟨ write functions 19 ⟩ +≡ (178)

```
void hwrite_leaders_type(int t)
{ if (t ≡ 2) hwritef(" center");
  else if (t ≡ 3) hwritef(" expand");
}
```

Reading the short format: ··· ⟹

⟨ cases to get content 18 ⟩ +≡ (179)

```
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;
```

⟨ get macros 17 ⟩ +≡ (180)

```
#define HGET_LEADERS(I)
  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( );
```

5.5 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 ⟩ +≡ (181)

typedef struct { **glue_t** *bs*, *ls*; **dimen_t** *lsl*; } **baseline_t**;

Reading the long format: − − − ⟹

⟨ symbols 2 ⟩ +≡ (182)

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

⟨ scanning rules 3 ⟩ +≡ (183)

```
baseline        return BASELINE;
```

⟨ parsing rules 5 ⟩ +≡ (184)

```
baseline: glue_node glue_node dimension
    { $$ = b000;
      if ($1) $$ |= b100;
      if ($2) $$ |= b010;
      if ($3 ≠ 0) { HPUT32($3); $$ |= b001; } };
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 18 ⟩ +≡ (185)

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

⟨ get macros 17 ⟩ +≡ (186)

```
#define HGET_BASELINE(I, B)
  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));
    }
  if ((I) & b001) HGET32((B).lsl); else B.lsl = 0;
  hwrite_dimension(B.lsl);
```

Writing the short format: ⟹ ···

⟨ put functions 12 ⟩ +≡ (187)

```
  uint8_t hput_baseline(baseline_t *b)
  { info_t 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);
  }
```

5.6 Ligatures

Ligatures occur only in horizontal lists. They specify characters that combines 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, an additional byte following the font byte and preceding the end byte gives the total size needed for the character codes. In the long format, we give the font, the character code, and then the replacement characters coded in utf8.

⟨ hint types 1 ⟩ +≡ (188)

```
  typedef struct { uint8_t f; list_t l; } lig_t;
```

Reading the long format: − − − ⟹

⟨ symbols 2 ⟩ +≡ (189)
```
%token LIGATURE "ligature"
%type <u> lig_cc
%type <lg> ligature
```

⟨ scanning rules 3 ⟩ +≡ (190)
```
ligature        return LIGATURE;
```

⟨ parsing rules 5 ⟩ +≡ (191)
```
replace_cc: | replace_cc TXT_CC { hput_utf8($2); };
lig_cc: UNSIGNED { $$ = hpos − hstart; hput_utf8($1); };
lig_cc: CHARCODE { $$ = hpos − hstart; hput_utf8($1); };
ligature: REFERENCE { REF(font_kind, $1); HPUT8($1); } lig_cc TXT_START
        replace_cc 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: ⟹ − − −

⟨ write functions 19 ⟩ +≡ (192)
```
void hwrite_ligature(lig_t *l)
{ uint32_t pos = hpos − hstart;
  hwritef("␣*%d", 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: · · · ⟹

⟨ cases to get content 18 ⟩ +≡ (193)
```
case TAG(ligature_kind, 1): { lig_t l; HGET_LIG(1, l); } break;
case TAG(ligature_kind, 2): { lig_t l; HGET_LIG(2, l); } break;
case TAG(ligature_kind, 3): { lig_t l; HGET_LIG(3, l); } break;
case TAG(ligature_kind, 4): { lig_t l; HGET_LIG(4, l); } break;
case TAG(ligature_kind, 5): { lig_t l; HGET_LIG(5, l); } break;
case TAG(ligature_kind, 6): { lig_t l; HGET_LIG(6, l); } break;
case TAG(ligature_kind, 7): { lig_t l; HGET_LIG(7, l); } break;
```

⟨ get macros 17 ⟩ +≡ (194)
```
#define HGET_LIG(I, L)
```

```
(L).f = HGET8;
if ((I) ≡ 7) (L).l.s = HGET8; else (L).l.s = (I);
(L).l.p = hpos − hstart; hpos += (L).l.s;
if ((I) ≡ 7)
{ uint8_t n = HGET8;
  if (n ≠ (L).l.s)
    QUIT("Sizes␣in␣ligature␣don't␣match␣%d!=%d", (L).l.s, n);
}
hwrite_ligature(&(L));
```

Writing the short format: ⟹ ···

⟨ put functions 12 ⟩ +≡ (195)

```
uint8_t hput_ligature(lig_t *l)
{ if (l→l.s < 7) return TAG(ligature_kind, l→l.s);
  else
  { memmove(hstart + l→l.p + 1, hstart + l→l.p, l→l.s); hpos++;
    *(hstart + l→l.p) = *hpos++ = l→l.s;
    return TAG(ligature_kind, 7);
  }
}
```

5.7 Hyphenation

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 hyphen 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 hyphen, followed by the *pre_break* list, then comes the replace-count followed by the *post_break* list. An empty *pre_break* or *post_break* list may be omitted.

In the short format, the three components of a hyphen node are stored in this order: *pre_break* list, *post_break* list, and *replace_count*. The *b100* bit in the info value indicates the presence of a *pre_break* list, the *b010* bit the presence of a *post_break* list, and the *b001* bit the presence of a replace-count. 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 *b001* 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 hyphen.

⟨ hint types 1 ⟩ +≡ (196)

```
typedef struct hyphen_t { bool x; list_t p, q; uint8_t r; } hyphen_t;
```

Reading the long format: − − − ⟹

⟨ symbols 2 ⟩ +≡ (197)

```
%token HYPHEN "hyphen"
%type <hy> hyphen hyphen_node
%type <l> opt_list
```

⟨ scanning rules 3 ⟩ +≡ (198)

```
hyphen          return HYPHEN;
```

⟨ parsing rules 5 ⟩ +≡ (199)

```
opt_list: { $$.p = hpos − hstart; $$.s = 0; $$.k = list_kind; }
  | list { if ($1.s ≡ 0) hpos = hpos − 2; $$ = $1; };
hyphen: explicit opt_list UNSIGNED opt_list
    { $$.x = $1; $$.p = $2; RNG("Replace␣count", $3, 0, 31); $$.r = $3;
      $$.q = $4; };
hyphen_node: start HYPHEN hyphen END
    { hput_tags($1, hput_hyphen(&($3))); $$ = $3; };
content_node: hyphen_node;
```

Writing the long format: ⟹ − − −

⟨ write functions 19 ⟩ +≡ (200)

```
void hwrite_hyphen(hyphen_t *h)
{ hwrite_explicit(h→x);
  if (h→p.s ≠ 0) hwrite_list(&(h→p));
  hwritef("␣%d", h→r);
  if (h→q.s ≠ 0) hwrite_list(&(h→q));
}
void hwrite_hyphen_node(hyphen_t *h)
{ hwrite_start(); hwritef("hyphen"); hwrite_hyphen(h); hwrite_end();
}
```

Reading the short format: $\cdots \Longrightarrow$

⟨ cases to get content 18 ⟩ +≡ (201)

```
case TAG(hyphen_kind, b001):
  { hyphen_t h; HGET_HYPHEN(b001, h); hwrite_hyphen(&h); } break;
case TAG(hyphen_kind, b010):
  { hyphen_t h; HGET_HYPHEN(b010, h); hwrite_hyphen(&h); } break;
case TAG(hyphen_kind, b011):
  { hyphen_t h; HGET_HYPHEN(b011, h); hwrite_hyphen(&h); } break;
case TAG(hyphen_kind, b100):
  { hyphen_t h; HGET_HYPHEN(b100, h); hwrite_hyphen(&h); } break;
case TAG(hyphen_kind, b101):
  { hyphen_t h; HGET_HYPHEN(b101, h); hwrite_hyphen(&h); } break;
case TAG(hyphen_kind, b110):
  { hyphen_t h; HGET_HYPHEN(b110, h); hwrite_hyphen(&h); } break;
case TAG(hyphen_kind, b111):
  { hyphen_t h; HGET_HYPHEN(b111, h); hwrite_hyphen(&h); } break;
```

⟨ get macros 17 ⟩ +≡ (202)

```
#define HGET_HYPHEN(I, Y)
  if ((I) & b100) hget_list(&((Y).p));
  else { (Y).p.p = hpos − hstart; (Y).p.s = 0; (Y).p.k = list_kind; }
  if ((I) & b010) hget_list(&((Y).q));
  else { (Y).q.p = hpos − hstart; (Y).q.s = 0; (Y).q.k = list_kind; }
  if ((I) & b001) { 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; }
```

⟨ get functions 16 ⟩ +≡ (203)

```
  void hget_hyphen_node(hyphen_t *h)
  { ⟨ read the start byte a 14 ⟩
    if (KIND(a) ≠ hyphen_kind ∨ INFO(a) ≡ b000)
      QUIT("Hyphen␣expected␣at␣0x%x␣got␣%s,%d", node_pos, NAME(a),
          INFO(a));
    HGET_HYPHEN(INFO(a), *h);
    ⟨ read and check the end byte z 15 ⟩
  }
```

Writing the short format: $\Longrightarrow \cdots$

⟨ put functions 12 ⟩ +≡ (204)

```
uint8_t hput_hyphen(hyphen_t *h)
{ info_t info = b000;
  if (h→p.s > 0) info |= b100;
  if (h→q.s > 0) info |= b010;
  if (h→x ∨ h→r ≠ 0 ∨ info ≡ b000)
  { info |= b001; HPUT8(h→r | (h→x ? #80 : #00)); }
  return TAG(hyphen_kind, info);
}
```

5.8 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 xdimen 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 discretionary hyphen),
  `ex_hyphen_penalty` (penalty for break after explicit hyphen),
  `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[5]; 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 is a displayed formula that interrupts a paragraph. Displays are described in the next section.

To summarize, a paragraph node in the long format specifies an extended dimension, an optional node list, and an optional parameter list. The extended dimension is given either as a reference or as an *xdimen* node (info bit *b100*); the same holds for the parameter list (info bit *b010*).

Reading the long format: − − − ⟹

⟨ symbols 2 ⟩ +≡ (205)
```
%token PAR "par"
%type < info > par
```

⟨ scanning rules 3 ⟩ +≡ (206)
```
par                return PAR;
```

⟨ parsing rules 5 ⟩ +≡ (207)
```
par: xdimen_ref param_ref list { $$ = b000; }
   | xdimen_ref param_list_node list { $$ = b010; }
   | xdimen_ref list { $$ = b010; }
   | xdimen_node param_ref list { $$ = b100; }
   | xdimen_node param_list_node list { $$ = b110; }
   | xdimen_node list { $$ = b110; };
content_node: start PAR par END { hput_tags($1, TAG(par_kind, $3)); };
```

Reading the short format: · · · ⟹

⟨ cases to get content 18 ⟩ +≡ (208)
```
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 17 ⟩ +≡ (209)
```
#define HGET_PAR(I)
  if ((I) & b100) { xdimen_t x;
    hget_xdimen_node(&x); hwrite_xdimen_node(&x); }
  else HGET_REF(xdimen_kind);
  if ((I) & b010) { list_t l; hget_param_list_node(&l);
    hwrite_param_list_node(&l); }
```

```
else HGET_REF(param_kind);
{ list_t l; hget_list(&l); hwrite_list(&l); }
```

5.9 Displays

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 sort 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: − − − ⟹

⟨ symbols 2 ⟩ +≡ (210)

```
%token DISPLAY "display"
%type < info > display
```

⟨ scanning rules 3 ⟩ +≡ (211)

```
display           return DISPLAY;
```

⟨ parsing rules 5 ⟩ +≡ (212)

```
display: list { $$ = b100; }
  | list hbox_node { $$ = b101; }
  | hbox_node list { $$ = b110; }
  | param_ref list { $$ = b000; }
  | param_ref list hbox_node { $$ = b001; }
  | param_ref hbox_node list { $$ = b010; }
  | param_list_node list { $$ = b100; }
  | param_list_node list hbox_node { $$ = b101; }
  | param_list_node hbox_node list { $$ = b110; };
content_node: start DISPLAY display END
    { hput_tags($1, TAG(display_kind, $3)); };
```

Reading the short format: $\cdots \Longrightarrow$

⟨ cases to get content 18 ⟩ +≡ (213)

```
case TAG(display_kind, b000): HGET_DISPLAY(b000); break;
case TAG(display_kind, b001): HGET_DISPLAY(b001); break;
case TAG(display_kind, b010): HGET_DISPLAY(b010); break;
case TAG(display_kind, b100): HGET_DISPLAY(b100); break;
case TAG(display_kind, b101): HGET_DISPLAY(b101); break;
case TAG(display_kind, b110): HGET_DISPLAY(b110); break;
```

⟨ get macros 17 ⟩ +≡ (214)

```
#define HGET_DISPLAY(I)
  if ((I) & b100) { list_t l; hget_param_list_node(&l);
    hwrite_param_list_node(&l); }
  else HGET_REF(param_kind);
  if ((I) & b010) hget_hbox_node();
  { list_t l; hget_list(&l); hwrite_list(&l); }
  if ((I) & b001) hget_hbox_node();
```

5.10 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 are implemented as just another type of list node.

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

⟨ symbols 2 ⟩ +≡ (215)

```
%token ADJUST "adjust"
%type <l> adjustment
```

⟨ scanning rules 3 ⟩ +≡ (216)

```
adjust          return ADJUST;
```

⟨ parsing rules 5 ⟩ +≡ (217)

```
  adjustment: estimate content_list { $$.k = adjust_kind; $$.p = $2;
        $$.s = (hpos − hstart) − $2; };

  content_node: start ADJUST adjustment END {
        hput_tags($1, hput_list($1 + 1, &($3))); };
```

Reading the short format: ··· ⟹

⟨ cases to get content 18 ⟩ +≡ (218)

```
case TAG(adjust_kind, 1):
  { list_t l; HGET_LIST(1, l); l.k = adjust_kind; hwrite_adjustments(&l); } break;
case TAG(adjust_kind, 2):
  { list_t l; HGET_LIST(2, l); l.k = adjust_kind; hwrite_adjustments(&l); } break;
case TAG(adjust_kind, 3):
  { list_t l; HGET_LIST(3, l); l.k = adjust_kind; hwrite_adjustments(&l); } break;
case TAG(adjust_kind, 4):
  { list_t l; HGET_LIST(4, l); l.k = adjust_kind; hwrite_adjustments(&l); } break;
case TAG(adjust_kind, 5):
  { list_t l; HGET_LIST(5, l); l.k = adjust_kind; hwrite_adjustments(&l); } break;
```

I guess the following should be incorporated into *hwrite_list*.

Writing the long format: ⟹ − − −

⟨ write functions 19 ⟩ +≡ (219)

```
void hwrite_adjustments(list_t *l)
{
  if (l→s ≡ 0) return;
  else { 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_content_node();
    hpos = hstart + h; hend = hstart + e;     /* restore hpos and hend */
  }
}
```

5.11 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 in the viewer.

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 consists of nonaligned content, for example extra glue or rules, and aligned content called items. The table's content is stacked, either vertical or horizontal, orthogonal to the alignment direction of the table. The aligned content of a table is packed in an outer item node, that contains a list of inner item nodes. 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 the list of inner table items. This is just for convenience; the first list must contain glue nodes and the second list must contain 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 *b001* bit is used to mark a vertical table and the *b110* 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 $_2$ ⟩ +≡ (221)

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

⟨ scanning rules $_3$ ⟩ +≡ (222)

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

⟨ parsing rules $_5$ ⟩ +≡ (223)

```
content_node: start ITEM content_node END { hput_tags($1, hput_item(1)); };
```

content_node: *start* ITEM *content_node* UNSIGNED END {
    hput_tags($1, *hput_item*($4)); };

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 | *b001*; } *content_node*: *start* TABLE *table* END {
    hput_tags($1, TAG(*table_kind*, $3)); };

Reading the short format: · · · ⟹

⟨ cases to get content 18 ⟩ +≡ (224)

```
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_t l; hget_list(&l); hwrite_list(&l); } break;
case TAG(item_kind, b001): hget_content_node(); break;
case TAG(item_kind, b010): hget_content_node(); hwritef(" 2"); break;
case TAG(item_kind, b011): hget_content_node(); hwritef(" 3"); break;
case TAG(item_kind, b100): hget_content_node(); hwritef(" 4"); break;
case TAG(item_kind, b101): hget_content_node(); hwritef(" 5"); break;
case TAG(item_kind, b110): hget_content_node(); hwritef(" 6"); break;
case TAG(item_kind, b111): hget_content_node(); hwritef(" %u", HGET8); break;
```

⟨ get macros 17 ⟩ +≡ (225)

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

Writing the short format: ⟹ ⋯

⟨ put functions 12 ⟩ +≡ (226)

```
uint8_t hput_item(uint32_t n)
{
  if (n ≡ 0) QUIT("Span count in item must not be zero");
  else if (n < 7) return TAG(item_kind, n);
  else if (n > #FF) QUIT("Span count %d must be less than 255", n);
  else { HPUT8(n);
    return TAG(item_kind, 7);
  }
}
```

6 Extensions to TEX

6.1 Images

Images behave pretty much like glue. They can stretch (or shrink) together with the surrounding glue to fill a horizontal or vertical box. Like glue, they stretch in the horizontal direction when filling an horizontal box and they stretch in the vertical direction as part of a vertical box. Stretchability and shrinkability are optional parts of an image node.

Unlike glue, images have both a width and a height. The relation of height to width, the aspect ratio, is preserved by stretching and shrinking.

While glue often has a zero width, images usually have a nonzero natural size and making them much smaller is undesirable. The natural width and height of an image are optional parts of an image node; typically however, this information is contained in the image data.

The only required part of an image node is the number of the auxiliary section where the image data can be found.

⟨ hint types 1 ⟩ +≡ (227)

```
typedef struct { uint16_t n; dimen_t w, h; stretch_t p, m; } image_t;
```

Reading the long format: − − − ⟹

⟨ symbols 2 ⟩ +≡ (228)

```
%token IMAGE "image"
%type < x > image image_dimen
```

⟨ scanning rules 3 ⟩ +≡ (229)

```
image                return IMAGE;
```

⟨ parsing rules 5 ⟩ +≡ (230)

```
image_dimen: dimension dimension { $$.w = $1; $$.h = $2; }
   | { $$.w = $$.h = 0; };
image: UNSIGNED image_dimen plus minus { $$.w = $2.w; $$.h = $2.h;
      $$.p = $3; $$.m = $4; RNG("Section␣number", $1, 3, max_section_no);
      $$.n = $1; };
content_node: start IMAGE image END { hput_tags($1, hput_image(&($3))); }
```

Writing the long format: ⟹ − − −

⟨ write functions 19 ⟩ +≡ (231)

```
void hwrite_image(image_t *x)
{ hwritef("␣%u", x→n);
  if (x→w ≠ 0 ∨ x→h ≠ 0) { hwrite_dimension(x→w);
    hwrite_dimension(x→h);
  }
  hwrite_plus(&x→p); hwrite_minus(&x→m);
}
```

Reading the short format: · · · ⟹

⟨ cases to get content 18 ⟩ +≡ (232)

```
case TAG(image_kind, b100): { image_t x; HGET_IMAGE(b100, x); } break;
case TAG(image_kind, b101): { image_t x; HGET_IMAGE(b101, x); } break;
case TAG(image_kind, b110): { image_t x; HGET_IMAGE(b110, x); } break;
case TAG(image_kind, b111): { image_t x; HGET_IMAGE(b111, x); } break;
```

⟨ get macros 17 ⟩ +≡ (233)

```
#define HGET_IMAGE(I, X)
  HGET16 ((X).n); RNG("Section␣number", (X).n, 3, max_section_no);
  if (I & b010) { HGET32((X).w); HGET32((X).h); }
  else (X).w = (X).h = 0;
  if (I & b001) { HGET_STRETCH((X).p); HGET_STRETCH((X).m); }
  else { (X).p.f = (X).m.f = 0.0; (X).p.o = (X).m.o = normal_o; }
  hwrite_image(&(X));
```

Writing the short format: ⟹ · · ·

⟨ put functions 12 ⟩ +≡ (234)

```
uint8_t hput_image(image_t *x)
{ info_t i = b100;

  HPUT16(x→n);
  if (x→w ≠ 0 ∨ x→h ≠ 0) { HPUT32(x→w); HPUT32(x→h); i |= b010;
  }
  if (x→p.f ≠ 0.0 ∨ x→m.f ≠ 0.0) { hput_stretch(&x→p);
    hput_stretch(&x→m); i |= b001;
  }
  return TAG(image_kind, i);
}
```

6.2 Colors

Colors are certainly one of the features you will find in the final HINT file format. Here some remarks must suffice.

A HINT viewer must be capable of rendering a page given just any valid position inside the content section. Therefore HINT files are stateless; there is no need to search for preceding commands that might change a state variable. As a consequence, we can not just define a "color change node". Colors could be specified as an optional parameter of a glyph node, but the amount of data necessary would be considerable. In texts, on the other hand, a color change control code would be possible because we parse texts only in forward direction. The current font would then become a current color and font with the appropriate changes for positions.

A more attractive alternative would be to specify colored fonts. This would require an optional color argument for a font. For example one could have a cmr10 font in black as font number 3, and a cmr10 font in blue as font number 4. Having 256 different fonts, this is definitely a possibility because rarely you would need that many fonts or that many colors. If necessary and desired, one could allow 16 bit font numbers of overcome the problem.

Background colors could be associated with boxes as an optional parameter.

6.3 Positions, 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. To render a page starting at that 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. Lets 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 give position.
- A page that ends exactly at the give 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 target position. Let’s call this the target 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 target line on the page, otherwise display an equal amount of lines before and after the target 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 target 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 target 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 target line.
- Once the page start (or end) is found, use TEX's page builder (or its reverse variant) to complete the page.

We call nodes that reference a position inside the content section a link node. As with other nodes, we can use predefined links. The first 256 of them can be referenced by a single byte. We should reserve reference number 0 for a link to the beginning of the content and reference number 1 for a link to the end of the content. Probably having only 256 links would be a severe restriction, hence we will allow also 16 bit reference numbers. If still more links are needed, links can be embedded directly in the content stream. We need two types of links, a start link and an end link such that the content between the two will constitute the visible part of the link.

In the short format, we will use the *b100* bit of the info value to distinguish them: 1 indicates start link, 0 indicates end link. The two low bits of the info value will be 0 for an 8 bit reference number, 1 for a 16 bit reference number, 2 for an immediate link without secondary position and current font, and 3 for an immediate link with 32 bit secondary position and current font. The link itself consists of a primary position, an optional secondary position, an optional current font, and a position type. The position type is 0 for the exact page top, 1 for the exact page bottom, and 2 for the approximate middle as described above.

In the long format, a position can not be expressed as a byte position; instead we use labels. A label is identified by a unique name expressed as a string. For example we can write `<label 'label10'>` and then we can use `'label10'` as a symbolic reference to the position of the node that follows the label node. When translating the long format to the short format, these label nodes will disappear. To keep readable label names, the links in the short format may specify an optional name that is used for labels. If no name is given, a label name is generated. When translating the short format to the long format, we test just before writing a new node whether there is a link to this node and insert a label if so. Because we write nodes in ascending order of positions, we can sort the labels in ascending order of position and compare *hpos* with the position of the next label in this order. Immediate back links pose a problem for this translation because the node has already been written without a label when we encounter the link node that refers to it. If we encounter such a link we must resort to a two pass translation: We log the information about the back link and continue with the translation. After the whole file is translated, we check the log, and if unresolved back links where found, we sort them into the previously incomplete list of links and repeat the translation.

When translating the long format to the short format, immediate forward links pose a similar problem: We can not encode the links position because we have not yet encountered the label. In case we have unused reference numbers for predefined links, we will convert the immediate link into a predefined link. Predefined links can be completed with positions when we find the labels, all we need to know to encode the link itself is the reference number. If all 16 bit numbers are already in use, we reserve the maximum amount of memory (8 bit for the type information, 32 bit for the primary position, 32 bit for the secondary position, and 8 bit for the font number) in the stream and keep a linked list of positions for the given label (the reserved space in the link nodes can be used of that purpose) and fill in the information once we find the corresponding label.

Links and Labels are not yet implemented.

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. Long footnotes may even be split and displayed at the bottom of the next 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 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 method.

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

* Like this one.

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 and uses these copies. These parameters are the *page_goal* and the *page_max_depth*; further the parameters *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, inserts specify an insert class n and go into box$n$. Material that goes into box255 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 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. There is no output routine 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. There are 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

Stream definitions occur only in page templates. They start with stream number and stream type. 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. For normal streams, a stream definition specifies the maximum insertion height, the magnification factor, and optional information about splitting the stream. In the short format, the info bit *b100* indicates the presence of splitting information. 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 material that goes before a nonempty stream, an extended dimension node, specifying the width of the inserted material, the top skip glue, and the material that goes after a nonempty stream.

Reading the long format: − − − ⟹

⟨symbols 2⟩ +≡ (235)

```
%token FIRST "first"
%token LAST "last"
%token TOP "top"
%token NOREFERENCE "*"
%type < info > stream_type stream_split stream_def
```

⟨scanning rules 3⟩ +≡ (236)

```
first     return FIRST;
last      return LAST;
top       return TOP;
\*        return NOREFERENCE;
```

⟨ parsing rules 5 ⟩ +≡ (237)

```
stream_type: FIRST { $$ = 1; } | LAST { $$ = 2; } | TOP { $$ = 3; }
  | xdimen_node integer { HPUT16($2); } stream_split { $$ = $4 + 0; };
stream_link: stream_ref | NOREFERENCE { HPUT8(255); };
stream_split: { $$ = b000; }
  | stream_link stream_link UNSIGNED
    { RNG("split␣ratio", $3, 0, 1000); HPUT16($3); $$ = b100; };
stream_def: start STREAM stream_ref stream_type
    list xdimen_node glue_node list END
    { hput_tags($1, TAG(stream_kind, $4)); };
```

Reading the short format: ··· ⟹

⟨ get functions 16 ⟩ +≡ (238)

```
static void hget_stream_split(void)
{ uint8_t n;
  int r;
  n = HGET8;
  if (n ≡ 255) hwritef("␣*");
  else { REF(stream_kind, n); hwritef("␣*%d", n); }
  n = HGET8;
  if (n ≡ 255) hwritef("␣*");
  else { REF(stream_kind, n); hwritef("␣*%d", n); }
  HGET16(r);
  RNG("split␣ratio", r, 0, 1000);
  hwritef("␣%d", r);
}
void hget_stream_def(void)
{
  if (KIND(*hpos) ≠ stream_kind) return;
  else { info_t i;
    list_t l;
    xdimen_t x;
    ⟨ read the start byte a 14 ⟩
    i = INFO(a);
    hwrite_start(); hwritef("stream");
    HGET_REF(stream_kind);
    if ((i & b011) ≡ 0) { uint16_t f;
      hget_xdimen_node(&x); hwrite_xdimen_node(&x);
      HGET16(f); RNG("magnification␣factor", f, 0, 1000);
      hwritef("␣%d", f);
      if (i & b100) hget_stream_split();
    }
    else if ((i & b011) ≡ 1) hwritef("␣first");
```

```
        else if ((i & b011) ≡ 2) hwritef("␣last");
        else if ((i & b011) ≡ 3) 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);
        hwrite_end();
        ⟨ read and check the end byte z 15 ⟩
      }
    }
```

7.2 Stream Nodes

Stream nodes occur in the content section where they must not be inside other nodes except toplevel paragraph nodes. (As we will see in section 7.3, they occur with a different meaning also in page templates.) A complete stream node contains in this order: the stream reference number, the *height* (*b100*), the *floating_penalty* (*b010*), the *split_max_depth* followed by the *split_top_skip* (*b010*), and the content list. All these components, except the stream reference number and the content list, are optional. In the short format, the presence of the optional parameters is indicated by the info bits shown above in parentheses. If the height is missing, it can be computed from the content list, and the global values of *floating_penalty*, *split_max_depth*, and *split_top_skip* serve as substitutes for other missing parameters.

Writing the short format: ⟹ · · ·

Reading the long format: − − − ⟹

⟨ symbols 2 ⟩ +≡ (239)

```
%token STREAM "stream"
%type < info > float_info stream_height stream
```

⟨ scanning rules 3 ⟩ +≡ (240)

```
stream            return STREAM;
```

⟨ parsing rules 5 ⟩ +≡ (241)

```
  float_info: penalty dimension { HPUT16($1); HPUT32($2); }
      glue_node { $$ = b011; }
    | penalty { HPUT16($1); $$ = b010; }
    | dimension { HPUT32($1); } glue_node { $$ = b001; }
    | { $$ = b000; };
  stream_height: { $$ = b000; } | TO dimension { HPUT32($2); $$ = b100; };
  stream: stream_ref stream_height float_info list { $$ = $2 | $3; };
  content_node: start STREAM stream END
      { hput_tags($1, TAG(stream_kind, $3)); };
```

Writing the long format: ⟹ − − −

Reading the short format: · · · ⟹

```
⟨ cases to get content 18 ⟩ +≡                                          (242)
case TAG(stream_kind, b000): HGET_STREAM(b000); break;
case TAG(stream_kind, b001): HGET_STREAM(b001); break;
case TAG(stream_kind, b010): HGET_STREAM(b010); break;
case TAG(stream_kind, b011): HGET_STREAM(b011); break;
case TAG(stream_kind, b100): HGET_STREAM(b100); break;
case TAG(stream_kind, b101): HGET_STREAM(b101); break;
case TAG(stream_kind, b110): HGET_STREAM(b110); break;
case TAG(stream_kind, b111): HGET_STREAM(b111); break;
```

```
⟨ get macros 17 ⟩ +≡                                                    (243)
#define HGET_STREAM(I)
  HGET_REF (stream_kind);
  if ((I) & b100) { scaled_t h; HGET32(h); hwritef("␣to");
    hwrite_dimension(h); }
  if ((I) & b010)
  { int16_t p; HGET16(p); RNG("Penalty", p, −10000, +10000); hwrite_signed(p); }
  if ((I) & b001)
  { scaled_t d; HGET32(d); hwrite_dimension(d); hget_glue_node(); }
  { list_t l; hget_list(&l); hwrite_list(&l); }
```

7.3 Page Templates

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 templates 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 . . .

The main part of a page template is a list of vertical material. To construct the page, this list will just be placed into a vertical box, but of course before doing so, the viewer will scan the list, replace all stream nodes found inside by the appropriate content streams, and set the glue.

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`.

At the end of the page template, a list of stream definitions provides information about the streams that are mentioned in the page's list of vertical material. In addition it might define streams that merely distribute content to other streams or streams that just accumulate material to be used on later pages.

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: − − − ⟹

⟨ symbols 2 ⟩ +≡ (244)

```
%token PAGE "page"
```

⟨ scanning rules 3 ⟩ +≡ (245)

```
page        return PAGE;
```

⟨ parsing rules 5 ⟩ +≡ (246)

```
page_priority: { HPUT8(1); }
    | UNSIGNED { RNG("page␣priority", $1, 0, 255); HPUT8($1); };
page_streams: | page_streams stream_def;
page: string { hput_string($1); } page_priority
      VBOX xdimen_node HBOX xdimen_node list page_streams;
```

Reading the short format: · · · ⟹

⟨ get functions 16 ⟩ +≡ (247)

```
void hget_page(void)
{ char *n;
  uint8_t p;
  list_t l;
  xdimen_t x;
  HGET_STRING(n); hwrite_string(n);
  p = HGET8; if (p ≠ 1) hwritef("␣%d", p);
  hwritef("␣vbox␣"); { hget_xdimen_node(&x); hwrite_xdimen_node(&x); }
  hwritef("␣hbox␣"); { hget_xdimen_node(&x); hwrite_xdimen_node(&x); }
  hget_list(&l); hwrite_list(&l);
  while (KIND(*hpos) ≡ stream_kind) hget_stream_def();
}
```

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. 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 #`FFFFFFFF` 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 ⟩ +≡ (248)

typedef struct { **uint8_t** *pg*; **uint32_t** *pos*; **bool** *on*; **int** *link*;
} **range_pos_t**;

* For a HINT viewer, a data structure which allows fast retrieval of all valid page templates for a given position is needed.

⟨ common variables 249 ⟩ ≡ (249)

```
range_pos_t *range_pos;
int next_range = 1, max_range;
int *page_on;
```

Used in 434, 435, and 437.

⟨ allocate data 250 ⟩ ≡ (250)

```
ALLOCATE(page_on, max_ref[page_kind] + 1, int);
ALLOCATE(range_pos, 2 * (max_ref[range_kind] + 1), range_pos_t);
```

Used in 300 and 305.

⟨ hint macros 11 ⟩ +≡ (251)

```
#define ALLOCATE(R, S, T)
  ( (R) = calloc((S), sizeof (T)),
      (((R) ≡ NULL) ? QUIT("Out␣of␣memory␣for␣" #R) : 0) )
#define REALLOCATE(R, S, T)
  ( (R) = realloc((R), (S) * sizeof (T)),
      (((R) ≡ NULL) ? QUIT("Out␣of␣memory␣for␣" #R) : 0) )
```

Reading the long format: − − − ⟹

⟨ symbols 2 ⟩ +≡ (252)

```
%token RANGE "range"
```

⟨ scanning rules 3 ⟩ +≡ (253)

```
range              return RANGE;
```

⟨ parsing rules 5 ⟩ +≡ (254)

```
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 19 ⟩ +≡ (255)

```
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 16 ⟩ +≡ (256)

```
void hget_range(info_t 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 = #FFFFFFFF;
  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 ≠ #FFFFFFFF)
  { range_pos[next_range].pg = pg;
    range_pos[next_range].on = false;
    range_pos[next_range].pos = to;
    next_range++;
  }
}
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)
    { range_pos_t 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 12 ⟩ +≡                                                    (257)
  void hput_range(uint8_t pg, bool on)
  { REF(range_kind, (next_range − 1)/2);
    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++;
  }
  extern void hput_definitions_end(void);
  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_t 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 = #FFFFFFFF;
        if (from ≠ 0)
        { info = info | b100; if (from > #FFFF) info = info | b001; }
        if (to ≠ #FFFFFFFF)
        { 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 11 ⟩ +≡                                                    (258)
#define MAX_BANNER 256
```

To check the banner, we have the function *hcheck_banner*; it returns *true* if successful.

```
⟨ function to check the banner 259 ⟩ ≡                                   (259)
  int version = 1, subversion = 0;
  char hbanner[MAX_BANNER + 1];

  bool hcheck_banner(char *magic)
  { int hbanner_size = 0;
    char *t;

    t = hbanner;
    if (strncmp(magic, hbanner, 4) ≠ 0)
      QUIT("This␣is␣not␣a␣%s␣file", magic);
    else t += 4;
    hbanner_size = (int) strnlen(hbanner, MAX_BANNER);
    if (hbanner[hbanner_size − 1] ≠ '\n')
      QUIT("Banner␣exceeds␣maximum␣size=0x%x", MAX_BANNER);
```

```
    if (*t ≠ '␣') QUIT("Space␣expected␣after␣%s", magic);
    else t++;
    version = strtol(t, &t, 10);
    if (*t ≠ '.') QUIT("Dot␣expected␣after␣version␣number␣%d", version);
    else t++;
    subversion = strtol(t, &t, 10);
    if (*t ≠ '␣' ∧ *t ≠ '\n')
      QUIT("Space␣expected␣after␣subversion␣number␣%d", subversion);
    MESSAGE("%s␣file␣version␣%d.%d:%s", magic, version, subversion, t);
    DBG(DBGDIR, "banner␣size=0x%x\n", hbanner_size);
    return true;
  }
```

Used in 429, 434, 435, and 437.

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.

Reading the short format: · · · ⟹

⟨ get file functions 260 ⟩ ≡ (260)

```
  void hget_banner(void)
  { int i;
    for (i = 0; i < MAX_BANNER; i++) { hbanner[i] = HGET8;
      if (hbanner[i] ≡ '\n') break;
    }
    hbanner[++i] = 0;
  }
```

Used in 429, 435, and 437.

To read a long format file, we use the function *fgetc*.

Reading the long format: − − − ⟹

⟨ read the banner 261 ⟩ ≡ (261)

```
  { int i = 0, c;
    do { c = fgetc(hin);
      if (c ≠ EOF) hbanner[i++] = (char) c;
      else break;
    } while (c ≠ '\n' ∧ i < MAX_BANNER);
    hbanner[i] = 0;
  }
```

Used in 434.

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: ⟹ ···

⟨ function to write the banner 262 ⟩ ≡ (262)

```
extern int version, subversion;

static size_t hput_banner(char *magic, char *s)
{ return fprintf(hout, "%s␣%d.%d␣%s\n", magic, version, subversion, s);
}
```

Used in 431, 434, and 435.

Writing the long format: ⟹ − − −

Writing the banner of a long format file is essentialy the same as for 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 ⟩ +≡ (263)

```
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 12 ⟩ +≡ (264)

```
static size_t hput_root(void);
static size_t hput_section(uint16_t n);
static void hput_optional_sections(void);

void 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);
```

```
  hput_optional_sections( );
}
```

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.

⟨ common variables 249 ⟩ +≡ (265)

```
uint8_t *hpos = NULL, *hstart = NULL, *hend = 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: ··· ⟹

⟨ get file macros 35 ⟩ +≡ (266)

```
#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 12 ⟩ +≡ (267)

```
void hput_error(void){
    if (hpos < hend) return;
    QUIT("HPUT␣overrun␣section␣%d␣pos="SIZE_F"\n" ,
        section_no, hpos − hstart ) ;
    }
```

⟨ put macros 268 ⟩ ≡ (268)

```
  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 430 and 434.

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 268 ⟩ +≡ (269)

```
  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.

⟨ hint macros 11 ⟩ +≡ (270)

```
#define MAX_TAG_DISTANCE 32
                    /* This is a guess; I need a tight upper bound. */
```

8.4 Mapping a Short Format File

Since modern computers with 64bit hardware have a huge address space, mapping the entire file into virtual memory is the most efficient way to read 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 adresses.

The following functions map and unmap a short format input file at address *hbase*.

⟨ map functions 271 ⟩ ≡ (271)

```
  ⟨ mmap and munmap declarations 272 ⟩
  static size_t hbase_size;
  uint8_t *hbase = NULL;
  extern char *in_name;
  void hget_map(void)
  { struct stat st;
    int fd;
    fd = open(in_name, O_RDONLY, 0);
    if (fd < 0) QUIT("Unable␣to␣open␣file␣%s", in_name);
    if (fstat(fd, &st) < 0) QUIT("Unable␣to␣get␣file␣size");
    hbase_size = st.st_size;
    hbase = mmap(NULL, hbase_size, PROT_READ, MAP_PRIVATE, fd, 0);
    if (hbase ≡ MAP_FAILED) { hbase = NULL;
      hbase_size = 0;
      QUIT("Unable␣to␣map␣file␣into␣memory");
    }
    close(fd);
    hpos = hstart = hbase;
    hend = hstart + hbase_size;
  }
  void hget_unmap(void)
  { munmap(hbase, hbase_size);
    hbase = NULL;
```

```
    hbase_size = 0;
    hpos = hstart = hend = NULL;
  }
```
Used in 429, 435, and 437.

A small complication arrises from the fact that the *mmap* and *munmap* functions and the associated header files are not available under the Windows operating system and not even under MinGW.

So we need to implement our own version of these functions. We do not implement general purpose replacements but only a replacement for the calls with the parameters used above. We start with the function *_get_osfhandle* to obtain a Windows HANDLE for the given file descriptor, then use *GetFileSize*, *CreateFileMapping*, and finally *MapViewOfFile*. The file is closed with *CloseHandle*.

⟨ *mmap* and *munmap* declarations 272 ⟩ ≡ (272)

```
#ifdef WIN32
#include <windows.h>
#include <io.h>
#define PROT_READ #1
#define MAP_PRIVATE #02
#define MAP_FAILED ((void *) −1)
  static HANDLE hMap;

  void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)
  { HANDLE hFile = (HANDLE) _get_osfhandle(fd);
    if (hFile ≡ INVALID_HANDLE_VALUE) QUIT("Unable␣to␣get␣file␣handle");
    hMap = CreateFileMapping(hFile, NULL, PAGE_READONLY, 0, 0, NULL);
    if (hMap ≡ NULL) QUIT("Unable␣to␣map␣file␣into␣memory");
    addr = MapViewOfFile(hMap, FILE_MAP_READ, 0, 0, 0);
    if (addr ≡ NULL) QUIT("Unable␣to␣obtain␣address␣of␣file␣mapping");
    CloseHandle(hFile);
    return addr;
  }

  int munmap(void *addr, size_t length)
  { UnmapViewOfFile(addr);
    CloseHandle(hMap);
    hMap = NULL;
    return 0;
  }
#else
#include <sys/mman.h>
#endif
```
Used in 271.

After mapping the file at address *hbase*, access to sections of the file is provided by setting the three pointers *hpos*, *hstart*, and *hend*. The value *hbase* ≡ NULL indicates, that no file is open.

To read sections of a short format input file, we use the function *hget_section*.

Reading the short format: $\cdots \Longrightarrow$

⟨ get file functions 260 ⟩ +≡ (273)

```
⟨ hdecompress function 275 ⟩
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) { hpos = hstart = dir[n].buffer;
    hend = hstart + dir[n].xsize;
  }
  else { hpos = hstart = hbase + dir[n].pos;
    hend = hstart + dir[n].size;
    if (dir[n].xsize > 0) hdecompress(n);
  }
}
```

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 12 ⟩ +≡ (274)

```
#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;
    const double buffer_factor = 1.4142136;                    /* √2 */
    pos = hpos − hstart;
    bsize = dir[section_no].bsize ∗ buffer_factor + 0.5;
    if (bsize < pos + n) bsize = pos + n;
    if (bsize ≥ #FFFFFFFF) bsize = #FFFFFFFF;
    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;
  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);
}
```

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.

⟨ *hdecompress* function 275 ⟩ ≡ (275)

```
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␣missmatch␣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;
    hpos = hstart = buffer;
    hend = hstart + dir[n].xsize;
  }
```

Used in 273.

⟨ *hcompress* function 276 ⟩ ≡ (276)

```
  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;
```

$dir[n].size = z.total\_out;$
}

Used in 294.

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: − − − ⟹

⟨ symbols $_2$ ⟩ +≡ (277)

```
%token DIRECTORY "directory"
%token SECTION "entry"
```

⟨ scanning rules $_3$ ⟩ +≡ (278)

```
directory     return DIRECTORY;
section       return SECTION;
```

⟨ parsing rules $_5$ ⟩ +≡ (279)

```
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 280 ⟩ ≡ (280)

```
typedef struct {
  uint64_t pos;
  uint32_t size, xsize;
  uint16_t section_no;
  char *file_name;
  uint8_t *buffer;
  uint32_t bsize;
} entry_t;
```

Used in 428, 430, 434, 435, and 437.

The function *new_directory* allocates the directory.

⟨ directory functions 281 ⟩ ≡ (281)

```
entry_t *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_t);
  dir[0].section_no = 0; dir[1].section_no = 1; dir[2].section_no = 2;
}
```

Used in 429, 431, 434, 435, and 437.

The function *hset_entry* fills in the appropriate entry.

⟨ directory functions 281 ⟩ +≡ (282)

```
void hset_entry(entry_t *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 `-f` and the `-g` option.

⟨ without `-f` skip writing an existing file 283 ⟩ ≡ (283)

```
if (¬option_force ∧ access(file_name, F_OK) ≡ 0) {
  MESSAGE("File␣'%s'␣exists.\n"
      "To␣rewrite␣the␣file␣use␣the␣-f␣option.\n", file_name);
  continue;
}
```

Used in 288.

The above code uses the *access* function, and we need to make sure it is defined:

⟨ make sure *access* is defined 284 ⟩ ≡ (284)

```
#ifdef WIN32
#include <io.h>
#define access(N, M) _access(N, M)
#define F_OK 0
#else
#include <unistd.h>
#endif
```

Used in 288.

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 behaviour. 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.

⟨ without `-g` compute a local *file_name* 285 ⟩ ≡ (285)

```
if (¬option_global) { int path_length = (int) strlen(file_name);
   ⟨ determine whether file_name is absolute or relative 286 ⟩
   if (file_name_length < stem_length + ext_length + path_length) {
      file_name_length = stem_length + ext_length + path_length;
      REALLOCATE(stem_name, file_name_length + 1, char);
   }
   strcpy(stem_name + stem_length, aux_ext[name_type]);
   strcpy(stem_name + stem_length + ext_length, file_name);
   DBG(DBGDIR, "Replacing␣auxiliar␣file␣name:\n\t%s\n->\t%s\n",
        file_name, stem_name);
   file_name = stem_name;
}
```

Used in 288 and 295.

⟨ determine whether *file_name* is absolute or relative 286 ⟩ ≡ (286)

```
enum {
   absolute = 0, relative = 1
} name_type;
char *aux_ext[2] = {".abs/", ".rel/"};
int ext_length = 5;

if (file_name[0] ≡ '/') { name_type = absolute;
   file_name++;
   path_length−−;
}
else if (path_length > 3 ∧ isalpha(file_name[0]) ∧ file_name[1] ≡
        ':' ∧ file_name[2] ≡ '/') { name_type = absolute;
```

```
    file_name[1] = '_';
  }
  else name_type = relative;                                    Used in 285.
```

It remains to create the direcories along the path we might have constructed.

```
⟨ make sure the path in file_name exists 287 ⟩ ≡                      (287)
  { char *path_end;
    path_end = file_name + 1;
    while (*path_end ≠ 0) {
      if (*path_end ≡ '/') { struct stat s;
        *path_end = 0;
        if (stat(file_name, &s) ≡ −1) {
#ifdef WIN32
          if (mkdir(file_name) ≠ 0)
#else
            if (mkdir(file_name, °777) ≠ 0)
#endif
              QUIT("Unable␣to␣create␣directory␣%s", file_name);
          DBG(DBGDIR, "Creating␣directory␣%s\n", file_name);
        }
        else if (¬S_ISDIR(s.st_mode))
          QUIT("Unable␣to␣create␣directory␣%s,␣file␣exists", file_name);
        *path_end = '/';
      }
      path_end++;
    }
  }                                                   Used in 288 and 366.
```

Writing the long format: ⟹ − − −

```
⟨ write functions 19 ⟩ +≡                                              (288)
  ⟨ make sure access is defined 284 ⟩
  extern char *stem_name;
  extern int stem_length;
  void hwrite_aux_files(void)
  { int i;
    DBG(DBGDIR, "Writing␣%d␣aux␣files\n", max_section_no − 2);
    for (i = 3; i ≤ max_section_no; i++) { FILE *f;
      char *file_name = dir[i].file_name;
      int file_name_length = 0;
      ⟨ without -g compute a local file_name 285 ⟩
      ⟨ without -f skip writing an existing file 283 ⟩
      ⟨ make sure the path in file_name exists 287 ⟩
      f = fopen(file_name, "wb");
```

```
      if (f ≡ NULL)
        QUIT("Unable␣to␣open␣file␣'%s'␣for␣writing", file_name);
      else { size_t s;
        hget_section(i);
        DBG(DBGDIR, "Writing␣file␣%s\n", file_name);
        s = fwrite(hstart, 1, dir[i].size, f);
        if (s ≠ dir[i].size) QUIT("writing␣file␣%s", file_name);
        fclose(f);
      }
    }
  }
```

We write the directory, and the directory entries in long format using the following functions.

⟨ write functions 19 ⟩ +≡ (289)

```
  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: · · · ⟹

⟨ get file macros 35 ⟩ +≡ (290)

```
#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 260 ⟩ +≡ (291)

```
  void hget_entry(entry_t *e)
  { ⟨ read the start byte a 14 ⟩
    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 15 ⟩
    DBG(DBGDIR, "entry␣%d:␣size=0x%x␣xsize=0x%x\n",
        e→section_no, e→size, e→xsize);
  }
```

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 position and size of 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 its position describes 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 260 ⟩ +≡                                              (292)
  static void hget_root(entry_t *root)
  { DBG(DBGDIR, "Get␣Root\n");
    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_t root = {0};
    hget_root(&root);
    DBG(DBGDIR, "Get␣Directory\n");
    new_directory(max_section_no + 1);
    dir[0] = root;
    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; }
  }
  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;
  }
```

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.

⟨ set the file sizes for optional sections 293 ⟩ ≡ (293)

```
{ int i;
  for (i = 3; i ≤ max_section_no; i++) { struct stat s;
    if (stat(dir[i].file_name, &s) ≠ 0)
      QUIT("Unable␣to␣obtain␣file␣size␣for␣'%s'", dir[i].file_name);
    dir[i].size = s.st_size;
    dir[i].xsize = 0;
  }
}
```

Used in 294.

The computation of the sizes of the mandatory sections will be explained later. Armed with these preparations, we can put the directory into the HINT file.

Writing the short format: ⟹ ···

⟨ put functions 12 ⟩ +≡ (294)

```
static void hput_entry(entry_t *e)
{ uint8_t b;
  if (e→size < #100 ∧ e→xsize < #100) b = 0;
  else if (e→size < #10000 ∧ e→xsize < #10000) b = 1;
  else if (e→size < #1000000 ∧ e→xsize < #1000000) b = 2;
  else b = 3;
  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 0: HPUT8(e→size); break;
  case 1: HPUT16(e→size); break;
  case 2: HPUT24(e→size); break;
  case 3: HPUT32(e→size); break;
  case b100 | 0: HPUT8(e→size); HPUT8(e→xsize); break;
  case b100 | 1: HPUT16(e→size); HPUT16(e→xsize); break;
  case b100 | 2: HPUT24(e→size); HPUT24(e→xsize); break;
  case b100 | 3: 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;
}

⟨ hcompress function 276 ⟩

extern bool option_compress;

void hput_directory(void)
{ int i;

  ⟨ set the file sizes for optional sections 293 ⟩
  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);
}
```

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. Where these files are found depends on the `-g` option. With that option given, the file names of the directory entrys are used unchanged. Without that option, the files are found in the *in_name*`.abs` and *in_name*`.rel` directories, as described in section 9.1.

Writing the short format: ⟹ ⋯

⟨ put functions 12 ⟩ +≡ (295)

```
static void hput_optional_sections(void)
{ int i;
  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;
    int file_name_length = 0;
    DBG(DBGDIR, "file␣%d:␣%s\n", dir[i].section_no, file_name);
    if (dir[i].xsize ≠ 0)
      DBG(DBGDIR,
          "Compressing␣of␣auxiliary␣files␣currently␣not␣supported");
    ⟨ without -g compute a local file_name 285 ⟩ 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␣directory␣size␣%u",
          fsize, dir[i].size);
  }
}
```

10 Definition Section

In a typical HINT file, there are many things that are used over and over again. For example the inter word 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 ⟩ +≡ (296)

```
%token DEFINITIONS "definitions"
```

⟨ scanning rules 3 ⟩ +≡ (297)

```
definitions      return DEFINITIONS;
```

⟨ parsing rules 5 ⟩ +≡ (298)

```
definition_section: START DEFINITIONS { hput_definitions_start(); }
      max_definitions definition_list
      END { hput_definitions_end(); };
definition_list: | definition_list def_node { REF($2.k, $2.n); };
```

Writing the long format: ⟹ − − −

⟨ write functions 19 ⟩ +≡ (299)

```
void hwrite_definitions_start(void)
{ section_no = 1; hwritef("<definitions");
}

void hwrite_definitions_end(void)
{ hwritef("\n>\n");
}
```

⟨ get functions 16 ⟩ +≡ (300)

```
void hget_definition_section(void)
{ DBG(DBGDEF, "Definitions\n");
  hget_section(1);
  hwrite_definitions_start();
  DBG(DBGDEF, "Reading list of maximum values\n");
  hget_max_definitions();
  ⟨ allocate data 250 ⟩
  hwrite_max_definitions();
  DBG(DBGDEF, "Reading list of definitions\n");
  while (hpos < hend)
  { def_t df; hget_def_node(&df);
    if (max_fixed[df.k] > max_default[df.k])
      QUIT("Definitions for kind %s not supported",
           definition_name[df.k]);
    if (df.n > max_ref[df.k] ∨ df.n ≤ max_fixed[df.k])
      QUIT("Definition %d for %s out of range [%d - %d]",
           df.n, definition_name[df.k], max_fixed[df.k] + 1, max_ref[df.k]);
  }
  hwrite_definitions_end();
}
```

Writing the short format: ⟹ · · ·

⟨ put functions 12 ⟩ +≡ (301)

```
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 $-1 \le max\_default[k] \le max\_ref[k] \le 255$. 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 is fixed for a given kind. Definitions with reference numbers lower 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 $max\_fixed[k] = {}^{\#}100 > max\_default[k]$.

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 302 ⟩ ≡ (302)

```
#define REF(K, N) if ((int)(N) > max_ref[K])
        QUIT("Reference␣%d␣to␣%s␣out␣of␣range␣[0␣-␣%d]", (N),
        definition_name[K], max_ref[K])
```

Used in 333.

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 is always 1 because at present, reference numbers—and therefore maximum values—are restricted to the range 0 to 255 in order to fit into a single byte. Other info values are reserved for future extensions. After reading the maximum values, we allocate data for the defininitions that come next.

Reading the long format: − − − ⟹

⟨ symbols 2 ⟩ +≡ (303)

```
%token MAX "max"
```

⟨ scanning rules 3 ⟩ +≡ (304)

```
max                return MAX;
```

⟨ parsing rules 5 ⟩ +≡ (305)

```
  max_definitions: START MAX max_list END
        { ⟨ allocate data 250 ⟩
        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); }
    | HYPHEN UNSIGNED { hset_max(hyphen_kind, $2); }
    | GLUE UNSIGNED { hset_max(glue_kind, $2); }
    | MATH UNSIGNED { hset_max(math_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); }
    | STREAM UNSIGNED { hset_max(stream_kind, $2); }
    | PAGE UNSIGNED { hset_max(page_kind, $2); }
    | RANGE UNSIGNED { hset_max(range_kind, $2); };
```

⟨ parsing functions 306 ⟩ ≡ (306)

```
void hset_max(kind_t k, int n)
{ RNG("Maximum", n, max_fixed[k] + 1, #FF);
  if (n > max_ref[k]) { max_ref[k] = n;
    DBG(DBGDEF, "Setting␣max␣%s␣to␣%d\n", definition_name[k], n);
  }
}
```

Used in 433.

Writing the long format: ⟹ − − −

⟨ write functions 19 ⟩ +≡ (307)

```
void hwrite_max_definitions(void)
{ kind_t k;
  hwrite_start(); hwritef("max");
  for (k = 0; k < 32; k++)
    if (max_ref[k] > max_default[k])
    { hwrite_start();
      hwritef("%s␣%d", definition_name[k], max_ref[k]);
      hwrite_end();
    }
  hwrite_end();
}
```

Reading the short format: $\cdots \Longrightarrow$

⟨ get file functions 260 ⟩ +≡ (308)

```
void hget_max_definitions(void)
{ kind_t k;
  ⟨ read the start byte a 14 ⟩
  if (a ≠ TAG(list_kind, 0)) QUIT("Start␣of␣maximum␣list␣expected");
  for (k = 0; k < 32; k++) max_ref[k] = max_default[k];
  while (true)
  { uint8_t n;
    if (hpos ≥ hend) QUIT("Unexpected␣end␣of␣maximum␣list");
    node_pos = hpos − hstart;
    HGETTAG(a);
    if (KIND(a) ≡ list_kind) break;
    if (INFO(a) ≠ 1) QUIT("Maximum␣info␣%d␣not␣supported", INFO(a));
    k = KIND(a);
    if (max_fixed[k] > max_default[k])
      QUIT("Maximum␣value␣for␣kind␣%s␣not␣supported",
           definition_name[k]);
    n = HGET8;
    RNG("Maximum␣number", n, max_ref[k], #FF);
    max_ref[k] = n;
    DBG(DBGDEF, "max(%s)␣=␣%d\n", definition_name[k], max_ref[k]);
    ⟨ read and check the end byte z 15 ⟩
  }
  if (INFO(a) ≠ 0) QUIT("End␣of␣maximum␣list␣with␣info␣%d", INFO(a));
}
```

Writing the short format: $\Longrightarrow \cdots$

⟨ put functions 12 ⟩ +≡ (309)

```
void hput_max_definitions(void)
{ kind_t k;
  DBG(DBGDEF, "Max␣Definitions␣Begin\n");
  HPUTTAG(list_kind, 0);
  for (k = 0; k < 32; k++)
    if (max_ref[k] > max_default[k]) {
      DBG(DBGDEF, "max(%s)␣=␣%d\n", definition_name[k], max_ref[k]);
      HPUTTAG(k, 1);
      HPUT8(max_ref[k]);
      HPUTTAG(k, 1);
    }
  HPUTTAG(list_kind, 0);
  DBG(DBGDEF, "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. Whenever we need this glue in the content section, we can say "<glue *71>". Because we restrict the number of definitions for every node type 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 the definitions. Instead they will write them in the new format immediately to the output.

Such a definition differs from a normal content node just by an extra byte value immediately following the keyword respectively start byte. The parser will handle definitions in any order, but of course, it does not harm to present them in a systematic way.

⟨ hint types 1 ⟩ +≡ (310)

```
typedef struct { kind_t k; int n; } def_t;
```

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

⟨ symbols 2 ⟩ +≡ (311)

```
%type < u > ref
%type < df > def_node
```

⟨ parsing rules 5 ⟩ +≡ (312)

```
ref: REFERENCE { RNG("Reference",$1,0,255); HPUT8($1); };
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($$,dimen_kind,$3); hput_tags($1,hput_dimen($4)); }
    | start MATH ref math END
        { DEF($$,math_kind,$3); hput_tags($1,hput_math(&($4))); }
    | 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 HYPHEN ref hyphen END
    { DEF($$, hyphen_kind, $3); hput_tags($1, hput_hyphen(&($4))); }
| start IMAGE ref image END
    { DEF($$, image_kind, $3); hput_tags($1, hput_image(&($4))); }
| start PARAM ref param_list 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)); };
```

⟨ parsing functions 306 ⟩ +≡ (313)

```
#define DEF(D, K, N) (D).k = K; (D).n = (N);
  RNG("Reference", N, max_fixed[K] + 1, max_ref[K]);
```

Writing the long format: ⟹ − − −

Reading the short format: ⋯ ⟹

⟨ get functions 16 ⟩ +≡ (314)

```
void hget_def_node(def_t *df)
{ ⟨ read the start byte a 14 ⟩
  df→k = KIND(a);
  df→n = HGET8;
  DBG(DBGTAGS, "Defining␣%s␣%d\n", definition_name[df→k], df→n);
  if (df→k ≡ range_kind) hget_range(INFO(a), df→n);
  else
  { hwrite_start(); hwritef("%s␣*%d", definition_name[df→k], df→n);
    if (df→k ≡ font_kind) hget_font_def(INFO(a), df→n);
    else if (df→k ≡ param_kind)
    { list_t l;
      l.k = param_kind; HGET_LIST(INFO(a), l); hwrite_param_list(&l); }
    else if (df→k ≡ page_kind) hget_page();
    else if (df→k ≡ dimen_kind) hget_dimen();
    else if (df→k ≡ xdimen_kind) { xdimen_t x;
      hget_xdimen(a, &x);
      hwrite_xdimen(&x);
    }
    else hget_content(a);
    hwrite_end();
  }
  ⟨ read and check the end byte z 15 ⟩
}
```

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. Therefore we restrict the definitions in parameter lists to such basic definitions.

⟨ parsing functions 306 ⟩ +≡ (315)

```
void check_param_def(def_t *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]);
  REF(df→k, df→n);
}
```

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.

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

⟨ symbols 2 ⟩ +≡ (316)

```
%token PARAM "param"
%type <u> def_list
%type <l> param_list param_list_node
```

⟨ scanning rules 3 ⟩ +≡ (317)

```
param        return PARAM;
```

⟨ parsing rules 5 ⟩ +≡ (318)

```
def_list: position | def_list def_node { check_param_def(&($2)); };
param_list: estimate def_list { $$.p = $2; $$.k = param_kind;
      $$.s = (hpos − hstart) − $2; };
param_list_node: start PARAM param_list END
    { if ($3.s > 0) hput_tags($1, hput_list($1 + 1, &($3))); };
```

Writing the long format: ⟹ − − −

⟨ write functions 19 ⟩ +≡ (319)

```
void hwrite_param_list(list_t *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)
  { def_t df; hget_def_node(&df);
  }
  hpos = hstart + h; hend = hstart + e;   /* restore hpos and hend */
}
void hwrite_param_list_node(list_t *l)
{ if (l→s ≠ 0)
  { hwrite_start(); hwritef("param");
    hwrite_param_list(l);
    hwrite_end();
  }
}
```

Reading the short format: ··· ⟹

⟨ get functions 16 ⟩ +≡ (320)

```
void hget_param_list_node(list_t *l)
{ if (KIND(*hpos) ≠ param_kind)
  { l→p = hpos − hstart; l→s = 0; l→k = param_kind; }
  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. Fonts are also the only data, that never occur directly 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 optional 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. Currently only `.pk` files are supported but the extension to `.ttf` and `.otf` files is imminent. Further, a font must specify an inter word glue and a default discretionary break. After that comes a list of up to 12 font specific parameters.

The optional font size specifies the desired font size. If omitted, we assign the value zero which implies 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. The info value will be *b001* if a font size is present, otherwise it is *b000*.

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 249 ⟩ +≡ (321)

char ∗∗*hfont_name*; /∗ dynamically allocated array of font names ∗/

⟨ hint basic types 6 ⟩ +≡ (322)

#**define** `MAX_FONT_PARAMS` 11

⟨ allocate data 250 ⟩ +≡ (323)

`ALLOCATE`(*hfont_name*, *max_ref* [*font_kind*] + 1, **char** ∗);

Reading the long format: − − − ⟹

⟨ symbols 2 ⟩ +≡ (324)

%**token** FONT `"font"`

%**type** $<$ *info* $>$ *font* *font_head*

⟨ scanning rules 3 ⟩ +≡ (325)

`font` **return** FONT;

⟨ parsing rules 5 ⟩ +≡ (326)

font: *font_head* *font_param_list*;

font_head: *string* UNSIGNED UNSIGNED
{ **uint8_t** *f* = $ < *u* > 0; *hfont_name*[*f*] = *strdup*($1);
$$ = *hput_font_head*(*f*, *hfont_name*[*f*], 0, $2, $3); }
| *string* *dimension* UNSIGNED UNSIGNED
{ **uint8_t** *f* = $ < *u* > 0; *hfont_name*[*f*] = *strdup*($1);
$$ = *hput_font_head*(*f*, *hfont_name*[*f*], $2, $3, $4); };

font_param_list: *glue_node* *hyphen_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* HYPHEN *fref* *hyphen* END { *hput_tags*($1, *hput_hyphen*(&($4))); }
| *start* GLUE *fref* *glue* END { *hput_tags*($1, *hput_glue*(&($4))); }
| *start* MATH *fref* *math* END { *hput_tags*($1, *hput_math*(&($4))); }
| *start* RULE *fref* *rule* END { *hput_tags*($1, *hput_rule*(&($4))); }
| *start* IMAGE *fref* *image* END { *hput_tags*($1, *hput_image*(&($4))); };

fref: REFERENCE
{ `RNG("Font␣parameter"`, $1, 0, `MAX_FONT_PARAMS`); `HPUT8`($1); };

Reading the short format: ··· ⟹

Writing the long format: ⟹ − − −

⟨ get functions 16 ⟩ +≡ (327)

```
static void hget_font_params(void)
{ hyphen_t h;

  hget_glue_node();
  hget_hyphen_node(&(h)); hwrite_hyphen_node(&h);
  DBG(DBGDEF, "Start␣font␣parameters\n");
  while (KIND(*hpos) ≠ font_kind)
  { def_t df;

    ⟨ read the start byte a 14 ⟩
    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 ≠ hyphen_kind ∧ df.k ≠ glue_kind ∧ df.k ≠ math_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[df.k], df.n);
    hget_content(a);
    hwrite_end();
    ⟨ read and check the end byte z 15 ⟩
  }
  DBG(DBGDEF, "End␣font␣parameters\n");
}
void hget_font_def(info_t i, uint8_t f)
{ char *n; dimen_t s = 0; uint16_t m, y;

  HGET_STRING(n); hwrite_string(n); hfont_name[f] = strdup(n);
  if (i & b001) { HGET32(s); if (s ≠ 0) 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 12 ⟩ +≡ (328)

```
uint8_t hput_font_head(uint8_t f, char *n, dimen_t s,
      uint16_t m, uint16_t y)
{ info_t i = b000;
  DBG(DBGDEF, "Defining␣font␣%d\nFont␣%s␣size␣0x%x\n", f, n, s);
  hput_string(n);
  if (s ≠ 0) { HPUT32(s); i = b001; }
  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>".

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

⟨ parsing rules 5 ⟩ +≡ (329)

```
xdimen_ref: REFERENCE { REF(xdimen_kind, $1); HPUT8($1); };
param_ref: REFERENCE { REF(param_kind, $1); HPUT8($1); };
stream_ref: REFERENCE { REF(stream_kind, $1); HPUT8($1); };
content_node: start PENALTY REFERENCE END
    { REF(penalty_kind, $3); HPUT8($3); hput_tags($1, TAG(penalty_kind, 0)); }
  | start KERN explicit REFERENCE END
    { REF(dimen_kind, $4); HPUT8($4);
      hput_tags($1, TAG(kern_kind, ($3) ? b100 : b000)); }
  | start KERN explicit XDIMEN REFERENCE END
    { REF(xdimen_kind, $5); HPUT8($5);
      hput_tags($1, TAG(kern_kind, ($3) ? b101 : b001)); }
  | start GLUE REFERENCE END
    { REF(glue_kind, $3); HPUT8($3); hput_tags($1, TAG(glue_kind, 0)); }
  | start LIGATURE REFERENCE END
    { REF(ligature_kind, $3); HPUT8($3); hput_tags($1, TAG(ligature_kind, 0)); }
  | start HYPHEN REFERENCE END
    { REF(hyphen_kind, $3); HPUT8($3); hput_tags($1, TAG(hyphen_kind, 0)); }
  | start MATH REFERENCE END
    { REF(math_kind, $3); HPUT8($3); hput_tags($1, TAG(math_kind, 0)); }
  | start RULE REFERENCE END
    { REF(rule_kind, $3); HPUT8($3); hput_tags($1, TAG(rule_kind, 0)); }
  | start IMAGE REFERENCE END
```

```
    { REF(image_kind, $3); HPUT8($3); hput_tags($1, TAG(image_kind, 0)); }
  | start LEADERS REFERENCE END
    { REF(leaders_kind, $3); HPUT8($3); hput_tags($1, TAG(leaders_kind, 0)); }
  | start BASELINE REFERENCE END
    { REF(baseline_kind, $3); HPUT8($3); hput_tags($1, TAG(baseline_kind, 0)); };
glue_node: start GLUE REFERENCE END
    { REF(glue_kind, $3); HPUT8($3); hput_tags($1, TAG(glue_kind, 0)); };
```

Reading the short format: ··· ⟹

⟨ cases to get content 18 ⟩ +≡ (330)

```
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(hyphen_kind, 0): HGET_REF(hyphen_kind); break;
case TAG(glue_kind, 0): HGET_REF(glue_kind); break;
case TAG(math_kind, 0): HGET_REF(math_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 17 ⟩ +≡ (331)

```
#define HGET_REF(K)
  { uint8_t n; n = HGET8; REF(K, n); hwritef("␣*%d", n); }
```

Writing the long format: ⟹ − − −

⟨ write functions 19 ⟩ +≡ (332)

```
  void hwrite_ref(uint8_t n)
  { hwritef("␣*%d", n);
  }
  void hwrite_ref_node(uint8_t 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 must 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, page templates, and page ranges. Each of these defaults has its own subsection below. Actually the defaults for extended dimensions and baselines are not needed by TEX's routines, but it is nice to have default values for the extended dimensions that represent `hsize`, `vsize`, or a zero baseline skip.

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. It is, however, possible to suppress definitions if the defined value is the same as the default.

For maximum flexibility and efficiency, this chapter defines a header file `hformat.h` and a C program `mkhformat` that generates the corresponding `hformat.c` file. The latter contains constant arrays containing the respective default information.

Here is the header file:

```
⟨ hformat.h 333 ⟩ ≡                                                    (333)
#ifndef _HFORMAT_H_
#define _HFORMAT_H_
  ⟨ debug macros 302 ⟩
  ⟨ debug constants 357 ⟩
  ⟨ hint macros 11 ⟩
  ⟨ hint basic types 6 ⟩
  ⟨ default names 336 ⟩
  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];
  extern int32_t int_defaults[MAX_INT_DEFAULT + 1];
```

```
  extern dimen_t dimen_defaults[MAX_DIMEN_DEFAULT + 1];
  extern xdimen_t xdimen_defaults[MAX_XDIMEN_DEFAULT + 1];
  extern glue_t glue_defaults[MAX_GLUE_DEFAULT + 1];
  extern baseline_t baseline_defaults[MAX_BASELINE_DEFAULT + 1];
#endif
```

And here is the *main* program of `mkhformat`:

```
⟨ mkhformat.c 334 ⟩ ≡                                                  (334)
#include <stdio.h>
#include "basetypes.h"
#include "hformat.h"
  int max_fixed[32], max_default[32];
  int32_t int_defaults[MAX_INT_DEFAULT + 1] = {0};
  dimen_t dimen_defaults[MAX_DIMEN_DEFAULT + 1] = {0};
  xdimen_t xdimen_defaults[MAX_XDIMEN_DEFAULT + 1] = {{0}};
  glue_t glue_defaults[MAX_GLUE_DEFAULT + 1] = {{{0}}};
  baseline_t baseline_defaults[MAX_BASELINE_DEFAULT + 1] = {{{{0}}}};

  int main(void)
  { kind_t k;
    int i;

    printf("#include␣\"basetypes.h\"\n"
    "#include␣\"hformat.h\"\n\n"
    ⟨ variables in hformat.c 358 ⟩);
    ⟨ define content_name and definition_name 7 ⟩
    for (k = 0; k < 32; k++) max_default[k] = −1, max_fixed[k] = #100;
    ⟨ define int_defaults 337 ⟩
    ⟨ define dimen_defaults 339 ⟩
    ⟨ define xdimen_defaults 341 ⟩
    ⟨ define baseline_defaults 345 ⟩
    ⟨ define page_defaults 347 ⟩
    ⟨ define range_defaults 349 ⟩
    ⟨ define max_ref, max_fixed and max_default 335 ⟩
    return 0;
  }
```

Above, we have set *max_default* to −1, meaning no defaults, and *max_fixed* to #`100`, meaning no definitions. The following subsections will 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. Then we can print out both arrays.

```
⟨ define max_ref, max_fixed and max_default 335 ⟩ ≡                    (335)
  max_fixed[font_kind] = max_fixed[ligature_kind] = max_fixed[hyphen_kind] =
        max_fixed[math_kind] = max_fixed[rule_kind] = max_fixed[image_kind] =
        max_fixed[leaders_kind] = max_fixed[param_kind] =
        max_fixed[stream_kind] = −1;
```

```
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 334.

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. It is never redefined. The default values are taken from `plain.tex`.

⟨ default names 336 ⟩ ≡ (336)

```
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 , } int_no_t;
#define MAX_INT_DEFAULT hang_after_no
```

Used in 333.

⟨ define *int_defaults* 337 ⟩ ≡ (337)

```
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;
```

$int\_defaults[widow\_penalty\_no] = 150;$
$int\_defaults[display\_widow\_penalty\_no] = 50;$
$int\_defaults[broken\_penalty\_no] = 100;$
$int\_defaults[pre\_display\_penalty\_no] = 10000;$
$int\_defaults[post\_display\_penalty\_no] = 0;$
$int\_defaults[inter\_line\_penalty\_no] = 0;$
$int\_defaults[double\_hyphen\_demerits\_no] = 10000;$
$int\_defaults[final\_hyphen\_demerits\_no] = 5000;$
$int\_defaults[adj\_demerits\_no] = 10000;$
$int\_defaults[looseness\_no] = 0;$
$int\_defaults[time\_no] = 720;$
$int\_defaults[day\_no] = 4;$
$int\_defaults[month\_no] = 7;$
$int\_defaults[year\_no] = 1776;$
$int\_defaults[hang\_after\_no] = 1;$
printf ("int32_t␣int_defaults[MAX_INT_DEFAULT+1]={");
for $(i = 0;\ i \le max\_default[int\_kind];\ i{+}{+})$ { *printf* ("%d", $int\_defaults[i]$);
  if $(i < max\_default[int\_kind])$ *printf* (",␣");
}
printf ("};\n\n"); Used in 334.

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 authors recommendation for the best "viewing experience".

⟨ default names 336 ⟩ +≡ (338)
typedef enum {
  $zero\_dimen\_no = 0, hsize\_dimen\_no = 1, vsize\_dimen\_no = 2,$
    $line\_skip\_limit\_no = 3, max\_depth\_no = 4, hang\_indent\_no = 5,$
    $emergency\_stretch\_no = 6, quad\_no = 7, math\_quad\_no = 8$
} **dimen_no_t**;
#**define** `MAX_DIMEN_DEFAULT` *math_quad_no*

⟨ define *dimen_defaults* 339 ⟩ ≡ (339)
$max\_default[dimen\_kind] =$ `MAX_DIMEN_DEFAULT`;
$max\_fixed[dimen\_kind] = zero\_dimen\_no;$
$dimen\_defaults[zero\_dimen\_no] = 0;$
$dimen\_defaults[hsize\_dimen\_no] = 6.5 * 72 *$ `ONE`;
$dimen\_defaults[vsize\_dimen\_no] = 8.9 * 72 *$ `ONE`;
$dimen\_defaults[line\_skip\_limit\_no] = 0;$
$dimen\_defaults[max\_depth\_no] = 4 *$ `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_t␣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 334.

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 336 ⟩ +≡ (340)
```
typedef enum {
  zero_xdimen_no = 0, hsize_xdimen_no = 1, vsize_xdimen_no = 2
} xdimen_no_t;
#define MAX_XDIMEN_DEFAULT vsize_xdimen_no
```

⟨ define *xdimen_defaults* 341 ⟩ ≡ (341)
```
max_default[xdimen_kind] = MAX_XDIMEN_DEFAULT;
max_fixed[xdimen_kind] = vsize_xdimen_no;
printf("xdimen_t␣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 334.

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 336 ⟩ +≡ (342)
```
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_t;
#define MAX_GLUE_DEFAULT par_fill_skip_no
```

⟨ define *dimen_defaults* 339 ⟩ +≡ (343)
```
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 = 10 * 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_t␣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");
```

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 336 ⟩ +≡ (344)

```
  typedef enum {
    zero_baseline_no = 0
  } baseline_no_t;
#define MAX_BASELINE_DEFAULT zero_baseline_no
```

⟨ define *baseline_defaults* 345 ⟩ ≡ (345)

max_default[*baseline_kind*] = `MAX_BASELINE_DEFAULT`;
max_fixed[*baseline_kind*] = *zero_baseline_no*;
{ **baseline_t** $z = \{\{\{0\}\}\}$;
printf(`"baseline_t␣baseline_defaults[MAX_BASELINE_DEFAULT+1]={{"`);
`PRINT_GLUE`(*z.bs*); *printf*(`",␣"`); `PRINT_GLUE`(*z.ls*);
printf(`",␣0x%x}};\n\n"`, *z.lsl*);
} Used in 334.

11.6 Page Templates

The zero page template is predefined, as well as stream 0 for the main content.

⟨ default names 336 ⟩ +≡ (346)
typedef enum {
zero_page_no = 0
} **page_no_t**;
#**define** `MAX_PAGE_DEFAULT` *zero_page_no*

⟨ define *page_defaults* 347 ⟩ ≡ (347)
max_default[*page_kind*] = `MAX_PAGE_DEFAULT`;
max_fixed[*page_kind*] = *zero_page_no*;
max_default[*stream_kind*] = 0;
max_fixed[*stream_kind*] = 0; Used in 334.

11.7 Page Ranges

The page range for the zero page template is the entire content section. It is predefined.

⟨ default names 336 ⟩ +≡ (348)
typedef enum {
zero_range_no = 0
} **range_no_t**;
#**define** `MAX_RANGE_DEFAULT` *zero_range_no*

⟨ define *range_defaults* 349 ⟩ ≡ (349)
max_default[*range_kind*] = `MAX_RANGE_DEFAULT`;
max_fixed[*range_kind*] = *zero_range_no*; Used in 334.

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 ⟩ +≡ (350)

```
%token CONTENT "content"
```

⟨ scanning rules 3 ⟩ +≡ (351)

```
content          return CONTENT;
```

⟨ parsing rules 5 ⟩ +≡ (352)

```
content_section: START CONTENT
    { hput_content_start(); } content_list END { hput_content_end();
      hput_range_defs(); };
```

Writing the Long Format: ⟹ − − −

⟨ write functions 19 ⟩ +≡ (353)

```
void hwrite_content_section(void)
{ section_no = 2;
  hwritef("<content");
  hsort_ranges();
  hget_content_section();
  hwritef("\n>\n");
}
```

Reading the Short Format: ⋯ ⟹

⟨ get functions 16 ⟩ +≡ (354)

```
void hget_content_section( )
{ DBG(DBGDIR, "Content\n");
  hget_section(2);
  hwrite_range( );
  while (hpos < hend) hget_content_node( );
}
```

Writing the Short Format: ⟹ ⋯

⟨ put functions 12 ⟩ +≡ (355)

```
void hput_content_start(void)
{ DBG(DBGDIR, "Content␣Section\n");
  section_no = 2;
  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 356 ⟩ ≡ (356)

```
  fprintf(stderr, "Usage:␣%s␣[options]␣filename%s\n", prog_name, in_ext);
  fprintf(stderr, "Options:\n"
  "\t␣-o␣file\t␣specify␣an␣output␣file␣name\n"
  "\t␣-g␣␣␣␣␣\t␣assume␣global␣names␣for␣auxiliar␣files\n"
  "\t␣-l␣␣␣␣␣\t␣redirect␣stderr␣to␣a␣log␣file\n"
  "\t␣-u␣␣␣␣␣\t␣enable␣writing␣utf8␣character␣codes\n"
  "\t␣-x␣␣␣␣␣\t␣enable␣writing␣hexadecimal␣character␣codes\n"
  "\t␣-c␣␣␣␣␣\t␣enable␣compression␣of␣section␣1␣and␣2\n");
#ifdef DEBUG
  fprintf(stderr,
      "\t␣-d␣XXX␣\t␣hexadecimal␣value.␣OR␣together␣these␣values:\n");
  fprintf(stderr, "\t\t\t␣XX=%03X␣␣␣basic␣debugging\n", DBGBASIC);
  fprintf(stderr, "\t\t\t␣XX=%03X␣␣␣tag␣debugging\n", DBGTAGS);
  fprintf(stderr, "\t\t\t␣XX=%03X␣␣␣node␣debugging\n", DBGNODE);
  fprintf(stderr, "\t\t\t␣XX=%03X␣␣␣definition␣debugging\n", DBGDEF);
  fprintf(stderr, "\t\t\t␣XX=%03X␣␣␣directory␣debugging\n", DBGDIR);
  fprintf(stderr, "\t\t\t␣XX=%03X␣␣␣range␣debugging\n", DBGRANGE);
  fprintf(stderr, "\t\t\t␣XX=%03X␣␣␣float␣debugging\n", DBGFLOAT);
  fprintf(stderr, "\t\t\t␣XX=%03X␣␣␣compression␣debugging\n", DBGCOMPRESS);
  fprintf(stderr, "\t\t\t␣XX=%03X␣␣␣buffer␣debugging\n", DBGBUFFER);
  fprintf(stderr, "\t\t\t␣XX=%03X␣␣␣flex␣debugging\n", DBGFLEX);
  fprintf(stderr, "\t\t\t␣XX=%03X␣␣␣bison␣debugging\n", DBGBISON);
  fprintf(stderr, "\t\t\t␣XX=%03X␣␣␣TeX␣debugging\n", DBGTEX);
  fprintf(stderr, "\t\t\t␣XX=%03X␣␣␣Page␣debugging\n", DBGPAGE);
  fprintf(stderr, "\t\t\t␣XX=%03X␣␣␣Font␣debugging\n", DBGFONT);
  fprintf(stderr, "\t\t\t␣XX=%03X␣␣␣Render␣debugging\n", DBGRENDER);
#endif
```

Used in 434, 435, and 437.

We define constants for different debug flags.

⟨ debug constants 357 ⟩ ≡ (357)

```
#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
```

Used in 333.

Next we define variables. Some of these variables go into `hformat.c` because it enables us to reuse them in other programs. Some are common variables that are needed in all three programs defined here. And some variables are just local variables in the *main* program.

The variable *in_name* is not local to *main* because it is used in the function *hget_map* (see page105). 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 auxiliar sections.

The `stretch` program will overwrite the *stem_name* using the name of the output file if it is set with the `-o` option.

⟨ variables in `hformat.c` 358 ⟩ ≡ (358)

```
"unsigned␣int␣debugflags=DBGNONE;\n"
```

Used in 334.

⟨ common variables 249 ⟩ +≡ (359)

```
bool option_utf8 = false;
bool option_hex = false;
bool option_force = false;
bool option_global = false;
bool option_compress = false;
char *in_name;
char *stem_name;
int stem_length = 0;
```

⟨ local variables in *main* 360 ⟩ ≡ (360)

```
char *prog_name;
char *in_ext;
char *out_ext;
char *file_name = NULL;
int file_name_length = 0;
bool option_log = false;
```

Used in 434, 435, and 437.

Processing the command line looks for options and then sets the input file name.

⟨ process the command line 361 ⟩ ≡ (361)

```
debugflags = DBGBASIC;
prog_name = argv[0];
if (argc < 2) goto explain_usage;
argv++;                                               /* skip the program name */
while (*argv ≠ NULL) {
  if ((*argv)[0] ≡ '-') { char option = (*argv)[1];
    switch (option) {
    default: goto explain_usage;
    case 'o': argv++;
      file_name_length = (int) strlen(*argv);
      ALLOCATE(file_name, file_name_length + 6, char);
                                                      /* extra space for extension */
      strcpy(file_name, *argv); break;
    case 'l': option_log = true; break;
    case 'u': option_utf8 = true; break;
    case 'x': option_hex = true; break;
    case 'f': option_force = true; break;
    case 'g': option_global = true; break;
    case 'c': option_compress = true; break;
    case 'd':
      argv++;
      if (*argv ≡ NULL) goto explain_usage;
      debugflags = strtol(*argv, NULL, 16);
      break;
    }
  }
  else                                                /* the input file name */
  { int path_length = (int) strlen(*argv);
    int ext_length = (int) strlen(in_ext);
    ALLOCATE(in_name, path_length + ext_length + 1, char);
    strcpy(in_name, *argv);
    if (path_length < ext_length ∨ strncmp(in_name + path_length − ext_length,
          in_ext, ext_length) ≠ 0) { strcat(in_name, in_ext);
      path_length += ext_length;
    }
    stem_length = path_length − ext_length;
    ALLOCATE(stem_name, stem_length + 6, char);
    strncpy(stem_name, in_name, stem_length);
    stem_name[stem_length] = 0;
    if (*(argv + 1) ≠ NULL) goto explain_usage;
  }
  argv++;
}
```

Used in 434, 435, and 437.

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 249 ⟩ +≡ (362)

```
FILE *hin = NULL, *hout = NULL;
```

⟨ variables in `hformat.c` 358 ⟩ +≡ (363)

```
"FILE␣*hlog=NULL;\n"
```

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 364 ⟩ ≡ (364)

```
#ifdef DEBUG
  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;
#else
  hlog = stderr;
#endif
```

Used in 434, 435, and 437.

Once we have established logging, we can try to open the other files.

⟨ open the input file 365 ⟩ ≡ (365)

```
  hin = fopen(in_name, "rb");
  if (hin ≡ NULL) QUIT("Unable␣to␣open␣input␣file␣%s", in_name);
```

Used in 434.

⟨ open the output file 366 ⟩ ≡ (366)

```
  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);
  }
  ⟨ make sure the path in file_name exists 287 ⟩
  hout = fopen(file_name, "wb");
  if (hout ≡ NULL) QUIT("Unable␣to␣open␣output␣file␣%s", file_name);
```

Used in 434 and 435.

The `stretch` program will replace the *stem_name* using the stem of the output file.

⟨ determine the *stem_name* from the output *file_name* 367 ⟩ ≡ (367)

$stem\_length = file\_name\_length - (\textbf{int})\ strlen(out\_ext);$
`ALLOCATE`$(stem\_name, stem\_length + 6, \textbf{char});$
$strncpy(stem\_name, file\_name, stem\_length);$
$stem\_name[stem\_length] = 0;$

Used in 435.

At the very end, we will close the files again.

⟨ close the input file 368 ⟩ ≡ (368)

if $(in\_name \neq$ `NULL`$)$ $free(in\_name);$
if $(hin \neq$ `NULL`$)$ $fclose(hin);$

Used in 434.

⟨ close the output file 369 ⟩ ≡ (369)

if $(file\_name \neq$ `NULL`$)$ $free(file\_name);$
if $(hout \neq$ `NULL`$)$ $fclose(hout);$

Used in 434 and 435.

⟨ close the log file 370 ⟩ ≡ (370)

if $(hlog \neq$ `NULL`$)$ $fclose(hlog);$
if $(stem\_name \neq$ `NULL`$)$ $free(stem\_name);$

Used in 434, 435, and 437.

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 371 ⟩ ≡ (371)

```
#ifndef _ERROR_H
#define _ERROR_H
#include <stdlib.h>
#include <stdio.h>
  extern FILE *hlog;
  extern uint8_t *hpos, *hstart;
#define LOG(...) (fprintf(hlog, __VA_ARGS__), fflush(hlog))
#define MESSAGE(...) (fprintf(hlog, __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 302 ⟩ +≡ (372)

```
#ifdef WIN32
#define SIZE_F "0x%x"
#else
#define SIZE_F "0x%zx"
#endif
#ifdef DEBUG
#define DBG(FLAGS, ...) ((debugflags & (FLAGS)) ? LOG(__VA_ARGS__) : 0)
#else
#define DBG(FLAGS, ...) 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 306 ⟩ +≡ (373)

```
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 374 ⟩ ≡ (374)

```
#ifdef DEBUG
#define YYDEBUG 1
  extern int yydebug;
#else
#define YYDEBUG 0
#endif
```

Used in 432 and 433.

Appendix

A 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: 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 section B.12 its *main* program is almost the same as the *main* program of the program `stretch`. The only difference is the removal of an output file and the replacement of the call to *hwrite_content_section* by *hskip_content_section*.

⟨ skip functions 375 ⟩ ≡ (375)

```
static void hskip_content_section(void)
{ DBG(DBGBASIC, "Skipping␣Content␣Section\n");
  hget_section(2);
  hpos = hend;
  while (hpos > hstart) hteg_content_node();
}
```

Used in 437.

The function *hteg_content_node* used above is the reverse version of the function *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`". There is of course no need for a long format file in reverse order, and hence, the `skip` program differs in another aspect from `stretch`: it does not produce any output and it 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 375 ⟩ +≡ (376)

```
static void hteg_content_node(void)
{ ⟨ skip the end byte z 377 ⟩
  hteg_content(z);
  ⟨ skip and check the start byte a 378 ⟩
}
static void hteg_content(uint8_t z)
{ switch (z)
  { ⟨ cases to skip content 385 ⟩
    default: 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 377 ⟩ ≡ (377)

```
uint8_t a, z;                        /* the start and the end byte */
uint32_t node_pos = hpos − hstart;
if (hpos ≤ hstart) return;
HTEGTAG(z);
```

Used in 376, 382, 394, 397, 400, and 417.

⟨ skip and check the start byte a 378 ⟩ ≡ (378)

```
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 376, 382, 394, 397, 400, and 417.

We replace the "GET" macros by the following "TEG" macros:

⟨ skip macros 379 ⟩ ≡ (379)

```
#define HTEG8(X) (hpos −= 1, (X) = hpos[0])
#define HTEG16(X) (hpos −= 2, (X) = (hpos[0] ≪ 8) + hpos[1])
#define HTEG24(X) (hpos −= 3, (X) = (hpos[0] ≪ 16) + (hpos[1] ≪ 8) + hpos[2])
#define HTEG32(X)
  (hpos −= 4, (X) = (hpos[0] ≪ 24) + (hpos[1] ≪ 16) + (hpos[2] ≪ 8) + hpos[3])
#define HTEGTAG(X) HTEG8(X), DBGTAG(X, hpos)
```

Used in 437.

Now we review step by step the different kinds of nodes.

A.1 Floating Point Numbers

⟨ skip functions 375 ⟩ +≡ (380)

```
static float32_t hteg_float32(void)
{ union { float32_t d; uint32_t bits; } u;
  HTEG32(u.bits);
  return u.d;
}
```

A.2 Extended Dimensions

```
⟨ skip macros 379 ⟩ +≡                                                   (381)
#define HTEG_XDIMEN(I, X)
  if ((I) & b001) HTEG32((X).v);
  if ((I) & b010) HTEG32((X).h);
  if ((I) & b100) HTEG32((X).w);
```

```
⟨ skip functions 375 ⟩ +≡                                                (382)
  static void hteg_xdimen_node(xdimen_t *x)
  { ⟨ skip the end byte z 377 ⟩
    switch (z) {
#if 0                          /* currently the info value 0 is not supported */
    case TAG(xdimen_kind, b000):                          /* see section 10.5 */
      { uint8_t n; HTEG8(n);
      } 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 378 ⟩
  }
```

A.3 Stretch and Shrink

```
⟨ skip macros 379 ⟩ +≡                                                   (383)
#define HTEG_STRETCH(S)
  { stch_t st; HTEG32(st.u); S.o = st.u & 3; st.u &= ~3; S.f = st.f; }
```

A.4 Glyphs

```
⟨ skip macros 379 ⟩ +≡                                                   (384)
#define HTEG_GLYPH (I, G) HTEG8 ((G).f);
  if (I ≡ 1) HTEG8((G).c);
  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 385 ⟩ ≡                                          (385)
case TAG(glyph_kind, 1): { glyph_t g; HTEG_GLYPH(1, g); } break;
case TAG(glyph_kind, 2): { glyph_t g; HTEG_GLYPH(2, g); } break;
case TAG(glyph_kind, 3): { glyph_t g; HTEG_GLYPH(3, g); } break;
```

```
case TAG(glyph_kind, 4): { glyph_t g; HTEG_GLYPH(4, g); } break;
```
Used in 376.

A.5 Penalties

⟨ skip macros 379 ⟩ +≡ (386)
```
#define HTEG_PENALTY(I, P)
  if (I ≡ 1) { int8_t n; HTEG8(n); P = n; }
  else { int16_t n; HTEG16(n); P = n; }
```

⟨ cases to skip content 385 ⟩ +≡ (387)
```
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;
```

A.6 Kerns

⟨ skip macros 379 ⟩ +≡ (388)
```
#define HTEG_KERN(I, X)
  if (((I) & b011) ≡ 2) HTEG32(X.w);
  else if (((I) & b011) ≡ 3)hteg_xdimen_node(&(X))
```

⟨ cases to skip content 385 ⟩ +≡ (389)
```
case TAG(kern_kind, b010): { xdimen_t x; HTEG_KERN(b010, x); } break;
case TAG(kern_kind, b011): { xdimen_t x; HTEG_KERN(b011, x); } break;
case TAG(kern_kind, b110): { xdimen_t x; HTEG_KERN(b110, x); } break;
case TAG(kern_kind, b111): { xdimen_t x; HTEG_KERN(b111, x); } break;
```

A.7 Mathematics

⟨ skip macros 379 ⟩ +≡ (390)
```
#define HTEG_MATH(I, M)
  if ((I) & b100) M.on = true; else M.on = false;
  if ((I) & b001) HTEG32(M.w); else M.w = 0;
```

⟨ cases to skip content 385 ⟩ +≡ (391)
```
case TAG(math_kind, b100): { math_t m; HTEG_MATH(b100, m); } break;
case TAG(math_kind, b010): { math_t m; HTEG_MATH(b010, m); } break;
case TAG(math_kind, b101): { math_t m; HTEG_MATH(b101, m); } break;
case TAG(math_kind, b011): { math_t m; HTEG_MATH(b011, m); } break;
```

A.8 Rules

⟨ skip macros 379 ⟩ +≡ (392)
```
#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 385 ⟩ +≡ (393)
```
case TAG(rule_kind, b011): { rule_t r; HTEG_RULE(b011, r); } break;
case TAG(rule_kind, b101): { rule_t r; HTEG_RULE(b101, r); } break;
case TAG(rule_kind, b001): { rule_t r; HTEG_RULE(b001, r); } break;
```

```
case TAG(rule_kind, b110): { rule_t r; HTEG_RULE(b110, r); } break;
case TAG(rule_kind, b111): { rule_t r; HTEG_RULE(b111, r); } break;
```

⟨ skip functions 375 ⟩ +≡ (394)

```
static void hteg_rule_node(void)
{ ⟨ skip the end byte z 377 ⟩
  if (KIND(z) ≡ rule_kind) { rule_t 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 378 ⟩
}
```

A.9 Glue

⟨ skip macros 379 ⟩ +≡ (395)

```
#define HTEG_GLUE(I, G)
  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) hteg_xdimen_node(&((G).w));
  else
  { (G).w.h = 0.0; (G).w.v = 0.0;
    if ((I) & b100) HTEG32((G).w.w); else (G).w.w = 0; }
```

⟨ cases to skip content 385 ⟩ +≡ (396)

```
case TAG(glue_kind, b001): { glue_t g; HTEG_GLUE(b001, g); } break;
case TAG(glue_kind, b010): { glue_t g; HTEG_GLUE(b010, g); } break;
case TAG(glue_kind, b011): { glue_t g; HTEG_GLUE(b011, g); } break;
case TAG(glue_kind, b100): { glue_t g; HTEG_GLUE(b100, g); } break;
case TAG(glue_kind, b101): { glue_t g; HTEG_GLUE(b101, g); } break;
case TAG(glue_kind, b110): { glue_t g; HTEG_GLUE(b110, g); } break;
case TAG(glue_kind, b111): { glue_t g; HTEG_GLUE(b111, g); } break;
```

⟨ skip functions 375 ⟩ +≡ (397)

```
static void hteg_glue_node(void)
{ ⟨ skip the end byte z 377 ⟩
  if (INFO(z) ≡ b000) HTEG_REF(glue_kind);
  else { glue_t g; HTEG_GLUE(INFO(z), g); }
  ⟨ skip and check the start byte a 378 ⟩
}
```

A.10 Boxes

⟨ skip macros 379 ⟩ +≡ (398)

```
#define HTEG_BOX(I, B) hteg_list (&(B.l));
  if ((I) & b100)
  { HTEG8(B.s); 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 385 ⟩ +≡ (399)

```
case TAG(hbox_kind, b000): { box_t b; HTEG_BOX(b000, b); } break;
case TAG(hbox_kind, b001): { box_t b; HTEG_BOX(b001, b); } break;
case TAG(hbox_kind, b010): { box_t b; HTEG_BOX(b010, b); } break;
case TAG(hbox_kind, b011): { box_t b; HTEG_BOX(b011, b); } break;
case TAG(hbox_kind, b100): { box_t b; HTEG_BOX(b100, b); } break;
case TAG(hbox_kind, b101): { box_t b; HTEG_BOX(b101, b); } break;
case TAG(hbox_kind, b110): { box_t b; HTEG_BOX(b110, b); } break;
case TAG(hbox_kind, b111): { box_t b; HTEG_BOX(b111, b); } break;
case TAG(vbox_kind, b000): { box_t b; HTEG_BOX(b000, b); } break;
case TAG(vbox_kind, b001): { box_t b; HTEG_BOX(b001, b); } break;
case TAG(vbox_kind, b010): { box_t b; HTEG_BOX(b010, b); } break;
case TAG(vbox_kind, b011): { box_t b; HTEG_BOX(b011, b); } break;
case TAG(vbox_kind, b100): { box_t b; HTEG_BOX(b100, b); } break;
case TAG(vbox_kind, b101): { box_t b; HTEG_BOX(b101, b); } break;
case TAG(vbox_kind, b110): { box_t b; HTEG_BOX(b110, b); } break;
case TAG(vbox_kind, b111): { box_t b; HTEG_BOX(b111, b); } break;
```

⟨ skip functions 375 ⟩ +≡ (400)

```
  static void hteg_hbox_node(void)
  { box_t b;
    ⟨ skip the end byte z 377 ⟩
    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 378 ⟩
  }
  static void hteg_vbox_node(void)
  { box_t b;
    ⟨ skip the end byte z 377 ⟩
    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 378 ⟩
  }
```

A.11 Extended Boxes

⟨ skip macros 379 ⟩ +≡ (401)

```
#define HTEG_SET(I)
  { list_t l; hteg_list(&l); }
  { stretch_t m; HTEG_STRETCH(m); }
  { stretch_t p; HTEG_STRETCH(p); }
  if ((I) & b010) { dimen_t a; HTEG32(a); }
  { dimen_t w; HTEG32(w); }
```

```
  { dimen_t d; if ((I) & b001) HTEG32(d); else d = 0; }
  { dimen_t h; HTEG32(h); }
  if ((I) & b100) { xdimen_t x;
    hteg_xdimen_node(&x); }
  else HTEG_REF(xdimen_kind);
#define HTEG_PACK(I)
  { list_t l; hteg_list(&l); }
  if ((I) & b001) { dimen_t d; HTEG32(d); }
  if ((I) & b100) { xdimen_t x;
    hteg_xdimen_node(&x); } else HTEG_REF(xdimen_kind);
```

⟨ cases to skip content 385 ⟩ +≡ (402)

```
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(b000); break;
case TAG(hpack_kind, b010): HTEG_PACK(b010); break;
case TAG(hpack_kind, b100): HTEG_PACK(b100); break;
case TAG(hpack_kind, b110): HTEG_PACK(b110); break;

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

A.12 Leaders

⟨ skip macros 379 ⟩ +≡ (403)

```
#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 (KIND(hpos[−1]) ≡ glue_kind) hteg_glue_node();
```

⟨ cases to skip content 385 ⟩ +≡ (404)

```
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;
```

A.13 Baseline Skips

⟨ skip macros 379 ⟩ +≡ (405)

```
#define HTEG_BASELINE(I, B)
  if ((I) & b001) HTEG32((B).lsl); else B.lsl = 0;
  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; }
```

⟨ cases to skip content 385 ⟩ +≡ (406)

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

A.14 Ligatures

⟨ skip macros 379 ⟩ +≡ (407)

```
#define HTEG_LIG(I, L)
  if ((I) ≡ 7) HTEG8((L).l.s); else (L).l.s = (I);
  hpos −= (L).l.s; (L).l.p = hpos − hstart;
  if ((I) ≡ 7) { uint8_t n; HTEG8(n);
    if (n ≠ (L).l.s)
      QUIT("Sizes␣in␣ligature␣don't␣match␣%d!=%d", (L).l.s, n);
  }
  HTEG8((L).f);
```

⟨ cases to skip content 385 ⟩ +≡ (408)

```
case TAG(ligature_kind, 1): { lig_t l; HTEG_LIG(1, l); } break;
case TAG(ligature_kind, 2): { lig_t l; HTEG_LIG(2, l); } break;
```

case TAG(*ligature_kind*, 3): { **lig_t** *l*; HTEG_LIG(3, *l*); } **break**;
case TAG(*ligature_kind*, 4): { **lig_t** *l*; HTEG_LIG(4, *l*); } **break**;
case TAG(*ligature_kind*, 5): { **lig_t** *l*; HTEG_LIG(5, *l*); } **break**;
case TAG(*ligature_kind*, 6): { **lig_t** *l*; HTEG_LIG(6, *l*); } **break**;
case TAG(*ligature_kind*, 7): { **lig_t** *l*; HTEG_LIG(7, *l*); } **break**;

A.15 Hyphenation

⟨ skip macros 379 ⟩ +≡ (409)
#**define** HTEG_HYPHEN(*I*, H)
  if ((*I*) & *b001*) HTEG8((H).*r*); **else** (H).*r* = 0;
  if ((*I*) & *b010*) *hteg_list*(&((H).*q*));
  else { (H).*q*.*p* = *hpos* − *hstart*; (H).*q*.*s* = 0; (H).*q*.*k* = *list_kind*; }
  if ((*I*) & *b100*) *hteg_list*(&((H).*p*));
  else { (H).*p*.*p* = *hpos* − *hstart*; (H).*p*.*s* = 0; (H).*p*.*k* = *list_kind*; }

⟨ cases to skip content 385 ⟩ +≡ (410)
case TAG(*hyphen_kind*, *b001*):
  { **hyphen_t** *h*; HTEG_HYPHEN(*b001*, *h*); } **break**;
case TAG(*hyphen_kind*, *b010*):
  { **hyphen_t** *h*; HTEG_HYPHEN(*b010*, *h*); } **break**;
case TAG(*hyphen_kind*, *b011*):
  { **hyphen_t** *h*; HTEG_HYPHEN(*b011*, *h*); } **break**;
case TAG(*hyphen_kind*, *b100*):
  { **hyphen_t** *h*; HTEG_HYPHEN(*b100*, *h*); } **break**;
case TAG(*hyphen_kind*, *b101*):
  { **hyphen_t** *h*; HTEG_HYPHEN(*b101*, *h*); } **break**;
case TAG(*hyphen_kind*, *b110*):
  { **hyphen_t** *h*; HTEG_HYPHEN(*b110*, *h*); } **break**;
case TAG(*hyphen_kind*, *b111*):
  { **hyphen_t** *h*; HTEG_HYPHEN(*b111*, *h*); } **break**;

A.16 Paragraphs

⟨ skip macros 379 ⟩ +≡ (411)
#**define** HTEG_PAR(*I*)
  { **list_t** *l*; *hteg_list*(&*l*); }
  if ((*I*) & *b010*) { **list_t** *l*; *hteg_param_list_node*(&*l*); }
  else HTEG_REF(*param_kind*);
  if ((*I*) & *b100*) { **xdimen_t** *x*; *hteg_xdimen_node*(&*x*); }
  else HTEG_REF(*xdimen_kind*);

⟨ cases to skip content 385 ⟩ +≡ (412)
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**;

A.17 Displays

⟨ skip macros 379 ⟩ +≡ (413)

```
#define HTEG_DISPLAY(I)
  if ((I) & b001) hteg_hbox_node();
  { list_t l; hteg_list(&l); }
  if ((I) & b010) hteg_hbox_node();
  if ((I) & b100) { list_t l; hteg_param_list_node(&l); } else
    HTEG_REF(param_kind);
```

⟨ cases to skip content 385 ⟩ +≡ (414)

```
case TAG(display_kind, b000): HTEG_DISPLAY(b000); break;
case TAG(display_kind, b001): HTEG_DISPLAY(b001); break;
case TAG(display_kind, b010): HTEG_DISPLAY(b010); break;
case TAG(display_kind, b100): HTEG_DISPLAY(b100); break;
case TAG(display_kind, b101): HTEG_DISPLAY(b101); break;
case TAG(display_kind, b110): HTEG_DISPLAY(b110); break;
```

A.18 Images

⟨ skip macros 379 ⟩ +≡ (415)

```
#define HTEG_IMAGE(I, X)
  if (I & b001) { HTEG_STRETCH((X).m);
    HTEG_STRETCH((X).p); }
  else { (X).p.f = (X).m.f = 0.0;
    (X).p.o = (X).m.o = normal_o; }
  if (I & b010) { HTEG32((X).h);
    HTEG32((X).w); }
  else (X).w = (X).h = 0;
  HTEG16((X).n);
```

⟨ cases to skip content 385 ⟩ +≡ (416)

```
case TAG(image_kind, b100): { image_t x; HTEG_IMAGE(b100, x); } break;
case TAG(image_kind, b101): { image_t x; HTEG_IMAGE(b101, x); } break;
case TAG(image_kind, b110): { image_t x; HTEG_IMAGE(b110, x); } break;
case TAG(image_kind, b111): { image_t x; HTEG_IMAGE(b111, x); } break;
```

A.19 Plain Lists, Texts, and Parameter Lists

⟨ skip functions 375 ⟩ +≡ (417)

```
  static void hteg_size_boundary(info_t info)
  { uint32_t n;
    if (info < 2) return;
    HTEG8(n);
    if (n-1 ≠ #100-info) QUIT("List␣size␣boundary␣byte␣0x%x␣does␣not␣m\
          atch␣info␣value␣%d␣at␣"SIZE_F, n, info, hpos - hstart);
  }
  static uint32_t hteg_list_size(info_t info)
  { uint32_t n;
```

```
    if (info ≡ 1) return 0;
    else if (info ≡ 2) HTEG8(n);
    else if (info ≡ 3) HTEG16(n);
    else if (info ≡ 4) HTEG24(n);
    else if (info ≡ 5) HTEG32(n);
    else QUIT("List␣info␣%d␣must␣be␣1,␣2,␣3,␣4,␣or␣5", info);
    return n;
  }
  static void hteg_list(list_t *l){ ⟨skip the end byte z 377⟩
      if (KIND(z) ≠ list_kind ∧ KIND(z) ≠ text_kind ∧
          KIND(z) ≠ param_kind)
      { hpos++;
        l→p = hpos − hstart; l→s = 0; l→k = list_kind; }
      else { uint32_t s;
        l→k = KIND(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 378⟩
      }
      }
      static void hteg_param_list_node(list_t *l)
      { if (KIND(*(hpos − 1)) ≠ param_kind) return;
        hteg_list(l);
      }
```

A.20 Adjustments

⟨cases to skip content 385⟩ +≡ (418)

```
case TAG(adjust_kind, b001): { list_t l; hteg_list(&l); } break;
```

A.21 Tables

⟨skip macros 379⟩ +≡ (419)

```
#define HTEG_TABLE(I)
  { list_t l; hteg_list(&l); }
  { list_t l; hteg_list(&l); }
  if ((I) & b100) { xdimen_t x; hteg_xdimen_node(&x); }
  else HTEG_REF(xdimen_kind)
```

⟨cases to skip content 385⟩ +≡ (420)

```
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_t 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; HTEG8(n); } break;
```

A.22 Stream Nodes

⟨ skip macros 379 ⟩ +≡ (421)

```
#define HTEG_STREAM(I)
  { list_t l; hteg_list(&l); }
  if ((I) & b001) { scaled_t d; hteg_glue_node(); HTEG32(d); }
  if ((I) & b010) { int16_t p; HTEG16(p); }
  if ((I) & b100) { scaled_t h; HTEG32(h); }
  HTEG_REF(stream_kind);
```

⟨ cases to skip content 385 ⟩ +≡ (422)

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

A.23 References

⟨ skip macros 379 ⟩ +≡ (423)

```
#define HTEG_REF(K) do { uint8_t n; HTEG8(n); } while (false)
```

⟨ cases to skip content 385 ⟩ +≡ (424)

```
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`(*hyphen_kind*, 0): `HTEG_REF`(*hyphen_kind*); **break**;
case `TAG`(*glue_kind*, 0): `HTEG_REF`(*glue_kind*); **break**;
case `TAG`(*math_kind*, 0): `HTEG_REF`(*math_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 Code and Header Files

B.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 425⟩ ≡ (425)

```
#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
  typedef float float32_t;
  typedef double float64_t;
#define INT32_MAX (2147483647)
#define PRIx64 "I64x"
#pragma warning(disable:4244 4996 4127)
#else
#include <stdint.h>
#include <stdbool.h>
```

```
#include <inttypes.h>
  typedef float float32_t;
  typedef double float64_t;
#ifdef WIN32
#include <io.h>
#endif
#endif
#if __SIZEOF_FLOAT__ ≠ 4
#error float32␣type␣must␣have␣size␣4
#endif
#if __SIZEOF_DOUBLE__ ≠ 8
#error float64␣type␣must␣have␣size␣8
#endif
#endif
```

B.2 hformat.h

The `hformat.c` file contains variables and functions that are needed in other compilation units. Together with the required type and macro definitions, the necessary information is contained in the `hformat.h` header file.

⟨ write function declarations 426 ⟩ ≡ (426)

```
#define hwritec(c) putc(c, hout)
#define hwritef(...) fprintf(hout, __VA_ARGS__)
  extern void hwrite_range(void);
  extern void hwrite_charcode(uint32_t c);
  extern void hwrite_ref_node(uint8_t k, uint8_t n);
  extern void hwrite_ref(uint8_t n);
  extern void hsort_ranges(void);
```

Used in 435.

B.3 hget.h

The `hget.h` file contains function prototypes for all the functions that read the short format.

⟨ get function declarations 427 ⟩ ≡ (427)

```
  extern void hget_content_node(void);
  extern int txt_length;
  extern int hget_txt(void);
  extern uint32_t hget_utf8(void);
  extern void hget_def_node(def_t *df);
  extern void hget_content_section(void);
  extern void hget_content(uint8_t a);
  extern void hget_xdimen_node(xdimen_t *x);
  extern float32_t hget_float32(void);
  extern void hget_list(list_t *l);
  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_node(list_t *l);
extern uint32_t hget_list_size(info_t info);
extern void hget_size_boundary(info_t info);
extern void hget_max_definitions(void);
extern void hget_font_def(info_t i, uint8_t f);
```

Used in 435.

⟨ hget.h 428 ⟩ ≡ (428)

```
⟨ get file macros 35 ⟩
⟨ directory entry type 280 ⟩
extern entry_t *dir;
extern uint16_t section_no, max_section_no;
extern uint8_t *hpos, *hstart, *hend;
extern void hget_map(void);
extern void hget_unmap(void);
extern void new_directory(uint32_t entries);
extern void hset_entry(entry_t *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_t *e);
extern void hget_directory(void);
extern void hclear_dir(void);
extern bool hcheck_banner(char *magic);
extern void hget_max_definitions(void);
```

B.4 hget.c

⟨ hget.c 429 ⟩ ≡ (429)

```
#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 "hformat.h"
#include "hget.h"
  uint8_t *hpos = NULL, *hstart = NULL, *hend = NULL;
  ⟨ map functions 271 ⟩
  ⟨ function to check the banner 259 ⟩
  ⟨ directory functions 281 ⟩
  ⟨ get file macros 35 ⟩
  ⟨ get file functions 260 ⟩
```

B.5 hput.h

The `hput.h` file contains function prototypes for all the functions that write the short format.

⟨ hput.h 430 ⟩ ≡ (430)

```
⟨ put macros 268 ⟩
⟨ hint macros 11 ⟩
⟨ hint types 1 ⟩
⟨ directory entry type 280 ⟩
extern entry_t *dir;
extern uint16_t section_no, max_section_no;
extern uint8_t *hpos, *hstart, *hend;
extern int next_range;
extern range_pos_t *range_pos;
extern int *page_on;
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 hput_tags(uint32_t pos, uint8_t tag);
extern uint8_t hput_glyph(glyph_t *g);
extern uint8_t hput_xdimen(xdimen_t *x);
extern uint8_t hput_int(int32_t p);
extern uint8_t hput_math(math_t *m);
extern uint8_t hput_rule(rule_t *r);
extern uint8_t hput_glue(glue_t *g);
extern uint8_t hput_list(uint32_t size_pos, list_t *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_t *d);
extern void hput_txt_local(uint8_t n);
extern info_t hput_box_dimen(dimen_t h, dimen_t d, dimen_t w);
extern info_t hput_box_shift(dimen_t a);
extern info_t hput_box_glue_set(int8_t s, float32_t r, order_t o);
extern void hput_stretch(stretch_t *s);
extern uint8_t hput_kern(kern_t *k);
extern void hput_utf8(uint32_t c);
extern uint8_t hput_ligature(lig_t *l);
extern uint8_t hput_hyphen(hyphen_t *h);
extern uint8_t hput_item(uint32_t n);
extern uint8_t hput_image(image_t *x);
extern void hput_string(char *str);
```

```
extern void hput_range(uint8_t pg, bool on);
extern void hput_max_definitions(void);
extern uint8_t hput_dimen(dimen_t d);
extern uint8_t hput_font_head(uint8_t f, char *n, dimen_t s, uint16_t
    m, uint16_t y);
extern void hput_range_defs(void);              /* declarations for HiTeX */
extern void hput_xdimen_node(xdimen_t *x);
extern void hput_directory(void);
extern void hput_hint(char *str);
extern void hput_list_size(uint32_t n, int i);
```

B.6 hput.c

```
⟨ hput.c 431 ⟩ ≡                                                    (431)
#include "basetypes.h"
#include <string.h>
#include <ctype.h>
#include <sys/stat.h>
#include <zlib.h>
#include "error.h"
#include "hformat.h"
#include "hput.h"
  uint8_t *hpos = NULL, *hstart = NULL, *hend = NULL;
  FILE *hout;
  int version = 1, subversion = 0;
  bool option_compress = false;
  bool option_global = false;
  int next_range;
  range_pos_t *range_pos;
  int *page_on;
  char *stem_name = NULL;
  int stem_length = 0;
  ⟨ directory functions 281 ⟩
  ⟨ function to write the banner 262 ⟩
  ⟨ put functions 12 ⟩
```

B.7 shrink.l

The definitions for lex are collected in the file shrink.l

```
⟨ shrink.l 432 ⟩ ≡                                                  (432)
%{
#include "basetypes.h"
#include <unistd.h>
#include "error.h"
#include "hformat.h"
  ⟨ hint types 1 ⟩
  ⟨ enable bison debugging 374 ⟩
```

```
#include "shrink.tab.h"
  ⟨ scanning macros 20 ⟩
  ⟨ scanning functions 59 ⟩
  int yywrap(void)
  { return 1;
  }
#ifdef _MSC_VER
#pragma warning ( disable: 4267 )
#endif
%}
%option yylineno batch stack
%option debug
%option nounistd nounput noinput noyy_top_state
  ⟨ scanning definitions 21 ⟩
%%
  ⟨ 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);
%%
```

B.8 `shrink.y`

The grammar rules for bison are collected in the file `shrink.y`.

```
⟨ shrink.y 433 ⟩ ≡                                                    (433)
%{
#include "basetypes.h"
#include <string.h>
#include <math.h>
#include "error.h"
#include "hformat.h"
#include "hput.h"
  char **hfont_name;
  extern void hset_entry(entry_t *e, uint16_t i, uint32_t size, uint32_t xsize,
      char *file_name);
  ⟨ enable bison debugging 374 ⟩
  extern int yylex(void);
  ⟨ parsing functions 306 ⟩
%}

%union { uint32_t u; int32_t i; char *s; float64_t f; glyph_t c;
  dimen_t d; stretch_t st; xdimen_t xd; kern_t kt; rule_t r; glue_t g;
  math_t m;  image_t x; list_t l; box_t h; hyphen_t hy; lig_t lg; ref_t rf;
  info_t info; order_t o; bool b; def_t df; }
```

```
%error_verbose
%start hint
  ⟨ symbols 2 ⟩

%%
  ⟨ parsing rules 5 ⟩
%%
```

B.9 shrink.c

`shrink` is a C program translating a HINT file in long format into a HINT file in short format.

⟨ shrink.c 434 ⟩ ≡ (434)

```
#include "basetypes.h"
#include <string.h>
#include <ctype.h>
#include <sys/stat.h>
#include <zlib.h>
#include "error.h"
#include "hformat.h"
  ⟨ hint types 1 ⟩
#include "shrink.tab.h"
  extern void yyset_debug(int lex_debug);
  extern int yylineno;
  extern FILE *yyin, *yyout;
  extern int yyparse(void);
  ⟨ put macros 268 ⟩
  ⟨ common variables 249 ⟩
  ⟨ function to check the banner 259 ⟩
  ⟨ directory entry type 280 ⟩
  ⟨ directory functions 281 ⟩
  ⟨ function to write the banner 262 ⟩
  ⟨ put functions 12 ⟩
  int main(int argc, char *argv[])
  { ⟨ local variables in main 360 ⟩
    in_ext = ".HINT";
    out_ext = ".hnt";
    ⟨ process the command line 361 ⟩
    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 364 ⟩
```

```
    ⟨ open the input file 365 ⟩
    ⟨ open the output file 366 ⟩
    yyin = hin;
    yyout = hlog;
    ⟨ read the banner 261 ⟩
    hcheck_banner("HINT");
    yylineno++;
    DBG(DBGBISON | DBGFLEX, "Parsing␣Input\n");
    yyparse();
    hput_directory();
    hput_hint("shrink");
    ⟨ close the output file 369 ⟩
    ⟨ close the input file 368 ⟩
    ⟨ close the log file 370 ⟩
    return 0;
  explain_usage: ⟨ explain usage 356 ⟩
    return 1;
  }
```

B.10 stretch.c

stretch is a C program translating a HINT file in short format into a HINT file in long format.

⟨ stretch.c 435 ⟩ ≡ (435)

```
#include "basetypes.h"
#include <math.h>
#include <string.h>
#include <ctype.h>
#include <zlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "error.h"
#include "hformat.h"
  ⟨ hint types 1 ⟩
  ⟨ common variables 249 ⟩
  ⟨ map functions 271 ⟩
  ⟨ function to check the banner 259 ⟩
  ⟨ function to write the banner 262 ⟩
  ⟨ directory entry type 280 ⟩
  ⟨ directory functions 281 ⟩
  ⟨ get file macros 35 ⟩
  ⟨ get file functions 260 ⟩
  ⟨ write function declarations 426 ⟩⟨ get function declarations 427 ⟩
  ⟨ write functions 19 ⟩
  ⟨ get macros 17 ⟩
  ⟨ get functions 16 ⟩
```

```
int main(int argc, char *argv[])
{ ⟨ local variables in main 360 ⟩
  in_ext = ".hnt";
  out_ext = ".HINT";
  ⟨ process the command line 361 ⟩
  ⟨ open the log file 364 ⟩
  ⟨ open the output file 366 ⟩
  ⟨ determine the stem_name from the output file_name 367 ⟩
  hget_map();
  hget_banner();
  hcheck_banner("hint");
  hput_banner("HINT", "stretch");
  hget_directory();
  hwrite_directory();
  hget_definition_section();
  hwrite_content_section();
  hwrite_aux_files();
  hget_unmap();
  ⟨ close the output file 369 ⟩
  ⟨ close the log file 370 ⟩
  return 0;
explain_usage: ⟨ explain usage 356 ⟩
  return 1;
}
```

B.11 hteg.h

⟨ skip function declarations 436 ⟩ ≡ (436)

```
static void hteg_content_node(void);
static void hteg_content(uint8_t z);
static void hteg_xdimen_node(xdimen_t *x);
static void hteg_list(list_t *l);
static void hteg_param_list_node(list_t *l);
static float32_t hteg_float32(void);
static void hteg_rule_node(void);
static void hteg_hbox_node(void);
static void hteg_vbox_node(void);
static void hteg_glue_node(void);
```

Used in 437.

B.12 skip.c

skip is a C program reading the content section of a HINT file in short format backwards.

⟨ skip.c 437 ⟩ ≡ (437)

```
#include "basetypes.h"
#include <string.h>
#include <zlib.h>
```

```
#include <sys/stat.h>
#include <fcntl.h>
#include "error.h"
#include "hformat.h"
  ⟨ hint types 1 ⟩
  ⟨ common variables 249 ⟩
  ⟨ map functions 271 ⟩
  ⟨ function to check the banner 259 ⟩
  ⟨ directory entry type 280 ⟩
  ⟨ directory functions 281 ⟩
  ⟨ get file macros 35 ⟩
  ⟨ get file functions 260 ⟩
  ⟨ skip macros 379 ⟩
  ⟨ skip function declarations 436 ⟩
  ⟨ skip functions 375 ⟩
  int main(int argc, char *argv[])
  { ⟨ local variables in main 360 ⟩
    in_ext = ".hnt";
    out_ext = ".bak";
    ⟨ process the command line 361 ⟩
    ⟨ open the log file 364 ⟩
    hget_map();
    hget_banner();
    hcheck_banner("hint");
    hget_directory();
    hskip_content_section();
    hget_unmap();
    ⟨ close the log file 370 ⟩
    return 0;
  explain_usage: ⟨ explain usage 356 ⟩
    return 1;
  }
```

C List of Format Definitions

C.1 Reading the Long Format

C.2 Writing the Long Format

C.3 Reading the Short Format

C.4 Writing the Short Format

Cross Reference of Code

⟨ put functions ⟩ Defined in section 12, 13, 34, 51, 72, 83, 91, 93, 104, 112, 122, 132, 140, 151, 161, 174, 187, 195, 204, 226, 234, 257, 264, 267, 274, 294, 295, 301, 309, 328, and 355. Used in sections 431 and 434.
⟨ put macros ⟩ Defined in section 268 and 269. Used in sections 430 and 434.
⟨ read and check the end byte z ⟩ Defined in section 15. Used in sections 16, 90, 121, 131, 138, 150, 160, 203, 238, 291, 308, 314, and 327.
⟨ read the banner ⟩ Defined in section 261. Used in section 434.
⟨ read the mantissa ⟩ Defined in section 61. Used in section 59.
⟨ read the optional exponent ⟩ Defined in section 63. Used in section 59.
⟨ read the optional sign ⟩ Defined in section 60. Used in section 59.
⟨ read the start byte a ⟩ Defined in section 14. Used in sections 16, 90, 121, 131, 138, 150, 160, 203, 238, 291, 308, 314, and 327.
⟨ return the binary representation ⟩ Defined in section 64. Used in section 59.
⟨ scanning definitions ⟩ Defined in section 21, 30, 37, 39, 41, 43, and 142. Used in section 432.
⟨ scanning functions ⟩ Defined in section 59. Used in section 432.
⟨ scanning macros ⟩ Defined in section 20, 23, 26, 29, 38, 40, 42, 44, 55, 58, and 145. Used in section 432.
⟨ scanning rules ⟩ Defined in section 3, 22, 25, 32, 46, 53, 57, 78, 86, 96, 100, 107, 116, 126, 144, 155, 163, 169, 176, 183, 190, 198, 206, 211, 216, 222, 229, 236, 240, 245, 253, 278, 297, 304, 317, 325, and 351. Used in section 432.
⟨ set the file sizes for optional sections ⟩ Defined in section 293. Used in section 294.
⟨ `shrink.c` ⟩ Defined in section 434.
⟨ `shrink.l` ⟩ Defined in section 432.
⟨ `shrink.y` ⟩ Defined in section 433.
⟨ skip and check the start byte a ⟩ Defined in section 378. Used in sections 376, 382, 394, 397, 400, and 417.
⟨ skip function declarations ⟩ Defined in section 436. Used in section 437.
⟨ skip functions ⟩ Defined in section 375, 376, 380, 382, 394, 397, 400, and 417. Used in section 437.
⟨ skip macros ⟩ Defined in section 379, 381, 383, 384, 386, 388, 390, 392, 395, 398, 401, 403, 405, 407, 409, 411, 413, 415, 419, 421, and 423. Used in section 437.
⟨ skip the end byte z ⟩ Defined in section 377. Used in sections 376, 382, 394, 397, 400, and 417.
⟨ `skip.c` ⟩ Defined in section 437.
⟨ `stretch.c` ⟩ Defined in section 435.
⟨ symbols ⟩ Defined in section 2, 4, 24, 31, 45, 47, 52, 77, 85, 95, 99, 106, 115, 125, 134, 143, 154, 162, 168, 175, 182, 189, 197, 205, 210, 215, 221, 228, 235, 239, 244, 252, 277, 296, 303, 311, 316, 324, and 350. Used in section 433.
⟨ variables in `hformat.c` ⟩ Defined in section 358 and 363. Used in section 334.
⟨ without `-f` skip writing an existing file ⟩ Defined in section 283. Used in section 288.
⟨ without `-g` compute a local *file_name* ⟩ Defined in section 285. Used in sections 288 and 295.
⟨ write a list ⟩ Defined in section 137. Used in section 136.
⟨ write a text ⟩ Defined in section 147. Used in section 136.
⟨ write function declarations ⟩ Defined in section 426. Used in section 435.

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] 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.

[4] 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.

[5] Donald E. Knuth. *The TEX book.* Computers & Typesetting, Volume A. Addison-Wesley Publishing Company, 1984.

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

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

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

[9] John R. Levine. *flex* & *bison*. O'Reilly Media, 2009. ISBN 978-0-596-15597-1.

[10] John R. Levine Tony Mason and Doug Brown. *lex* & *yacc*. O'Reilly Media, 2012.

[11] Martin Ruckert. Computer Modern Roman fonts for ebooks. *TUGboat*, 37(3):277–280, 2016.

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

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

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

Index

I

K

L

M

T

Y

Z