ANT
A TYPESETTING SYSTEM

“ant is not \TeX.”

Achim Blumensath
December 16, 2007

Contents

1. Invoking ant ........................................ 3
2. The markup language ............................. 4
3. AL – the ant language ............................ 4
   3.1. Lexical conventions .......................... 5
   3.2. Literals ........................................ 6
   3.3. Expressions ................................... 7
   3.4. Statements .................................... 9
   3.5. Patterns ....................................... 9
   3.6. Declarations .................................. 10
   3.7. Built in AL-commands ....................... 10
4. Typesetting commands ............................ 15
   4.1. Program control ............................. 15
   4.2. Page layout .................................. 16
   4.3. Galleys and paragraphs ..................... 19
   4.4. Boxes ......................................... 20
4.5. Parameters ............................................. 21
4.6. Fonts ................................................. 25
4.7. Tables .................................................. 28
4.8. Colour and graphics ................................. 29
4.9. Mathematics .......................................... 30
4.10. Macros and environments .......................... 32
4.11. Counters and references ............................ 33
4.12. Parsing ............................................... 35
4.13. Nodes .................................................. 36
4.15. Dimensions ......................................... 37

5. Overview over the source code ...................... 38
5.1. The runtime library .................................. 38
5.2. The typesetting library ............................. 40
5.3. The layout engine .................................... 40
5.4. The parser ............................................. 42
5.5. The virtual machine .................................. 43
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

\texttt{ant [options] \langle input file \rangle}

Currently, the following options are supported:

\begin{itemize}
\item [--help] prints a help message.
\item [--format=\langle format \rangle] selects the output format. Supported are dvi, xdvi, pdf (default), ps, and svg. The latter two are only partially implemented.
\end{itemize}
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 \texttt{ANT} therefore also provides a scripting language called \texttt{AL}. Syntactically \texttt{AL} resembles (a subset of) the Haskell programming language. But there are two notable semantic differences: (i) evaluation in \texttt{AL} is strict, not lazy and (ii) \texttt{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 `[]`.

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

\[
\begin{align*}
    \text{lid} & ::= \text{lc} (\text{lc} \mid \text{uc})^* \text{tail}^* \\
    \text{uid} & ::= \text{uc} (\text{lc} \mid \text{uc})^* \text{tail}^* \\
    \text{tail} & ::= \_ (\text{lc} \mid \text{uc})^* \_ \text{dd}^* \_ \text{sy}^* \\
    \text{symbol} & ::= \text{sy}^* 
\end{align*}
\]

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 _.

\[\begin{align*}
\text{number} & \ ::= \ \text{decimal} \ | \ \text{fraction} \\
\text{fraction} & \ ::= \ \text{natural} \ / \ \text{natural} \\
\text{natural} & \ ::= \ 0b \ \text{bin} \ | \ 0o \ \text{oct} \ | \ \text{dec} \ | \ 0x \ \text{hex} \\
\text{decimal} & \ ::= \ 0b \ \text{bin} \ [ \ . \ \text{bin}] \ | \ 0o \ \text{oct} \ [ \ . \ \text{oct}] \ | \ \text{dec} \ [ \ . \ \text{dec}] \ | \ 0x \ \text{hex} \ [ \ . \ \text{hex}] \\
\text{bin} & \ ::= \ 0b \ \text{bd} \ [(\_ | \ \text{bd})* \ \text{bd}] \\
\text{oct} & \ ::= \ 0o \ \text{od} \ [(\_ | \ \text{od})* \ \text{od}] \\
\text{dec} & \ ::= \ \text{dd} \ [(\_ | \ \text{dd})* \ \text{dd}] \\
\text{hex} & \ ::= \ 0x \ \text{hd} \ [(\_ | \ \text{hd})* \ \text{hd}] \\
\text{bd} & \ ::= \ 0 \ | \ 1 \\
\text{od} & \ ::= \ 0 \ | \ ⋯ \ | \ 7 \\
\text{dd} & \ ::= \ 0 \ | \ ⋯ \ | \ 9 \\
\text{hd} & \ ::= \ \text{dd} \ | \ \text{a} \ | \ ⋯ \ | \ \text{f} \ | \ \text{A} \ | \ ⋯ \ | \ \text{F}
\end{align*}\]

Examples:

\[3.4 \ 3/4 \ 0xf.a.4 \ 0o64/0b101 \ 1_000_000\]

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

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Unicode Codepoint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'</td>
<td>U0027</td>
<td>apostrophe</td>
</tr>
<tr>
<td>&quot;</td>
<td>U0022</td>
<td>double quote</td>
</tr>
<tr>
<td>&quot;</td>
<td>U005C</td>
<td>backslash</td>
</tr>
<tr>
<td>\b</td>
<td>U0007</td>
<td>bell</td>
</tr>
<tr>
<td>\e</td>
<td>U001B</td>
<td>ESC</td>
</tr>
<tr>
<td>\f</td>
<td>U000C</td>
<td>form feed</td>
</tr>
<tr>
<td>\n</td>
<td>U000A</td>
<td>newline</td>
</tr>
<tr>
<td>\r</td>
<td>U000D</td>
<td>carriage return</td>
</tr>
</tbody>
</table>

Symbol. Symbols are uppercase identifiers. The symbols \texttt{True} and \texttt{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.

\[
expr ::= \text{expr post-opp} \\
| \text{expr bin-op expr} \\
| \text{simple-expr}^+ \\
| \text{number simple-expr}^+ \\
| \text{local lid}^+ ; \text{expr} \\
| \text{local decl} ; \text{expr} \\
| \text{local begin decl-list end expr} \\
| \text{expr where decl-list end} \\
\]

... Function application is written without parenthesis or commas:

\[
\text{foldl (+) 0 [0,1,2,3,4,5]} 
\]

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

\[
4 \text{sind 45} \quad \text{is equivalent to} \quad 4 * \text{sind 45}. 
\]

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

\[
\text{local begin } x_1 := _{; \ldots x_n := _{; \text{end}}}
\]

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

<table>
<thead>
<tr>
<th>priority</th>
<th>assoc.</th>
<th>operators</th>
<th>priority</th>
<th>assoc.</th>
<th>operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>right</td>
<td>$</td>
<td>7</td>
<td>left</td>
<td>*/ mod</td>
</tr>
<tr>
<td>1</td>
<td>left</td>
<td>&gt;&gt;</td>
<td>8</td>
<td>left</td>
<td>~</td>
</tr>
<tr>
<td>2</td>
<td>right</td>
<td></td>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>right</td>
<td>&amp;&amp;</td>
<td>9</td>
<td>left</td>
<td>!!!</td>
</tr>
<tr>
<td>4</td>
<td>non</td>
<td>&lt;= &lt;= &lt;= &lt;= &lt;= &lt;= &lt;= &lt;=</td>
<td>5</td>
<td>left</td>
<td>\text{land lor lxor lsr lsl}</td>
</tr>
<tr>
<td>5</td>
<td>right</td>
<td>++</td>
<td></td>
<td>postfix</td>
<td>postfix operators</td>
</tr>
<tr>
<td>6</td>
<td>left</td>
<td>+ -</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Simple expressions.** Simple expressions are either literals, variables, or complex expressions enclosed in parenthesis. We discuss the various cases separately.
3. AL – the ant language

\[
\text{simple-expr} ::= \text{lid} \mid \text{symbol} \mid \text{number} \mid \text{character} \mid \text{string} \mid _ \mid \ldots
\]

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

\[
(\text{local } x ; x)
\]

\[
\text{simple-expr} ::= \ldots
\]

\[
\mid \text{pre-op \ simple-expr}
\]

\[
\mid (\ \text{expr} \ )
\]

\[
\mid (\ \text{bin-op} \ \text{expr})
\]

\[
\mid (\ \text{expr} \ \text{bin-op})
\]

\[
\mid (\ \text{bin-op})
\]

\[
\ldots
\]

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

\[
\text{simple-expr} ::= \ldots
\]

\[
\mid (\ \text{expr} \ ,\ \text{expr})^+ \ )
\]

\[
\mid [\ \text{expr} \ ,\ \text{expr}]^+ \ ]
\]

\[
\mid [\ \text{expr} \ ,\ \text{expr}]^* : \text{expr}
\]

\[
\ldots
\]

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

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

The following control constructs are available:

\[
\text{simple-expr} ::= \ldots
\]

\[
\mid \text{begin \ stmt-list-expr \ end}
\]

\[
\mid \text{do \ expr-stmt-list \ end}
\]

\[
\mid \text{if} \ \text{expr} \ \text{then} \ \text{stmt-list-expr}
\]

\[
(\text{elseif} \ \text{expr} \ \text{then} \ \text{stmt-list-expr})^*
\]

\[
\text{else} \ \text{stmt-list-expr \ end}
\]

\[
\mid \text{match} \ \text{expr} \ \text{with} \ \{ \ \text{match-body} \ }
\]

\[
\text{match-body} ::= \text{match-clause} \ (\mid \text{match-clause})^*
\]

\[
\text{match-clause} ::= \text{pattern} [\& \ \text{expr}] := \text{stmt-list-expr}
\]

\[
\text{stmt-list-expr} ::= (\text{stmt} ,)^* \text{expr}
\]

\[
\text{expr-or-stmt} ::= \text{expr} \mid \text{stmt}
\]

\[
\text{expr-stmt-list} ::= (\text{expr-or-stmt} ;)^* \text{expr-or-stmt}
\]
Lambda expressions, i.e., unnamed functions, are written as list of patterns and corresponding expressions similarly to \texttt{match} constructs:

\[
\textit{expr} ::= \ldots \mid \{ \textit{fun-body} \}
\]
\[
\textit{fun-body} ::= \textit{fun-clause} (\mid \textit{fun-clause})^*
\]
\[
\textit{fun-clause} ::= \textit{pattern}^* [\& \textit{expr}] := \textit{stmt-list-expr}
\]

For example:

\[
\{ [] := 0 \mid [x] := x \mid [x, y : _] := x + y \}
\]
\[
\{ x \& x > 0 := 1 \\
 0 > x \& x = 0 := 0 \\
 x \& x < 0 := -1 \}
\]

3.4. Statements

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

\[
\textit{stmt} ::= \textit{expr} = \textit{expr} \\
   \mid \textit{if} \textit{expr} \text{ then } \textit{stmt} \\
   (\textit{elseif} \textit{expr} \text{ then } \textit{stmt})^* \\
   [\textit{else } \textit{stmt}] \texttt{end}
\]

3.5. Patterns

\[
\textit{pattern} ::= _ \\
   \mid \textit{lid} \\
   \mid \textit{number} \\
   (\textit{pattern}) \\
   (\textit{pattern}, \textit{pattern})^* \\
   [\textit{pattern}, \textit{pattern}]^* \\
   [\textit{pattern}, \textit{pattern}] : \textit{pattern} \\
   \textit{lid} = \textit{pattern} \\
   \textit{pattern} = \textit{lid}
\]

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

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

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

\[
x_0 := _; \ldots x_n := _.
\]

The \text{declare...} statements can be used to declare the priority and associativity of operators.

3.7. Built in AL-commands

Control constructs. The function

\[
\text{error msg}
\]

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:

\[
\text{is_unbound } x \quad \quad \text{is_symbol } x \\
\text{is_bool } x \quad \quad \text{is_function } x \\
\text{is_number } x \quad \quad \text{is_list } x \\
\text{is_char } x \quad \quad \text{is_tuple } x
\]

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

\[
\text{||} \quad \&\& \quad \text{not}
\]

Comparison operators. The operators

\[
== \quad <> \quad > \quad < \quad >= \quad <=
\]

compare their arguments without modifying them. Equality == and inequal-
antity $<>$ are defined for all types. The other relations for numbers, characters, lists, and tuples where the latter ones are ordered lexicographically.

$$\min x y \quad \max x y$$

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 $\sim$ can be used for numbers and tuples. Multiplication $*$ and division $/$ are defined for the following types:

$$num_0 * num_1 \Rightarrow num_0 \cdot num_1$$

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

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

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

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

$$num_0 / num_1 \Rightarrow num_0 / num_1$$

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

$\text{abs}$ $x$ 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:

$$\text{round } x$$ in case of ties off zero

$$\text{truncate } x$$ in case of ties towards zero

$$\text{ceiling } x$$ up

$$\text{floor } x$$ down

The bitwise logical operations can be applied only to integers.
3. AL – the ant language

\begin{verbatim}
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
\end{verbatim}

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

\begin{verbatim}
pi              the constant \( \pi \)
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 \( x \)
\end{verbatim}

Lists and tuples. Lists and tuples are treated like functions mapping indices to values. That is, to access the \( n \)th 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 \texttt{length}.

\begin{verbatim}
length x.
\end{verbatim}

If the argument is of some other type \texttt{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

\[
\text{format_string} \quad \text{format arg}_1 \ldots \text{arg}_n
\]

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

\[
\text{to_list} \quad \text{to_tuple} \quad \text{x}
\]

can be used to convert between tuples and lists.

\[
\text{dir} \quad \text{angle vec}
\]

\text{dir} \( \text{angle} \) returns the unit vector in the given direction and \( \text{angle} \ (x, y) \) returns the angle (in degrees) of the given vector.

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

\[
\text{local d := \{ Foo := 1 | Bar := 2 \};}
\]

\[
d \text{Foo}
\]

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

\[
\text{add_to_dict} \quad \text{symbol value dictionary}
\]

It returns the new dictionary.

**Characters.** To test the category of a character the following functions can be used:
3. AL – the ant language

```
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\}}
```
4. Typesetting commands

defines a command \texttt{mirror} that reverses its argument.

The command

\begin{verbatim}\texttt{relax}\end{verbatim}
does nothing.

\begin{verbatim}\texttt{beginliteral \{string\} \end{literal}\end{verbatim}

converts the string into glyphs without interpreting it.

\begin{verbatim}\texttt{include \{file name\}}\end{verbatim}

reads the given file. The command

\begin{verbatim}\texttt{jobname}\end{verbatim}

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

\begin{verbatim}\texttt{ps_warning message}\end{verbatim}

\begin{verbatim}\texttt{ps_error message}\end{verbatim}

4.2. Page layout

\texttt{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

\begin{verbatim}\texttt{newpagelayout \{name\} \{page-width\} \{page-height\}}\end{verbatim}

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

\begin{verbatim}\texttt{newpagearea \{x\} \{y\} \{width\} \{height\} \{max-top\} \{max-bot\} \{type\} \{parameters\}}\end{verbatim}

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 \texttt{\{type\}} is \texttt{galley} then the contents is taken from the galley specified by the dictionary \texttt{\{parameters\}}.

\begin{verbatim}\texttt{name \{string\} name of the galley.}\end{verbatim}
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 \( \langle \text{type} \rangle \) is \texttt{direct} then \( \langle \text{parameters} \rangle \) contains \texttt{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 \texttt{page}. The marks from previous pages are given as global \texttt{AL}-variables where the name is prefixed with \texttt{OldMark}, i.e., a mark named \texttt{Foo} can be accessed by the command

\begin{verbatim}
local x;
do
  ps_get_global x OldMarkFoo;
  ps_insert_string x;
end
\end{verbatim}

Similarly, marks found in the current page get the prefix \texttt{NewMark}.

If \( \langle \text{type} \rangle \) is \texttt{float} then the area is used to display floats. The dictionary \( \langle \text{parameters} \rangle \) contains the following entries:

alignment (default \texttt{top}) either \texttt{top} or \texttt{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, \( \langle \text{type} \rangle \) can be \texttt{footnote} in which case the area is used to display footnotes. The dictionary \( \langle \text{parameters} \rangle \) contains the following entries:

separator (\texttt{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.
4. Typesetting commands

**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
The predefined page layouts can be used with the routines

\texttt{\texttt{simple\_page\_layout page\_size division baseline}}
\texttt{two\_column\_page\_layout page\_size division baseline}

The first one creates a galley named \texttt{main} and two page layouts \texttt{left} and \texttt{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 \texttt{page\_size} contains the paper size, \texttt{division} determines the margins (a good value is 9), and \texttt{baseline} is the font height (including leading).

4.3. Galleys and paragraphs

The layout process of \texttt{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 \texttt{(measure)} one uses the command

\texttt{\texttt{\textbackslash newgalley \texttt{name} \texttt{measure}}}

The corresponding \texttt{AL}-command is

\texttt{ps\_new\_galley name measure}

All material between the commands

\texttt{\textbackslash begin\_galley \texttt{name}}
\texttt{\textbackslash end\_galley}

is appended to the galley \texttt{(name)}.

As in \TeX{} the end of a paragraph is marked by either one of the following commands:

\texttt{\textbackslash endgraf}
\texttt{\textbackslash par}

An empty line is automatically translated to the command sequence \texttt{\textbackslash par}. 
4. Typesetting commands

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 \texttt{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:

- \texttt{letter}
- \texttt{operator}
- \texttt{inner}
- \texttt{ordinary}
- \texttt{punct}
- \texttt{subscript}
- \texttt{binop}
- \texttt{open}
- \texttt{superscript}
- \texttt{relation}
- \texttt{close}

For \texttt{letter}, the character is given by a \texttt{UNICODE} number while in the other cases a glyph index is expected.

\penalty \{number\}

inserts a break point with the given penalty.

\discretionary * [\texttt{penalty}] \{\texttt{pre-break}\} \{\texttt{post-break}\} \{\texttt{no-break}\}

inserts a break point with the given parameters. The * indicates a break caused by hyphenation. The default for \texttt{(penalty)} is 0, if * is omitted, and \texttt{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 \texttt{(skip)} if necessary.

\hbox \{(body)\}
\hbox to \{width\} \{(body)\}
\hbox spread \{amount\} \{(body)\}

create a horizontal box around \texttt{(body)}.

\vbox \{(body)\}
4. Typesetting commands

\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:

- \texttt{width} (skip) the width of the image.
- \texttt{height} (skip) the height of the image.
- \texttt{dpi} (number) the resolution of the image.

DVI specials can be created with the AL-command

\texttt{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:

\[(range) : (indent)\]
4. Typesetting commands

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:
4. Typesetting commands

- **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:
4. Typesetting commands

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.
4. Typesetting commands

\texttt{script-size} (number) the script size.

\texttt{script-script-size} (number) the double script size.

The macros

\begin{verbatim}
\FontFamilyRoman
\FontFamilySans
\FontFamilyTypewriter
\FontFamilyMath
\FontFamilyExtensions
\FontFamilySymbols
\end{verbatim}

contain the default families used by the font commands below.

\begin{verbatim}
\FontSeriesMedium
\FontSeriesBold
\end{verbatim}

contain the default series used by the font commands below.

\begin{verbatim}
\FontShapeUpright
\FontShapeItalic
\FontShapeSlanted
\FontShapeSmallCaps
\end{verbatim}

contain the default shapes used by the font commands below.

\begin{verbatim}
\FontSizeTiny
\FontSizeScript
\FontSizeFootnote
\FontSizeSmall
\FontSizeNormal
\FontSizeLargeI
\FontSizeLargeII
\FontSizeLargeIII
\FontSizeHugeI
\FontSizeHugeII
\end{verbatim}

contain the default sizes used by the font commands below.

\begin{verbatim}
\rmfamily
\sffamily
\ttfamily
\end{verbatim}

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 \texttt{ANT} you have to declare them. The \texttt{AL}-command
\begin{verbatim}
ps_declare_font font-file family series shape sizes parameters
\end{verbatim}
tells \texttt{ANT} that the file \texttt{font-file} contains a font in the given \texttt{family}. The \texttt{parameters}
take the form of a dictionary containing the following entries. Each of them is optional.

\begin{description}
\item[Encoding] the encoding vector of the font.
\item[HyphenGlyph] the index of the hyphen glyph.
\item[SkewGlyph] the index of the skew glyph.
\item[Scale] an optional scaling factor for the font.
\item[LetterSpacing] amount of additional letter spacing.
\item[Adjustments] additional kerning and ligature commands.
\end{description}
AutoLigatures  boolean to enable automatic creation of ligatures.

BorderKern  list of tuples containing kerning values for margin kerning.

Example:

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

To define mathematical symbols one can use the following commands.

\begin{verbatim}
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
\end{verbatim}

4.7. Tables

The commands

\begin{verbatim}
\begintable
\endtable
\newtableentry
\newtablerow
\end{verbatim}

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

\begin{verbatim}
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
\end{verbatim}

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 \textsc{al}-commands are

\texttt{ps\_set\_colour colour}
\texttt{ps\_set\_bg\_colour colour}
\texttt{ps\_set\_alpha alpha}

Colours can be specified in one of three formats:

- (Grey, x)
- (RGB, \textit{red}, \textit{green}, \textit{blue})
- (CMYK, \textit{cyan}, \textit{magenta}, \textit{yellow}, \textit{black})

The following \textsc{al}-commands can be used to draw lines or filled shapes.

\texttt{ps\_stroke path}
\texttt{ps\_fill path}
\texttt{ps\_clip path}
\texttt{ps\_set\_line\_width width}
\texttt{ps\_set\_line\_cap line-cap}
\texttt{ps\_set\_line\_join line-join}
\texttt{ps\_set\_miter\_limit limit}

To construct paths \textsc{ant} provides the following \textsc{al}-commands:

\texttt{make\_path point}
\texttt{close\_path cycle path}
\texttt{path\_add\_point point path}
\texttt{path\_add\_in\_dir vector path}
\texttt{path\_add\_in\_angle angle path}
\texttt{path\_add\_in\_curl curl path}
\texttt{path\_add\_in\_tension tension path}
\texttt{path\_add\_out\_dir vector path}
\texttt{path\_add\_out\_angle angle path}
\texttt{path\_add\_out\_curl curl path}
\texttt{path\_add\_out\_tension tension path}
path_add_control_points point point path

You start a path with \texttt{make_path} at the given point. You can add new points with \texttt{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

\begin{verbatim}
\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
}}}
\end{verbatim}

4.9. Mathematics

\begin{verbatim}
$ (math) $
\begin{math}
\end{math}
\begin{text}
\end{text}
\_ (subscript)
\^ (superscript)
\end{verbatim}

The usual math commands. \texttt{\begin{text}} and \texttt{\end{text}} can be used to enter text-mode when in math-mode. Note that both \texttt{\begin{math}} and \texttt{\end{math}}, and \texttt{\begin{text}} and \texttt{\end{text}} nest.

\texttt{\frac (numerator) (denominator)}
\textit{ant 0.8} \hfill 4. Typesetting commands

\begin{verbatim}
\genfrac{left}{right}{thickness}{numerator}{denominator}
\sqrt{body}
create a fraction and a root.
\overline{body}
\underline{body}
put a line atop or below \textit{body}.
\left\{delimiter\middle\{body\}\right\{delimiter
adjusts the height of the delimiters to that of the \textit{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 \textit{body}.
\indexposition{pos}
\limits
\nolimits
determines where the following sub- and superscripts are placed. \textit{pos} can take
the values left, right, and vert. \texttt{\limits} and \texttt{\nolimits} are shorthands
for \texttt{\indexposition{vert}} and \texttt{\indexposition{right}}, respectively. For
example, the command
\prod\indexposition{left}^a_b \indexposition{vert}^c_d \indexposition{right}^e_f
produces the output
\end{verbatim}
In addition, all the usual mathematical symbols are defined: \( \alpha, \sim \ldots \)

### 4.10. Macros and environments

\[
\alpha \prod_{d}^{c} f
\]

define a new command. For \texttt{\definecommand\{name\} [arguments] \{body\}} has to be a command sequence in the sense of \TeX, while in the case of \texttt{\definepattern\{name\} [arguments] \{body\}} 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 \texttt{(arguments)} consists of a list of letters specifying the type of the arguments:

- \texttt{m} mandatory argument
- \texttt{s} optional \texttt{*}
- \texttt{o} optional argument with empty default
- \texttt{O{(default)}} optional argument with default value

\[
\begin{array}{c}
\texttt{\savecommand\{name\}} \\
\texttt{\restorecommand\{name\}} \\
\texttt{\savepattern\{name\}} \\
\texttt{\restorepattern\{name\}} \\
\end{array}
\]

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

\[
\begin{array}{c}
\texttt{\definecommand\{\foo\}\{old\}} \\
\texttt{\foo} \\
\texttt{\savecommand\foo} \\
\texttt{\definecommand\{\foo\}\{new\}} \\
\texttt{\foo} \\
\texttt{\restorecommand\foo} \\
\texttt{\foo} \\
\end{array}
\]

produces the output \texttt{old new old}.

\[
\begin{array}{c}
\texttt{\defineenvironment\{name\} [arguments] \{begin-body\} \{end-body\}} \\
\end{array}
\]

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:

\begin{verbatim}
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
\end{verbatim}

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:

\begin{verbatim}
\newcounter [super-counter] (name)
\setcounter (name) (value)
\addtocounter (name) (value)
\getcounter (format) (name)
\end{verbatim}

If \textbf{(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
- \textbf{r}\{text\} repeats \{text\} \textit{i} times
- \textbf{s}\{text 1\}...\{text n\} returns \{text \textit{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
4. Typesetting commands

- `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`
4. Typesetting commands

\begin{verbatim}
set_current_math_params \textit{params}
set_par_shape \textit{shape}
set_colour \textit{colour}
\end{verbatim}

Page layout:

\begin{verbatim}
new_page_layout \textit{name} \textit{page-width} \textit{page-height}
select_page_layout \textit{name}
\end{verbatim}

Fonts:

\begin{verbatim}
set_math_font \textit{definition}
adapt_fonts_to_math_style
\end{verbatim}

Space factor:

\begin{verbatim}
get_space_factor \textit{env char}
adjust_space_factor \textit{char}
\end{verbatim}

4.15. Dimensions

The following constants are defined:

\[
\begin{align*}
\text{pt} &= 1 & \text{pc} &= 12 \text{ pt} \\
\text{in} &= 72.27 \text{ pt} & \text{sp} &= 1/65536 \text{ pt} \\
\text{bp} &= 1/72 \text{ in} & \text{dd} &= 1238/1157 \text{ pt} \\
\text{cm} &= 1/2.54 \text{ in} & \text{cc} &= 12 \text{ dd} \\
\text{mm} &= 0.1 \text{ cm}
\end{align*}
\]

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.

\begin{verbatim}
make_dim \textit{base} \textit{stretch} \textit{stretch-order} \textit{shrink} \textit{shrink-order}
fixed_dim \textit{base}
dim_zero
dim_1pt
dim_12pt
dim_fil
dim_fill
dim_ss
dim_filneg
dim_equal \textit{dim} \textit{dim}
dim_add \textit{dim} \textit{dim}
\end{verbatim}
5. Overview over the source code

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.
edImage  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.
Beziers  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)
```
5. Overview over the source code

`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.
5. Overview over the source code

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.