ANT

A TYPESETTING SYSTEM

"ant is not TEX."

Achim Blumensath

December 16, 2007

Contents

ANT is a typesetting system inspired by TEX. Although TEX does a very good job when typesetting mathematical articles and books – the task it has been designed for – it can become very difficult, cumbersome, or even impossible to meet the typographical requirements of texts outside this narrow scope. For instance, the current draft of the new output routine for LATEX consists of more than 100 pages. Unfortunately, it is also very difficult to extend the functionality of TEX since its source code is a total mess. Even after 20 years there are only versions with minor modifications available.

For these reasons I decided to rewrite ANT from scratch aiming for a simple, clean, and modular design. In particular it is easily possible to replace parts of ANT with other implementations, say, adding an XML parser, output routines for PDF files, or a different page layout algorithm.

The current version of ANT implements all the major features of TEX but a lot of minor things are still missing. In addition, ANT provides several improvements over TEX:

- a saner macro language (no catcodes);
- a builtin high-level scripting language;
- UNICODE support;
- support for various font formats including Type1, TrueType, and OpenType;
- partial support for advanced OpenType features;
- support for colour and graphics;
- simple page layout specifications;
- river detection.

1. Invoking ant

ANT translates its input file into a DVI- or PDF-file. Rudimentary support for PostScript and SVG output is also implemented. The program is invoked as

`ant` [*options*] ⟨*input file*⟩

Currently, the following options are supported:

`--help` prints a help message.

`--format=`⟨*format*⟩ selects the output format. Supported are `dvi`, `xdvi`, `pdf` (default), `ps`, and `svg`. The latter two are only partially implemented.

`--src-specials` enables the generation of source specials.

`--debug=`⟨*flags*⟩ enables debug messages. ⟨*flags*⟩ is a combination of the following letters:

`a`	AL-commands	`l`	line breaking
`b`	AL-bytecode	`m`	macro expansion
`e`	the typesetting engine	`p`	page layout
`g`	galley breaking	`s`	various stacks
`i`	the current input		

---

2. The markup language

The markup language of ANT is quite similar to the syntax of TEX. In particular, the rules regarding tokens and braces are the same. One notable exception is that ANT stores macros as plain strings without breaking them into tokens. This solves all those issues TEX has with its catcodes. But beware that, if you define the macro

```
\definecommand\foo[m]{\bar#1}
```

then `\foo{text}` will expand to `\bartext`.

Another difference between ANT and TEX is that one can use arithmetic expressions to specify skips or dimensions:

```
\hskip{2em + (4/7)*5cm}
```

3. AL – the ant language

The requirements on a markup language for authors are quite different from those on a programming language for implementing these markup commands. For instance, the TEX macro language serves rather well as a markup language but it is quite unsuited for the implementation of packages. Besides the markup language ANT therefore also provides a scripting language called AL. Syntactically AL resembles (a subset of) the Haskell programming language. But there are two notable semantic differences: (i) evaluation in AL is strict, not lazy and (ii) AL includes a solver for linear equations and, therefore, supports variables whose value is not yet determined.

3.1. Lexical conventions

We distinguish six classes of characters according to their UNICODE category:

White space (*ws*): Mn, Mc, Me, Zs, Zl, Zp, Cc, Cf, Cs, Co, Cn
Lowercase letters (*lc*): Ll, Lm, Lo
Uppercase letters (*uc*): Lu, Lt
Digits (*dd*): Nd, Nl, No
Symbols (*sy*): Pc, Pd, Ps, Pe, Pi, Pf, Po, Sm, Sc, Sk, So
Special characters (*sp*): `" ' , ; _ ( ) [ ] { }`

Comments. A line comment starts with `;;` and extends to the end of the line, and block comments are delimited by `[;` and `;]`.

Identifiers. There are three types of identifiers: lowercase and symbolic identifiers represent variables while uppercase identifiers are used for symbols.

lid ::= *lc* (*lc* | *uc*)* *tail*?

uid ::= *uc* (*lc* | *uc*)* *tail*?

tail ::= _ (*lc* | *uc*)+ | _ *dd* + | _ *sy*+

symbol ::= *sy*+

Examples:

```
lowercase_Identifier_2  op_+

Uppercase_12_*&/_45

<<|:  **  +/-
```

The following symbols and keywords are reserved:

```
begin    local   declare_infix_left
do       match   declare_infix_non
else     then    declare_infix_right
elseif   where   declare_prefix
end      with    declare_postfix
if
=    :=    |    .    :
```

3.2. Literals

Numbers. Numerical constants can be written either using decimal notation or as fraction. Supported bases are 2, 8, 10, and 16. A sequence of digits may be interleaved with arbitrary many underscores `_`.

number ::= *decimal* | *fraction*

fraction ::= *natural* `/` *natural*

natural ::= `0b` *bin* | `0o` *oct* | *dec* | `0x` *hex*

decimal ::= `0b` *bin* [`.` *bin*] | `0o` *oct* [`.` *oct*] | *dec* [`.` *dec*] | `0x` *hex* [`.` *hex*]

bin ::= `0b` *bd* [(`_` | *bd*)* *bd*]
oct ::= `0o` *od* [(`_` | *od*)* *od*]
dec ::= *dd* [(`_` | *dd*)* *dd*]
hex ::= `0x` *hd* [(`_` | *hd*)* *hd*]

bd ::= `0` | `1`
od ::= `0` | ··· | `7`
dd ::= `0` | ··· | `9`
hd ::= *dd* | `a` | ··· | `f` | `A` | ··· | `F`

Examples:

```
3.4  3/4  0xfa.4  0o64/0b101  1_000_000
```

Strings and characters. Character constants are enclosed in apostrophes `'`, string constants are deliminated by double quotes `"`. A string constant is just an abbreviation for a list of characters. The following escape sequences are recognised:

`\'`	U0027	apostrophe	`\t`	U0009	tabulator
`\"`	U0022	double quote	`\`*ddd*	—	character with 8 bit UNICODE
`\\`	U005C	backslash			*ddd* in decimal
`\b`	U0007	bell	`\x`*hh*	—	character with 8 bit UNICODE
`\e`	U001B	ESC			*hh* in hexadecimal
`\f`	U000C	form feed	`\u`*hhhh*	—	character with 16 bit UNICODE
`\n`	U000A	newline			*hhhh* in hexadecimal
`\r`	U000D	carriage return			

Symbols. Symbols are uppercase identifiers. The symbols `True` and `False` are used as boolean values.

3.3. Expressions

Expressions consist of simple expressions combined by operators. We distinguish between binary, prefix, and postfix operators.

expr ::= *expr post-op*
| *expr bin-op expr*
| *simple-expr*$^+$
| *number simple-expr*$^+$
| `local` *lid*$^+$ `;` *expr*
| `local` *decl* `;` *expr*
| `local begin` *decl-list* `end` *expr*
| *expr* `where` *decl-list* `end`
| ...

Function application is written without parenthesis or commas:

```
foldl (+) 0 [0,1,2,3,4,5]
```

If a list of simple expressions starts with a number then the symbol `*` is inserted between the number and the following terms, i.e.,

`4 sind 45` is equivalent to `4 * sind 45`.

The `local` and `where` constructs allow local definitions of functions and variables where `local` $x_1 \ldots x_n$ is an abbreviation for

`local begin` x_1 `:= _;` ... x_n `:= _; end`

The following table lists all predefined operators in order of increasing priority:

priority	assoc.	operators	priority	assoc.	operators
0	right	`$`	7	left	`* / mod`
1	left	`>>`	8	left	`^`
2	right	`||`	9	right	`o`
3	right	`&&`	9	left	`!!`
4	non	`== <> > < >= <=`			function application
5	left	`land lor lxor lsr lsl`		prefix	prefix operators: `~`
5	right	`++`		postfix	postfix operators
6	left	`+ -`			

Simple expressions. Simple expressions are either literals, variables, or complex expressions enclosed in parenthesis. We discuss the various cases separately.

simple-expr ::= *lid* | *symbol* | *number* | *character* | *string* | `_` | ...

The symbol `_` indicates an unnamed variable without value. It is an abbreviation for the expression

```
(local x ; x)
```

simple-expr ::= ...
| *pre-op simple-expr*
| `(` *expr* `)`
| `(` *bin-op expr*`)`
| `(` *expr bin-op* `)`
| `(` *bin-op* `)`
...

Partial application of binary operators are written in parenthesis. So `(+)` denotes the function $(x, y) \mapsto x + y$ and `(+ 1)` is the function $x \mapsto x + 1$.

simple-expr ::= ...
| `(` *expr* (`,` *expr*)$^+$ `)`
| `[` *expr* (`,` *expr*)$^+$ `]`
| `[` *expr* (`,` *expr*)* `:` *expr*`]`
...

Tuples are written with parenthesis and commas, lists are set in square brackets. The tail of a list is separated by a colon `:`. Examples:

```
(1, 0, 0)     [0,1,2,3]     [x:xs]
```

The following control constructs are available:

simple-expr ::= ...
| `begin` *stmt-list-expr* `end`
| `do` *expr-stmt-list* `end`
| `if` *expr* `then` *stmt-list-expr*
(`elseif` *expr* `then` *stmt-list-expr*)*
`else` *stmt-list-expr* `end`
| `match` *expr* `with` `{` *match-body* `}`
match-body ::= *match-clause* (`|` *match-clause*)*
match-clause ::= *pattern* [`&` *expr*] `:=` *stmt-list-expr*
stmt-list-expr ::= (*stmt* `,`)* *expr*
expr-or-stmt ::= *expr* | *stmt*
expr-stmt-list ::= (*expr-or-stmt* `;`)* *expr-or-stmt*

Lambda expressions, i.e., unnamed functions, are written as list of patterns and corresponding expressions similarly to `match` constructs:

expr ::= ... | `{` *fun-body* `}`
fun-body ::= *fun-clause* (`|` *fun-clause*)*
fun-clause ::= *pattern*+ [`&` *expr*] `:=` *stmt-list-expr*

For example:

```
{ [] := 0 | [x] := x | [x, y : _] := x + y }
```

```
{ x & x >  0 := 1
| x & x == 0 := 0
| x & x <  0 := -1 }
```

3.4. Statements

A statement is an equation or an `if` statement of equations. Note that, for statements, the `else` part of an `if` statement may be omitted.

stmt ::= `expr` = `expr`
| `if` *expr* `then` *stmt*
(`elseif` *expr* `then` *stmt*)*
[`else` *stmt*] `end`

3.5. Patterns

pattern ::= `_`
| *lid*
| *number*
| `(` *pattern* `)`
| `(` *pattern* (`,` *pattern*)+ `)`
| `[` *pattern* (`,` *pattern*)+ `]`
| `[` *pattern* (`,` *pattern*)* `:` *pattern* `]`
| *lid* = *pattern*
| *pattern* = *lid*

Pattern can be used to check the structure of values and to access their components. For instance, the pattern `(0, x)` can be matched with a pair whose first component is the number 0 and whose second component will be bound to the variable `x`.

3.6. Declarations

decl ::= *lid pattern** [`&` *expr*] `:=` *stmt-list-expr*
| *pattern bin-op pattern* [`&` *expr*] `:=`
stmt-list-expr
| *lid*$^+$
| `declare_infix_left` *num lid*$^+$
| `declare_infix_non` *num lid*$^+$
| `declare_infix_right` *num lid*$^+$
| `declare_prefix` *lid*$^+$
| `declare_postfix` *lid*$^+$

The first two cases are used to declare functions. A list of identifiers `x_0` ... `x_n` is an abbreviation for the declarations

`x_0 := _; ... x_n := _.`

The `declare_`... statements can be used to declare the priority and associativity of operators.

3.7. Built in AL-commands

Control constructs. The function

`error` *msg*

can be used to abort the computation with a given error message.

Types. To test the type of a value the following functions can be used:

`is_unbound` *x*	`is_symbol` *x*
`is_bool` *x*	`is_function` *x*
`is_number` *x*	`is_list` *x*
`is_char` *x*	`is_tuple` *x*

Logical operations. The operators for disjunction, conjunction, and negation are

`||  &&  not`

Comparison operators. The operators

`==  <>  >  <  >=  <=`

compare their arguments without modifying them. Equality `==` and inequal-

ity <> are defined for all types. The other relations for numbers, characters, lists, and tuples where the latter ones are ordered lexicographically.

`min` *x y* `max` *x y*

compute, respectively, the minimum and maximum of *x* and *y*.

General arithmetic. The usual arithmetic operations

`+ - * / ^ quot mod ~ abs`

are supported. Addition + is defined for numbers, tuples, and lists:

$num_0 + num_1 \Rightarrow num_0 + num_1$

$(x_0, \ldots, x_n) + (y_0, \ldots, y_n) \Rightarrow (x_0 + y_0, \ldots, x_n + y_n)$

$[x_0, \ldots, x_n] + [y_0, \ldots, y_m] \Rightarrow [x_0, \ldots, x_n, y_0, \ldots, y_m]$

Subtraction - and the unary minus ~ can be used for numbers and tuples. Multiplication * and division / are defined for the following types:

$num_0 * num_1 \Rightarrow num_0 \cdot num_1$

$num * (x_0, \ldots, x_n) \Rightarrow (num * x_0, \ldots, num * x_n)$

$(x_0, \ldots, x_n) * num \Rightarrow (x_0 * num, \ldots, x_n * num)$

$(x_0, \ldots, x_n) * (y_0, \ldots, y_n) \Rightarrow x_0 * y_0 + \cdots + x_n * y_n$

$t * [x_0, \ldots, x_n] \Rightarrow \sum_{k=0}^{n} \binom{n}{k} (1-t)^{n-k} t^k x_k$

$num_0 / num_1 \Rightarrow num_0 / num_1$

$(x_0, \ldots, x_n) / num \Rightarrow (x_0 / num, \ldots, x_n / num)$

`abs` *x* evaluates to $|x|$ if *x* is a number and to

$\sqrt{x_0^2 + \cdots + x_{n-1}^2}$ if $x = (x_0, \ldots, x_{n-1})$.

Integer arithmetic. For integers the following additional functions are defined. There are four functions to round numbers:

`round` *x*	in case of ties off zero
`truncate` *x*	in case of ties towards zero
`ceiling` *x*	up
`floor` *x*	down

The bitwise logical operations can be applied only to integers.

`land` x y	bitwise and
`lor` x y	bitwise or
`lxor` x y	bitwise exclusive or
`lneg` x	bitwise negation
`lsr` x y	bitwise shift right
`lsl` x y	bitwise shift left

Real arithmetic. In contrast to other functions on numbers, the following ones are of limited precision.

`pi`	the constant π
`sqrt` x	square root
`exp` x	e^x
`log` x	natural logarithm
`sin` x	sine of x in radians
`cos` x	cosine of x in radians
`tan` x	tangent of x in radians
`arcsin` x	arcsine of x in radians
`arccos` x	arccosine of x in radians
`arctan` x	arctangent of x in radians
`sind` x	sine of x in degree
`cosd` x	cosine of x in degree
`tand` x	tangent of x in degree
`arcsind` x	arcsine of x in degree
`arccosd` x	arccosine of x in degree
`arctand` x	arctangent of x in degree
`sinh` x	hyperbolic sine of x
`cosh` x	hyperbolic cosine of x
`tanh` x	hyperbolic tangent of x
`arcsinh` x	hyperbolic arcsine of x
`arccosh` x	hyperbolic arccosine of x
`arctanh` x	hyperbolic arctangent of x

Lists and tuples. Lists and tuples are treated like functions mapping indices to values. That is, to access the nth element of a list or tuple one can apply the list to the integer n. Note that indices start at 0.

The length of a list or tuple can be obtained by the function

`length` x.

If the argument is of some other type `length` returns 1. The function

`to_string` *x*

converts its argument into a string. A more general printf-like alternative is given by the function

`format_string` *format* $arg_1 \ldots arg_n$

It supports the following conversion specifications:

`s`	string
`d`	decimal numeral
`x`	lower case hexadecimal numeral
`X`	upper case hexadecimal numeral
`r`	lower case roman numeral
`R`	upper case roman numeral
`a`	lower case alphabetic numeral
`A`	upper case alphabetic numeral

The functions

`to_list` *x*
`to_tuple` *x*

can be used to convert between tuples and lists.

`dir` *angle* `angle` *vec*

`dir` *angle* returns the unit vector in the given direction and `angle` (x, y) returns the angle (in degrees) of the given vector.

Dictionaries. A dictionary is a finite mapping from symbols to arbitrary values. For instance,

```
local d := { Foo := 1 | Bar := 2 };
d Foo
```

yields 1. In order to add a new entry to a dictionary or to modify an existing one there exists the command

`add_to_dict` *symbol value dictionary*

It returns the new dictionary.

Characters. To test the category of a character the following functions can be used:

```
char_is_letter c          char_is_symbol c
char_is_mark c            char_is_separator c
char_is_number c          char_is_control c
char_is_punct c           char_is_space c
```

`char_is_space` is a short hand for the disjunction of `char_is_separator` and `char_is_control`. The UNICODE category of a character can be looked up with the function

```
char_category c
```

It returns one of the following symbols

```
Lu  Ll  Lt  Lm  Lo
Mn  Mc  Me
Nd  Nl  No
Pc  Pd  Ps  Pe  Pi  Pf  Po
Sm  Sc  Sk  So
Zs  Zl  Zp
Cc  Cf  Cs  Co  Cn
```

The name of a character can be obtained by

```
char_name c
```

To convert a character to uppercase, lowercase, or titlecase one can use

```
to_upper c
to_lower c
to_title c
```

Symbols. To convert a string into a symbol one can use the function

```
to_symbol str.
```

The function

```
generate_symbol x
```

creates a new unique symbol without textual representation. Its argument *x* is ignored.

File operations. The command

```
serialise file name value
```

can be used to write an AL-value into a file. The return value is either `True` or `False` depending on whether the operation was successful. Note that serialisa-

tion of functions is not supported. Any functions in *value* will be replaced by an 'unknown' value. To read the value from a file you can use the command

`unserialise` *file name* .

4. Typesetting commands

This section contains a description of all typesetting commands. The commands are grouped by topic and each section contains both markup commands and AL-commands.

4.1. Program control

To call AL-commands within your document the following commands are provided:

`\beginALdeclarations` ⟨*code*⟩ `\endALdeclarations`
`\ALmacro` ⟨*expr*⟩
`\ALcommand` ⟨*expr*⟩

A list of AL-declarations can be entered by surrounding them with `\beginALdeclarations` and `\endALdeclarations`. The command `\ALmacro` evaluates a given AL-expression and inserts the result – which should be a string – at the current position into the input. `\ALcommand` is a more powerful version of `\ALmacro`. Its argument is an AL-expression which should yield a function of type

parse-state → parse-state.

This function is invoked with the current parse-state.

Example:

```
\beginALdeclarations
mirror :=
    local arg;
    do
        ps_arg_expanded arg;
        ps_insert_string (reverse arg);
    end
\endALdeclarations
\definecommand\mirror{\ALcommand{mirror}}
```

defines a command `\mirror` that reverses its argument.

The command

`\relax`

does nothing.

`\beginliteral` ⟨*string*⟩ `\endliteral`

converts the string into glyphs without interpreting it.

`\include` ⟨*file name*⟩

reads the given file. The command

`\jobname`

expands to the basename of the input file. To output error messages you can use the following AL-commands:

`ps_warning` *message*
`ps_error` *message*

4.2. Page layout

ANT has a much more sophisticated algorithm for page layout than LaTeX. Every page is divided into several areas that can be filled with contents independently. The command

`\newpagelayout` ⟨*name*⟩ ⟨*page-width*⟩ ⟨*page-height*⟩

defines a new page layout with the given name and dimensions. All subsequent `\newpagearea` commands affect this layout.

`\newpagearea` ⟨*x*⟩ ⟨*y*⟩ ⟨*width*⟩ ⟨*height*⟩ ⟨*max-top*⟩ ⟨*max-bot*⟩ ⟨*type*⟩ ⟨*parameters*⟩

adds a new area with the given dimensions to the page. The areas of a page are filled in order with content. When areas overlap only that part of the current area is considered that is not already filled with material from another area. Currently, there are four different area types.

If ⟨*type*⟩ is `galley` then the contents is taken from the galley specified by the dictionary ⟨*parameters*⟩.

`name` (string) name of the galley.

`top-skip` (skip, default 1em) minimal whitespace above the text.

`bottom-skip` (skip, default 1em) minimal whitespace below the text.

`min-size` (skip, default 5em) minimal height. If there is less space left the area remains empty.

`grid-size` (skip, default 1em) If non-zero all baseline positions are rounded to a multiple of this value.

If ⟨*type*⟩ is `direct` then ⟨*parameters*⟩ contains ANT code for the contents of the area. When this code is evaluated it can access the number of the current page via the counter `page`. The marks from previous pages are given as global AL-variables where the name is prefixed with `OldMark`, i.e., a mark named `Foo` can be accessed by the command

```
local x;
do
  ps_get_global x OldMarkFoo;
  ps_insert_string x;
end
```

Similarly, marks found in the current page get the prefix `NewMark`.

If ⟨*type*⟩ is `float` then the area is used to display floats. The dictionary ⟨*parameters*⟩ contains the following entries:

`alignment` (default `top`) either `top` or `bottom`.

`top-skip` (skip, default 1em) minimal whitespace above the first float.

`bottom-skip` (skip, default 1em) minimal whitespace below the last float.

`float-sep` (dimension, default 1em) whitespace between floats.

Finally, ⟨*type*⟩ can be `footnote` in which case the area is used to display footnotes. The dictionary ⟨*parameters*⟩ contains the following entries:

`separator` (ANT code, default empty) code to typeset the separator above the footnote area.

`top-skip` (skip, default 1em) minimal whitespace above the first footnote.

`bottom-skip` (skip, default 1em) minimal whitespace below the last footnote.

`grid-size` (skip, default 1em) If non-zero all baseline positions are rounded to a multiple of this value.

`line-params` line dictionary.

`par-params` paragraph dictionary.

`line-break-params` line-break dictionary.

`hyphen-params` hyphenation dictionary.

To build a sequence of pages one uses the command

`\shipoutpages` [*number-of-pages*] ⟨*even-layout*⟩ ⟨*odd-layout*⟩

⟨*number-of-pages*⟩ specifies the number of pages to output. If it is zero then ANT creates pages until all the galleys are empty. The other arguments indicate the names of the page layouts used for even and odd numbered pages, respectively.

A float can be inserted by the following commands:

`\floatbox` ⟨*body*⟩
`\floatpar` ⟨*body*⟩
`\floatgalley` ⟨*body*⟩

The difference between them lies in the mode the ⟨*body*⟩ is typeset in. `\floatbox` uses vertical mode, `\floatpar` horizontal mode, and `\floatgalley` paragraph mode.

`\nextpagelayout` ⟨*layout*⟩

changes the layout of the following page. Note that

```
\nextpagelayout{foo}
```

takes effect only in vertical mode. Otherwise, one has to put the command inside a `\vadjust` command:

```
\vadjust{\nextpagelayout{foo}}
```

The corresponding AL-commands are

`ps_shipout_pages` *number even odd*
`ps_new_page_layout` *name width height*
`ps_new_area` *name pos-x pos-y width height max-top max-bot type param*

The file `page-layout.ant` contains two predefined page layouts and a helper function returning the dimensions of common paper formats.

`get_page_size` *format*

returns a pair consisting of the dimensions of the paper format. Supported formats are

```
A_3  A_4  A_5  A_6
B_3  B_4  B_5  B_6
Letter  Legal  Executive
```

The predefined page layouts can be used with the routines

`simple_page_layout` *page-size division baseline*
`two_column_page_layout` *page-size division baseline*

The first one creates a galley named `main` and two page layouts `left` and `right`, each consisting of a single text block with headers and footers. The second one does the same, except that the text block consists of two columns. The parameter *page-size* contains the paper size, *division* determines the margins (a good value is 9), and `baseline` is the font height (including leading).

4.3. Galleys and paragraphs

The layout process of ANT consists of two steps. In the first one, a set of galleys is constructed from the given paragraphs. Such a galley is a continuous run of text of a fixed width but of unlimited length. In the second step, parts of these galleys are used to assemble the actual pages.

To create a new galley of width ⟨*measure*⟩ one uses the command

`\newgalley` ⟨*name*⟩ ⟨*measure*⟩

The corresponding AL-command is

`ps_new_galley` *name measure*

All material between the commands

`\begingalley` ⟨*name*⟩
`\endgalley`

is appended to the galley ⟨*name*⟩.

As in TEX the end of a paragraph is marked by either one of the following commands:

```
\endgraf
\par
```

An empty line is automatically translated to the command sequence `\par`.

4.4. Boxes

`\char` ⟨*number*⟩
`\glyph` ⟨*number*⟩
`\mathchar` ⟨*math-code*⟩ ⟨*small-font*⟩ ⟨*small-char*⟩ ⟨*large-font*⟩ ⟨*large-char*⟩

Prints a single glyph. `\char` expects a UNICODE number and `\glyph` the index of the glyph. The arguments of `\mathchar` consists of the math-code, the font family and the character. The first pair specifies the normal version, the second one is used by scalable delimiters. Supported math-codes are:

```
letter          operator        inner
ordinary        punct           subscript
binop           open            superscript
relation        close
```

For `letter`, the character is given by a UNICODE number while in the other cases a glyph index is expected.

`\penalty` ⟨*number*⟩

inserts a break point with the given penalty.

`\discretionary` * [*penalty*] ⟨*pre-break*⟩ ⟨*post-break*⟩ ⟨*no-break*⟩

inserts a break point with the given parameters. The * indicates a break caused by hyphenation. The default for ⟨*penalty*⟩ is 0, if * is omitted, and `hyphen-penalty` otherwise.

`\hskip` ⟨*skip*⟩
`\vskip` ⟨*skip*⟩
`\kern` ⟨*skip*⟩

insert horizontal and vertical glue.

`\ensurevskip` ⟨*skip*⟩

determines the amount of vertical glue at the end of the current galley and increases it to ⟨*skip*⟩ if necessary.

`\hbox` {⟨*body*⟩}
`\hbox to` ⟨*width*⟩ {⟨*body*⟩}
`\hbox spread` ⟨*amount*⟩ {⟨*body*⟩}

create a horizontal box around ⟨*body*⟩.

`\vbox` {⟨*body*⟩}

`\vbox to` ⟨*width*⟩ {⟨*body*⟩}
`\vbox spread` ⟨*amount*⟩ {⟨*body*⟩}

create a vertical box around ⟨*body*⟩.

`\phantom` ⟨*body*⟩
`\hphantom` ⟨*body*⟩
`\vphantom` ⟨*body*⟩

create an empty box of the same width and/or height as that of ⟨*body*⟩.

`\hleaders` ⟨*width*⟩ ⟨*body*⟩

creates a box of width ⟨*width*⟩ that is filled with copies of ⟨*body*⟩.

`\vadjust *` ⟨*body*⟩

adds ⟨*body*⟩ below the line containing the `\vadjust` command. If `*` is present the material will be inserted above the line.

`\rule` ⟨*width*⟩ ⟨*height*⟩ ⟨*depth*⟩

creates a rule of the given dimensions.

`\image` [*options*] ⟨*file name*⟩

inserts the given image. The ⟨*options*⟩ dictionary may contain the following options:

`width`	(skip)	the width of the image.
`height`	(skip)	the height of the image.
`dpi`	(number)	the resolution of the image.

DVI specials can be created with the AL-command

`ps_dvi_special` *string*

4.5. Parameters

The parameters governing the typesetting process are grouped into several dictionaries. Each of these dictionaries can be modified by the command

`\setparameter` ⟨*parameter*⟩ ⟨*dictionary*⟩

⟨*parameter*⟩ is the name of the dictionary and ⟨*dictionary*⟩ its new value. A dictionary consists of entries of the form

⟨*key*⟩ [= ⟨*value*⟩]

separated by semicolons or commas.

To modify parameters locally one can surround the corresponding section by the commands

```
\begingroup
\endgroup
```

Most parameter dictionaries come in two versions: those with the prefix `this-` refer only to the following paragraph while those without effect all paragraphs. Currently the following parameter dictionaries are defined:

```
font
paragraph       this-paragraph
line            this-line
line-break      this-line-break
hyphenation     this-hyphenation
space           this-space
math            this-math
```

The `font` dictionary contains the following entries:

`family` (string) font family.

`series` (string) font series.

`shape` (string) font shape.

`size` (number) font size.

The `paragraph` and `this-paragraph` dictionaries contain the following entries:

`measure` (skip) the line width.

`par-indent` (dimension) the indent of the first line.

`par-fill-skip` (dimension) the whitespace at the end of the last line.

`left-skip` (dimension) the left margin.

`right-skip` (dimension) the right margin.

`left-par-shape` specifies the left indentation of each line. Its value is a comma-separated list of entries of the form:

⟨*range*⟩ : ⟨*indent*⟩

`right-par-shape` similar to `left-par-shape` but for the right side.

`par-skip` (dimension) the whitespace between paragraphs.

`left-annotation` (ANT code) This value specifies material that is added to the left of every line (useful, e.g., for line numbering, adding a vertical bar, etc.). The code should evaluate to a box of width zero.

`right-annotation` (ANT code) This value specifies material that is added to the right of every line. The code should evaluate to a box of width zero.

`post-process-line` (not implemented) code to annotate the lines of the paragraph.

The `line` and `this-line` dictionaries contain the following entries:

`baseline-skip` (dimension) the distance between one baseline and the next.

`line-skip-limit` (skip) the minimal distance between lines.

`line-skip` (dimension) If the current value of `baseline-skip` leads to less than `line-skip-limit` space between two lines then this space is set to `line-skip`.

`leading` (string) The method to determine the amount of space between lines (see below).

`club-penalty` (number) The penalty for breaking after the first line of a paragraph.

`widow-penalty` (number) The penalty for breaking before the last line of a paragraph.

Currently, there are four leading methods implemented:

`fixed` The distance between baselines is always `baseline-skip`.

`register` The distance between baselines is always a multiple of `baseline-skip`.

`TeX` This is the TEX method based on `baseline-skip`, `line-skip-limit`, and `line-skip`.

`skyline` The TEX method but the shape of the lines is taken into account when calculating their minimal distance.

The `line-break` and `this-line-break` dictionaries contain the following entries:

`pre-tolerance` (number)

`tolerance` (number)

`looseness` (integer) The line-breaking algorithms returns a paragraph that has `looseness` more lines than the optimal solution.

`line-penalty` (number) penalty for the number of lines.

`adj-demerits` (number) demerits for two consecutive lines with different spacing.

`double-hyphen-demerits` (number) the demerits for two consecutive lines ending in a hyphen.

`final-hyphen-demerits` (number) the demerits for the second but last line ending in a hyphen.

`emergency-stretch` (dimension) additional stretchability for each line, for the case that no acceptable solution exists.

`simple-breaking` (bool) when true ANT uses a faster line-breaking algorithm that yields slightly worse results. (It does not support breaking of ligatures and river detection.)

`river-demerits` (number) the demerits for a river.

`river-threshold` (skip) minimal amount whitespace has to overlap to count as a river.

The `hyphenation` and `this-hyphenation` dictionaries contain the following entries:

`hyphen-table` (string) The name of the hyphenation table.

`hyphen-penalty` (number) The penalty for breaking words.

`ex-hyphen-penalty` (number) The penalty for consecutive hyphenated lines.

`left-hyphen-min` (integer) The minimal number of letters before a word break.

`right-hyphen-min` (integer) The minimal number of letters after a word break.

`script-lang` (string) The name of the current script and language systems. These names are font specific.

The `space` and `this-space` dictionaries contain the following entries:

`space-factor` (number)

`space-skip` (dimension)

`xspace-skip` (dimension)

`victorian-spacing` (boolean) When true ANT increases the spacing after punctuation.

The `math` and `this-math` dictionaries contain the following entries:

`thin-math-skip` (dimension)

`med-math-skip` (dimension)

`thick-math-skip` (dimension)

`script-space` (dimension)

`rel-penalty` (number)

`binop-penalty` (number)

`delimiter-factor` (number)

`delimiter-shortfall` (skip)

`null-delimiter-space` (dimension)

4.6. Fonts

The text font is changed with the command

`\setparameter{font}` ⟨*font-specification*⟩

(see below). To change the math fonts one can use the command

`\setmathfont` ⟨*font-specification*⟩

where ⟨*font-specification*⟩ is a dictionary containing the entries

`math-family` (integer) the number of the math-family to change. If this key is omitted all families are changed.

`family` (string) the font family.

`series` (string) the font series.

`shape` (string) the font shape.

`text-size` (number) the text size.

`script-size` (number) the script size.

`script-script-size` (number) the double script size.

The macros

```
\FontFamilyRoman
\FontFamilySans
\FontFamilyTypewriter
\FontFamilyMath
\FontFamilyExtensions
\FontFamilySymbols
```

contain the default families used by the font commands below.

```
\FontSeriesMedium
\FontSeriesBold
```

contain the default series used by the font commands below.

```
\FontShapeUpright
\FontShapeItalic
\FontShapeSlanted
\FontShapeSmallCaps
```

contain the default shapes used by the font commands below.

```
\FontSizeTiny
\FontSizeScript
\FontSizeFootnote
\FontSizeSmall
\FontSizeNormal
\FontSizeLargeI
\FontSizeLargeII
\FontSizeLargeIII
\FontSizeHugeI
\FontSizeHugeII
```

contain the default sizes used by the font commands below.

```
\rmfamily
\sffamily
\ttfamily
```

change the font family.

```
\mdseries
\bfseries
```

change the font series.

```
\upshape
\itshape
\slshape
\scshape
```

change the font shape.

```
\tiny
\scriptsize
\footnotesize
\small
\normalsize
\large
\Large
\LARGE
\huge
\Huge
```

change the font size.

```
\normalfont
```

restores the normal font.

To make fonts available to ANT you have to declare them. The AL-command

`ps_declare_font` *font-file family series shape sizes parameters*

tells ANT that the file *font-file* contains a font in the given *family*. The *parameters* take the form of a dictionary containing the following entries. Each of them is optional.

`Encoding`	the encoding vector of the font.
`HyphenGlyph`	the index of the hyphen glyph.
`SkewGlyph`	the index of the skew glyph.
`Scale`	an optional scaling factor for the font.
`LetterSpacing`	amount of additional letter spacing.
`Adjustments`	additional kerning and ligature commands.

`AutoLigatures` boolean to enable automatic creation of ligatures.

`BorderKern` list of tuples containing kerning values for margin kerning.

Example:

```
local ot_1 := ("\u0393", ... "\u00a8");
do
  ps_declare_font "cmti10.tfm" "Computer Modern Roman"
    "medium" "italic" (10,12) { Encoding := ot_1 };
end
```

To define mathematical symbols one can use the following commands.

`ps_define_math_symbol` *name math-code font glyph*
`ps_define_root_symbol` *name small-font small-glyph large-font large-glyph*
`ps_define_math_accent` *name font glyph*
`ps_set_math_code` *char math-code small-font small-glyph large-font large-glyph*

4.7. Tables

The commands

```
\begintable
\endtable
\newtableentry
\newtablerow
```

can be used to typeset a table. The entries of a row are separated by `\newtableentry` commands, and the rows by `\newtablerow` commands. The position of a table entry is stored in five counters:

`table-entry:left` the first column

`table-entry:right` the last column

`table-entry:top` the first row

`table-entry:baseline` the row of the baseline of the entry

`table-entry:bottom` the last row

These counters can be modified to create entries spanning several columns or rows.

4.8. Colour and graphics

You can change the colour with the commands

`\setgreycolour` ⟨*grey*⟩
`\setrgbcolour` ⟨*red*⟩ ⟨*green*⟩ ⟨*blue*⟩
`\setcmykcolour` ⟨*cyan*⟩ ⟨*magenta*⟩ ⟨*yellow*⟩ ⟨*black*⟩

They take effect until the end of the current box. The corresponding AL-commands are

`ps_set_colour` *colour*
`ps_set_bg_colour` *colour*
`ps_set_alpha` *alpha*

Colours can be specified in one of three formats:

`(Grey,` *x*`)`
`(RGB,` *red*`,` *green*`,` *blue*`)`
`(CMYK,` *cyan*`,` *magenta*`,` *yellow*`,` *black*`)`

The following AL-commands can be used to draw lines or filled shapes.

`ps_stroke` *path*
`ps_fill` *path*
`ps_clip` *path*
`ps_set_line_width` *width*
`ps_set_line_cap` *line-cap*
`ps_set_line_join` *line-join*
`ps_set_miter_limit` *limit*

To construct paths ANT provides the following AL-commands:

`make_path` *point*
`close_path` *cycle path*
`path_add_point` *point path*
`path_add_in_dir` *vector path*
`path_add_in_angle` *angle path*
`path_add_in_curl` *curl path*
`path_add_in_tension` *tension path*
`path_add_out_dir` *vector path*
`path_add_out_angle` *angle path*
`path_add_out_curl` *curl path*
`path_add_out_tension` *tension path*

`path_add_control_points` *point point path*

You start a path with `make_path` at the given point. You can add new points with `path_add_point`. For every point you can specify the tangent of the incoming and the outgoing curve with the remaining commands. For instance, you can draw a circle with radius 10 pt by

```
\vbox to 20pt{\vss\hbox to 20pt{%
\ALcommand{
  local begin
    u := 10pt;
    circle :=
      do
        path_add_point (u,2u);
        path_add_point (0,u);
        path_add_point (u,0);
        close_path True;
      end
      (make_path (2u,u));
  end
  do
    ps_set_line_width 0.6pt;
    ps_stroke circle
  end
}}}
```

4.9. Mathematics

$ ⟨*math*⟩ $
\beginmath
\endmath
\begintext
\endtext
_ ⟨*subscript*⟩
^ ⟨*superscript*⟩

The usual math commands. `\begintext` and `\endtext` can be used to enter text-mode when in math-mode. Note that both `\beginmath` and `\endmath`, and `\begintext` and `\endtext` nest.

`\frac` ⟨*numerator*⟩ ⟨*denominator*⟩

`\genfrac` ⟨*left*⟩ ⟨*right*⟩ ⟨*thickness*⟩ ⟨*numerator*⟩ ⟨*denominator*⟩
`\sqrt` ⟨*body*⟩

create a fraction and a root.

`\overline` ⟨*body*⟩
`\underline` ⟨*body*⟩

put a line atop or below ⟨*body*⟩.

`\left` ⟨*delimiter*⟩ ⟨*body*⟩ `\middle` ⟨*body*⟩ `\right` ⟨*delimiter*⟩

adjusts the height of the delimiters to that of the ⟨*body*⟩.

`\displaystyle`
`\textstyle`
`\scriptstyle`
`\scriptscriptstyle`

selects the math mode.

`\mathord` ⟨*body*⟩
`\mathop` ⟨*body*⟩
`\mathbin` ⟨*body*⟩
`\mathrel` ⟨*body*⟩
`\mathopen` ⟨*body*⟩
`\mathclose` ⟨*body*⟩
`\mathpunct` ⟨*body*⟩
`\mathinner` ⟨*body*⟩

sets the math-code of the ⟨*body*⟩.

`\indexposition` ⟨*pos*⟩
`\limits`
`\nolimits`

determines where the following sub- and superscripts are placed. ⟨*pos*⟩ can take the values `left`, `right`, and `vert`. `\limits` and `\nolimits` are shorthands for `\indexposition{vert}` and `\indexposition{right}`, respectively. For example, the command

```
\prod\indexposition{left}^a_b \indexposition{vert}^c_d
    \indexposition{right}^e_f
```

produces the output

$$\prod_{d}^{c} {}_{b}^{a}\!\!\prod_{f}^{e}$$

In addition, all the usual mathematical symbols are defined: `\alpha`, `\sim`,...

4.10. Macros and environments

`\definecommand` ⟨*name*⟩ [*arguments*] ⟨*body*⟩
`\definepattern` ⟨*name*⟩ [*arguments*] ⟨*body*⟩

define a new command. For `\definecommand` ⟨*name*⟩ has to be a command sequence in the sense of TEX, while in the case of `\definepattern` it can be any sequence of symbols. Furthermore, expanding a command works the same way as in TEX. The next symbol after the command cannot be a letter and the following white space is deleted. For patterns, these restrictions do not hold. The parameter ⟨*arguments*⟩ consists of a list of letters specifying the type of the arguments:

`m` mandatory argument

`s` optional `*`

`o` optional argument with empty default

`O{`⟨*default*⟩`}` optional argument with default value

`\savecommand` ⟨*name*⟩
`\restorecommand` ⟨*name*⟩
`\savepattern` ⟨*name*⟩
`\restorepattern` ⟨*name*⟩

These commands can be used to define commands and patterns locally:

```
\definecommand{\foo}{old}
\foo
\savecommand\foo
\definecommand{\foo}{new}
\foo
\restorecommand\foo
\foo
```

produces the output `old new old`.

`\defineenvironment` ⟨*name*⟩ [*arguments*] ⟨*begin-body*⟩ ⟨*end-body*⟩

creates a new environment. Note that the arguments can be used in both bodies.

`\begin` ⟨*name*⟩
`\end` ⟨*name*⟩

starts and ends an environment.

The corresponding AL-commands are:

`ps_set_default_char_cmd` *execute expand*
`ps_define_command` *name execute expand*
`ps_define_pattern` *pattern execute expand*
`ps_save_command` *name*
`ps_restore_command` *name*
`ps_save_pattern` *pattern*
`ps_restore_pattern` *pattern*
`ps_lookup_command` *result name*
`ps_push_env` *name arguments*
`ps_pop_env` *arguments name*
`ps_set_env_args` *arguments*
`ps_top_env` *name arguments*
`ps_lookup_env` *result name*
`ps_define_env` *name execute-begin expand-begin execute-end expand-end*

4.11. Counters and references

ANT has built in counters that can be used to number sections, theorems, etc. The following markup commands are provided:

`\newcounter` [*super-counter*] ⟨*name*⟩
`\setcounter` ⟨*name*⟩ ⟨*value*⟩
`\addtocounter` ⟨*name*⟩ ⟨*value*⟩
`\getcounter` ⟨*format*⟩ ⟨*name*⟩

If ⟨*super-counter*⟩ is given the new counter is reset every time the value of the super-counter changes. The format can be one of the following letters:

`1`	arabic number
`a`	lowercase alphabetic letter
`A`	uppercase alphabetic letter
`i`	lowercase roman number
`I`	uppercase roman number
`r`⟨*text*⟩	repeats ⟨*text*⟩ *i* times
`s`⟨*text 1*⟩…⟨*text n*⟩	returns ⟨*text i*⟩

Some counters are predefined:

`year` the year

`month` the month (1 to 12)

`day` the day (1 to 31)

`day-of-week` day of the week (0 means Sunday)

The equivalent AL-commands are:

`ps_new_counter` *name value super*
`ps_get_counter` *value name*
`ps_set_counter` *name value*

In addition to counters there are also global variables that can be accessed only via AL-commands.

`ps_get_global` *result name*
`ps_set_global` *name value*

These can be used to hold AL-values that are globally needed. Each global variable is referenced by a symbol. Example:

```
ps_set_global Counter 17
...
local x;
ps_get_global x Counter
```

The file `references.ant` provides an implementation of references on top of these global variables. Furthermore, it contains commands to preserve the value of global variables across runs of ANT.

To declare that a global variable should be preserved in this way you can use the command

`ps_declare_persistent_global` *name*

References can be created with the AL-command

`ps_add_reference` *name value*

Its value is retrieved by

`ps_lookup_reference` *result name*

The corresponding markup commands are

`\addreference` ⟨*name*⟩ ⟨*value*⟩
`\lookupreference` ⟨*name*⟩

The file `references.ant` also provides the following two commands

`\currentpage`
`\saveposition` ⟨*command*⟩

The first command expands to the number of the current page, the second one defines a new macro ⟨*command*⟩ that expands to this number. (You need two runs of ANT until these values are available.) These commands are based on the AL-commands

`ps_get_current_page` *page*
`ps_get_current_position` *page*
`ps_get_current_line` *line*

The first command stores the number of the current page in *page*. The second one stores a triple consisting of the current page number and the current coordinates. The last command returns the number of the current line.

4.12. Parsing

To read the next argument one can use the following AL-commands:

`ps_next_char` *char*
`ps_get_char` *char pos*
`ps_remove_chars` *num*
`ps_insert_string` *str*
`ps_location` *loc*
`ps_arg_expanded` *arg*
`ps_arg_execute` *arg mode*
`ps_arg_num` *arg*
`ps_arg_int` *arg*
`ps_arg_skip` *arg*
`ps_arg_dim` *arg*
`ps_arg_key_val` *arg*
`ps_opt_expanded` *arg default*
`ps_opt_key_val_int` *arg*
`ps_opt_int` *arg default*
`ps_arg_TeX_dim` *arg*

The following commands run the parser on various inputs:

`ps_execute_next_char` *finished*
`ps_execute_stream` *string*

`ps_execute_argument`
`ps_run_parser` *result mode*

4.13. Nodes

The following commands provide low-level access to the interface of the typesetting engine.

`ps_current_mode` *mode*
`ps_open_node_list` *mode*
`ps_close_node_list` *nodes mode*
`ps_add_node` *node*

4.14. Environment commands

Most of the functions below return environment commands, i.e., functions of type *location* → *environment* → *environment*.

Font parameters:

`em` *env*
`ex` *env*
`mu` *env*

Galleys:

`new_galley` *name measure*
`select_galley` *name*

Galley parameters:

`set_par_params` *params*
`set_line_params` *params*
`set_line_break_params` *params*
`set_hyphen_params` *params*
`set_space_params` *params*
`set_math_params` *params*
`set_current_par_params` *params*
`set_current_line_params` *params*
`set_current_line_break_params` *params*
`set_current_hyphen_params` *params*
`set_current_space_params` *params*

`set_current_math_params` *params*
`set_par_shape` *shape*
`set_colour` *colour*

Page layout:

`new_page_layout` *name page-width page-height*
`select_page_layout` *name*

Fonts:

`set_math_font` *definition*
`adapt_fonts_to_math_style`

Space factor:

`get_space_factor` *env char*
`adjust_space_factor` *char*

4.15. Dimensions

The following constants are defined:

`pt` = 1	`pc` = 12 pt
`in` = 72.27 pt	`sp` = 1/65536 pt
`bp` = 1/72 in	`dd` = 1238/1157 pt
`cm` = 1/2.54 in	`cc` = 12 dd
`mm` = 0.1 cm	

These are postfix operators, i.e., you can write `10pt`, `2cm`, etc.

A dimension consists of a base value together with two values that specify how much it can be stretched and shrunk.

`make_dim` *base stretch stretch-order shrink shrink-order*
`fixed_dim` *base*
`dim_zero`
`dim_1pt`
`dim_12pt`
`dim_fil`
`dim_fill`
`dim_ss`
`dim_filneg`
`dim_equal` *dim dim*
`dim_add` *dim dim*

`dim_neg` *dim*
`dim_sub` *dim dim*
`dim_mult` *num dim*
`dim_max` *dim dim*
`dim_min` *dim dim*
`dim_max_stretch` *dim*
`dim_max_shrink` *dim*
`dim_max_value` *dim*
`dim_min_value` *dim*
`dim_shift_base` *dim delta*
`dim_shift_base_upto` *dim delta*
`dim_inc_upto` *dim delta*
`dim_dec_upto` *dim delta*
`dim_resize_upto` *dim delta*
`adjustment_ratio` *dim size*
`dim_scale_badness` *ratio*
`dim_scale` *dim ratio*
`dim_scale_upto` *dim ratio*

5. Overview over the source code

ANT consist of five parts whose detailed descriptions follow in the sections below.

(1) The *runtime library* contains IO-routines, functions to load fonts, and so on.
(2) The *typesetting library* consists of the actual layout routines.
(3) The *layout engine* is an interpreter for a simple typesetting language.
(4) The *parser* translates the markup language into this internal language.
(5) Finally, there is a *virtual machine* for the scripting language.

5.1. The runtime library

The runtime library contains four groups of modules. There are modules defining datatypes and algorithms.

Bitmap datatype for bitmaps.

Dim implementation of types for dimensions.

DynamicTrie implementation of generic tries.

DynUCTrie implementation of UNICODE tries.
Trie implementation of packed tries.
PTable datatype for tables with a current element.
SymbolSet simple list-based type to store sets of symbols.
Hyphenation implementation of hyphenation tries.
Substitute routines for pattern matching and substitution.
JustHyph routines for justification and hyphenation.
There are modules for font handling.
FontMetric datatype for font metrics.
Encodings encoding tables for OT1, OTT, OML, and OMS.
GlyphMetric datatype for glyph metrics.
GlyphBitmap simple bitmap datatype to store glyph images.
LoadFont loading of fonts.
LoadImage loading of images.
FontFT loading of fonts via the FreeType library.
FontPK loading of PK-fonts.
FontTFM loading of TFM font metrics.
FontVirtual support for virtual fonts.
FreeType bindings for the FreeType library.
OpenType routines to parse OpenType tables.
Type1 routines to embed Type1 fonts.
There are modules for document formats.
Graphic datatypes for the primitive graphic commands.
PageDescription the datatype typeset documents are stored in.
Bezier routines to compute Bezier splines.
GenerateDVI routine to write DVI files.
PDF routines to load and write PDF files.
GeneratePDF routine to output a document as PDF file.
GeneratePostScript routine to write PostScript files.
GenerateSVG routine to output a document as SVG file.
And there are modules for file handling.
UCStream wrapper to read files and strings.

Logging output routines for error and debugging messages.

KPathSea bindings for the kpathsea library which implements a database for file name lookup.

5.2. The typesetting library

The typesetting library consists of the following modules:

Box definition of the various types of boxes.

Builder generic datatype for an engine assembling boxes.

Compose implements several builders for paragraphs.

HBox layout routines for horizontal boxes.

VBox layout routines for vertical boxes.

MathLayout all the various functions to layout mathematical material.

Glyph layout routines for accents and extendable glyphs.

Table layout routines for tables.

ParLayout the linebreaking algorithm.

Galley datatype for galleys.

Page datatypes for the page layout algorithm.

PageLayout the page layout algorithms.

AreaGalley layout of galley areas.

FloatVertical layout for float areas.

Footnote layout for footnote areas.

5.3. The layout engine

The layout engine consists of the following modules:

Environment definition of the state of the layout engine.

Node definition of the engine commands.

Evaluate implementation of the commands of the engine.

Fonts database for the installed fonts and the font selection mechanism.

HyphenTable This is a generated file which contains the hyphenation trie.

Job datatype to describe the current job. It contains the names of the input and output files, the date, and so on.

Output converts the pages into the format expected by the output routine and writes the DVI file.

The engine translates an abstract description of the document into a sequence of pages. The commands of this description are defined in the module Node:

`Nodes ⟨commands⟩

`Command ⟨loc⟩ ⟨env-modification⟩

`CommandBox ⟨loc⟩ ⟨contents⟩

`GfxCommand ⟨loc⟩ ⟨gfx-command⟩

`NewGalley ⟨loc⟩ ⟨name⟩ ⟨measure⟩

`NewLayout ⟨loc⟩ ⟨name⟩ ⟨width⟩ ⟨height⟩

`NewArea ⟨loc⟩ ⟨name⟩ ⟨x⟩ ⟨y⟩ ⟨width⟩ ⟨height⟩ ⟨max-top⟩ ⟨max-bot⟩ ⟨contents⟩

`ShipOut ⟨loc⟩ ⟨even-layout⟩ ⟨odd-layout⟩ ⟨number⟩

`AddToGalley ⟨loc⟩ ⟨name⟩ ⟨contents⟩

`PutGalleyInVBox ⟨loc⟩ ⟨align⟩ ⟨name⟩

`ModifyGalleyGlue ⟨loc⟩ ⟨function⟩

`Paragraph ⟨loc⟩ ⟨contents⟩

`BeginGroup ⟨loc⟩

`EndGroup ⟨loc⟩

`Float ⟨loc⟩ ⟨area⟩ ⟨contents⟩

`Glyph ⟨loc⟩ ⟨index⟩

`Letter ⟨loc⟩ ⟨code⟩

`Space ⟨loc⟩

`Glue ⟨loc⟩ ⟨width⟩ ⟨height⟩ ⟨implicit⟩ ⟨discardable⟩

`Break ⟨loc⟩ ⟨penalty⟩ ⟨hyphen⟩ ⟨pre-break⟩ ⟨post-break⟩ ⟨no-break⟩

`Rule ⟨loc⟩ ⟨width⟩ ⟨height⟩ ⟨depth⟩

`Image ⟨loc⟩ ⟨file⟩ ⟨width⟩ ⟨height⟩

`Accent ⟨loc⟩ ⟨accent⟩ ⟨body⟩

`HBox ⟨loc⟩ ⟨contents⟩

`HBoxTo ⟨loc⟩ ⟨width⟩ ⟨contents⟩

`HBoxSpread ⟨loc⟩ ⟨amount⟩ ⟨contents⟩

`VBox ⟨loc⟩ ⟨contents⟩

`VBoxTo ⟨loc⟩ ⟨width⟩ ⟨contents⟩

`VBoxSpread ⟨loc⟩ ⟨amount⟩ ⟨contents⟩
`Phantom ⟨loc⟩ ⟨horiz⟩ ⟨vert⟩ ⟨contents⟩
`HLeaders ⟨loc⟩ ⟨width⟩ ⟨contents⟩
`VInsert ⟨loc⟩ ⟨below⟩ ⟨contents⟩
`Table ⟨loc⟩ ⟨contents⟩
`TableEntry ⟨loc⟩ ⟨left⟩ ⟨right⟩ ⟨top⟩ ⟨baseline⟩ ⟨bottom⟩ ⟨contents⟩
`Math ⟨loc⟩ ⟨contents⟩
`MathCode ⟨loc⟩ ⟨math-code⟩ ⟨contents⟩
`MathChar ⟨loc⟩ ⟨math-char⟩
`SubScript ⟨loc⟩ ⟨script⟩
`SuperScript ⟨loc⟩ ⟨script⟩
`Fraction ⟨loc⟩ ⟨numerator⟩ ⟨denominator⟩ ⟨left⟩ ⟨right⟩ ⟨rule⟩
`Underline ⟨loc⟩ ⟨body⟩
`Overline ⟨loc⟩ ⟨body⟩
`MathAccent ⟨loc⟩ ⟨font-family⟩ ⟨character⟩ ⟨body⟩
`Root ⟨loc⟩ ⟨font-family⟩ ⟨character⟩ ⟨font-family⟩ ⟨character⟩ ⟨body⟩
`LeftRight ⟨loc⟩ ⟨contents⟩
`MathStyle ⟨loc⟩ ⟨style⟩
`IndexPosition ⟨loc⟩ ⟨pos⟩

5.4. The parser

The parser consists of the following modules:

CharCode contains mappings from characters to cat-codes.

Group implementations of `\begingroup` and `\endgroup`.

Mode commands to switch between the modes.

ParseState the state of the parser.

Parser the basic parsing routines.

ParseArgs parsing routines for command arguments.

Macro implementation of macros and environments.

Counter implementation of counters.

ALBindings AL-bindings for the various typesetting commands.

ALCoding conversion routines to and from AL-types.

ALDim AL-wrapper for dimensions.

ALEnvironment AL-wrapper for the environment of the engine.

ALGraphics AL-wrapper for graphic commands.

ALNodes AL-wrapper for nodes.

ALParseState AL-wrapper for the parse-state.

Primitives implementation of all primitive ANT commands.

Run This module contains the main entry point for the parser.

5.5. The virtual machine

The virtual machine consists of the following modules:

Types definitions of all types.

Opaque definitions to extend the virtual machine by user defined opaque types.

Lexer the lexer.

Parser the parser.

Scope datatype for scopes.

Compile the compiler.

Evaluate the core of the machine that evaluates expressions.

Machine a collection of helper functions for evaluation of expressions.

Serialise functions to write AL-values into a file and retrieve them again.

Primitives definitions of all primitive AL-commands.

Ali the main module for a primitive standalone AL-interpreter.