

Printing (Opal-)Programs with "opal2x.sty"

Peter Pepper

October 4, 2010

Contents

1	Example	2
2	The opal and program Environments	4
2.1	Customization	4
2.2	Embedded programs	5
3	Layout commands	5
3.1	Basic layout commands	5
3.2	Keyword-oriented layout commands	6
3.3	Empty Lines, Rules, and Shrinking Boxes	8
3.4	Comments and Line Numbers	8
3.5	Spaces	10
4	Fonts (for Identifiers, Keywords, etc.)	10
4.1	Lisp-like identifiers	11
5	Miscellaneous	11
6	Special Symbols	11
6.1	Predefined symbols	11
6.2	Defining your own symbols	12
6.3	Sub/superscripts	13
7	How the Scanner Recognizes Symbols	13
7.1	Using the scanner in math mode	14
8	Trouble shooting	14

1 Example

The use of this style is illustrated by the following example.

STRUCTURE Example	1
IMPORT Nat ONLY nat: SORT	2
0 1 2 - =	3
String COMPLETELY	4
FUN fib: nat → nat -- the Fibonacci function	5
DEF fib(n) ==	6
IF n = 0 ∨	7
n = 1 THEN 1 --termination	8
ELSE fib(n - 1) + fib(n - 2)	9
FI	10
FUN fib: string → string -- for I/O	11
DEF fib(s) == (fib(s!)) --overloading	12

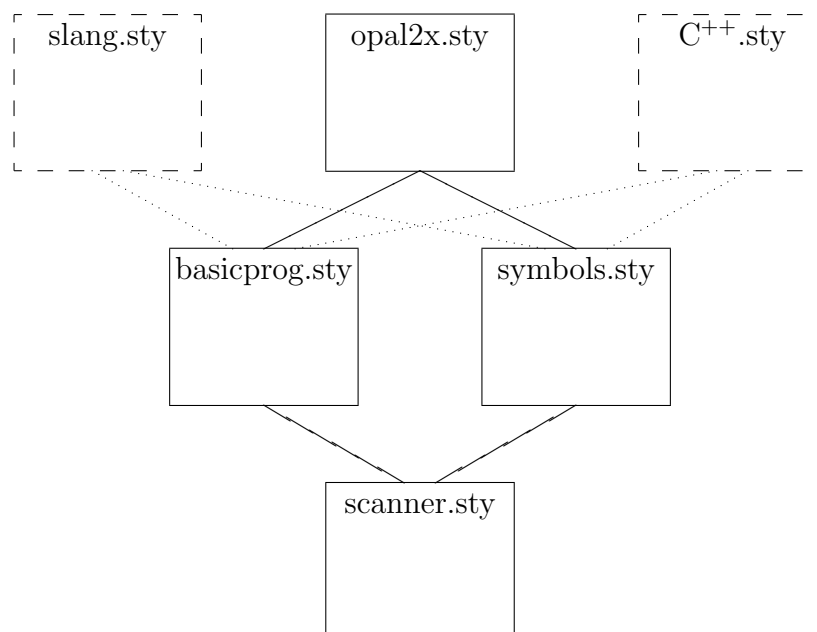
In order to obtain this text you have to write

```
\begin{opal}[fn]
STRUCTURE Example
-----
IMPORT Nat ONLY nat:\SORT
0~1~2~-~=
String COMPLETELY
-----
FUN fib: nat -> nat \- the Fibonacci function
DEF fib(n) ==
IF n=0~/
n=1 THEN 1 -- termination
ELSE fib(n-1)+ fib(n-2)
FI
FUN fib: string -> string \- for I/O
DEF fib(s) == (fib(s!)) -- overloading
\end{opal}
```

How does the style work?

The style essentially provides an environment `\begin{opal}... \end{opal}` within which program layouting is supported. The layout itself is determined by way of the OPAL keywords and the basic commands `\[`, `\&`, and `\]` (see below).

The style `opal2x.sty` is based on three other styles (which can also be used independently). The overall dependency relationship is illustrated by the following diagram (which also points out that styles for other languages could be as easily defined).



An essential feature of the style is the *scanner*. It recognizes identifiers, keywords, etc. in `program` environments. Above all, it allows to use ‘fancy commands’ for having more readable input (such as `<=>` for \Leftrightarrow).

Loading the style(s)

The styles are loaded by

```
\usepackage[ams,suppress]{opal2x}
\usepackage{basicprog}
\usepackage[ams,suppress]{symbols}
\usepackage{scanner}
```

Note: Loading any of the styles automatically also loads the styles on which it depends. Hence it suffices to load `opal2x.sty`.

The *options* have the following effect:

- When `ams` is given, the style `amssymb` is automatically loaded.
- When `suppress` is given, the special symbols (see Table 9) are prepared, but not yet defined. So the user can define them at some suitable point in the text, in particular also locally in certain environments.

Note: There may be problems, when the style is loaded after `[german]babel` due to conflicting macro declarations (but most of these problems appear to be solved).

2 The opal and program Environments

The `opal` environment and the `program` environment are written in the same form using the same optional arguments.

```
\begin{opal}[pcfn]
...
\end{opal}

\begin{program}[pcfn]
...
\end{program}
```

The `opal` environment actually creates a `program` environment after introducing a number of OPAL-specific definitions (keywords etc.).

The optional arguments — which can be given in any selection and order¹ — have the following effects:

p[lain]	No special options apply. (This is the default.)
c[enter]	The program is centered on the page.
f[ramed]	The program is framed.
n[umbered]	Line numbers are automatically generated.

2.1 Customization

A number of commands can be used to customize the appearance of the style. They can be redefined arbitrarily often in the text.

- `\ProgramWidth{...}` sets the width of programs. The default is `\textwidth`.
- `\ProgramLeftIndent{...}` sets the left indentation of programs; default `\parindent`.
- `\ProgramRightIndent{...}`: Analogous; default is `0pt`.
- `\ProgramInnerIndent{...}` determines the indentation of main keywords (such as `FUN`) relative to top keywords (such as `SIGNATURE`). The default is `1em`.
- `\ProgramSkip{...}` determines the vertical distance of programs from the surrounding text. The default is `\parskip`.
- `\EmptyLineSkip{...}` sets the amount of vertical space that is effected by an empty input line (see section 3.3).

Finally, the user can insert his own definitions into `program` and `opal` environments by means of the commands

- `\renewcommand{\ProgramPrelude}{...}`

¹... but not all combinations make sense

- `\renewcommand{\OpalPrelude}{...}`

`\ProgramPrelude` is executed at the beginning of the `program` environment — in analogy to `\everymath` for math mode. And `\OpalPrelude` is in addition executed at the beginning of `opal` environments.

2.2 Embedded programs

Whereas the `program` environment is comparable to displayed math, there is also an `embeddedProgram` environment that is comparable to normal math mode. This environment is provided by

```
\begin{embeddedProgram} ... \end{embeddedProgram}
```

or by its shorthand form

```
\(...\)
```

which both can be written in the middle of a text in order to obtain the special features of programs (such as keywords and special symbols). Of course, the layout commands are not meaningful here.

In analogy to the `program` environment, one can also introduce private definitions at the beginning of `embedded programs` by using

```
\renewcommand{\EmbeddedProgramPrelude}{...}
```

3 Layout commands

In the `program` and `opal` environments programs can be typeset by using two kinds of commands.

- The *basic commands* leave full control of the layout to the user (at the cost of less readable input).
- The *keyword-oriented commands* employ a bunch of heuristics for automatic layout generation.²

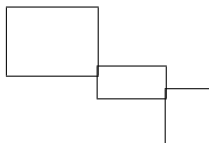
3.1 Basic layout commands

The design philosophy of the style `basicprog.sty` is to provide a very small set of basic layout commands, on which all other commands (see later on) can be based.

Essentially, a program environment consists of *nested tabular environments*³. The main problem is that in programs the following arrangement of boxes is quite typical:

²As always, heuristics don't work satisfactorily in all circumstances. Then it is possible to combine the advanced layout features with the basic commands.

³We use the L^AT_EX terminology here, even though we actually create T_EX `halign` commands here



Unfortunately, this kind of layout is not provided by \TeX , and therefore a lot of calculations are necessary in order to achieve the desired effect. These calculations are effected by the following commands.

- `\[` opens a new tabular environment.
- `\&` separates two columns.
- `\]` closes a tabular environment.

Between any two of these commands \TeX is in math mode.

3.2 Keyword-oriented layout commands

The heuristic layouting should be controlled by commands that correspond to keywords in the underlying language (such as `DEF`, `IF`, `THEN`, etc.). These are usually introduced by language-specific styles such as `opal2x.sty`. The style `basicprog.sty` only provides means for introducing such special commands in other styles.

The keywords of `opal2x.sty`

Within the `opal` environment every OPAL keyword such as `SIGNATURE`, `DEF`, `IF`, `AS`, etc. actually is a \TeX command⁴ that not only produces the keyword but also generates basic layout commands according to certain heuristics.

Tables 3 and 4 list all keyword-oriented layout commands defined in `opal2x.sty` together with the commands, by which they are generated.

Note 1: All these keyword-oriented layout commands can be combined with the primitive commands `\[`, `\&`, `\]`, `\-`, `\%` etc.

Note 2: Together with any “fancy” layout command (such as e.g. `IF`) the style also generates a standard \LaTeX command (such as `\IF`) that merely creates the corresponding keyword.

Note 3: Between any two of these commands \TeX is in math mode.

The keyword-generating commands of `basicprog.sty`

Table 5 lists the commands through which layout-defining keywords can be introduced.

- For instance, `\BeginKeyWord{IF}` has the following effects:

⁴Note that *no* backslash is needed! This is made possible by the scanner described in Section 7.

- It creates a “fancy command” `IF` (recognized by the scanner) that embodies a certain layout heuristics including the keyword `IF`.
- In addition, it also creates a command `\IF` that represents the plain keyword `IF`.

The following list illustrates these commands by way of characteristic examples.

- `\TopKeyWord{STRUCTURE}`.

This command first closes all open tabular environments. The rest is equivalent to `\[\STRUCTURE~\`.

- `\MainKeyWord{DEF}`.

This command first closes all open tabular environments. The rest is equivalent to `\[...\DEF~\`. (The ... indicate that here a space of size `\ProgramInnerIndent` is inserted.)

- `\FollowKeyWord{ONLY}`.

This is essentially equivalent to `\&~\ONLY~\&\`.

- `\BeginKeyWord{LET}`.

This is essentially equivalent to `\[~\LET~\&\`.

- `\IfLikeKeyWord{IF}`.

This is essentially equivalent to `\[~\IF~\&\`.

- `\ThenLikeKeyWord{THEN}`.

This is essentially equivalent to `\]\&\THEN~\&\`.

- `\ElseLikeKeyWord{ELSE}`.

This is essentially equivalent to `\]\&~\ELSE~\&\`.

- `\EndKeyWord{FI}`.

This is essentially equivalent to `\]\~\FI~\]`.

- `\PlainKeyWord{AS}`.

This is essentially equivalent to `~\AS~`.

Note 1: Most of these keywords generate slightly different layouts, when they appear as the first element on a new line.

Note 2: All of these keywords can be provided with an optional argument that determines the width of its keyword. For instance

- `\BeginKeyWord[1em]{OTHERWISE}`

shortens the keyword to the length 1em. This way, the keyword doesn't spoil the layout by creating huge gaps after the corresponding IFs.

Note 3: The `\IfLikeKeyword` uses some special heuristics to cope with the layout of OPAL's special IF-sequences.

3.3 Empty Lines, Rules, and Shrinking Boxes

A number of commands can be used to customize the appearance of programs.

- `\Break` enables a page break at this point.

Note: The `\Break` command automatically closes all open tabular environments! Therefore it is only meaningful between top or main keywords (see below). `\Break` should be on a line of its own.

- **Empty lines** in the input of a program do *not* really create an empty line in the output, but rather only a larger vertical space between the two surrounding lines. The size of this space can be set by the macro `\EmptyLineSpace{...}` (default is `.6\baselineskip`).

If one really wants an empty line, one must write “invisible” texts such as `~` or `{ }`. (In this case the empty line will also be included in the automatic numbering, if this feature is set for the program.)

- `\Rule` draws a horizontal line accross the program — provided that the frame option is on.

Note: The `\Rule` command has essentially the same properties as the `\Break` command!

- `-----` (5 dashes⁵) is equivalent to `\Rule`.
- `\0` sets the width of the following tabular environment to zero. That is, you write `\0[...]` in order to obtain a piece of program that does (almost) not affect the layout of the surrounding program.
- `\Shrink{...}` sets its argument text into a box of width zero. (The argument must not include tabular stuff like `\[, \&, \]`, or related commands.)

3.4 Comments and Line Numbers

The `program` and `opal` environments provide features for aligning comments and for adding line numbers.

⁵Note that the scanner is employed to recognize this symbol.

Comments

Two kinds of comments are recognized (by the scanner). They reach from the start symbol to the end of the line.

T_EX comments are started – by default – by the command `\TeXCmnt`. This possibility is provided, because the symbol `%` is a normal character in many programming languages.

T_EX comments are – as usual – completely ignored, including the end of line.

The style `opal2x.sty` introduces the symbol `\%` as the start symbol for T_EX comments (see below).

Program comments are started – by default – by the symbol `\Cmnt`. They turn everything until (but excluding) the end of line into a comment, designated by the command `\@Cmnt`. This command can be redefined by other styles in order to customize the layout of comments.

The scanner is switched *off* inside these comments!

However, if `\ActivateMathScan` is switched on, then the scanner is reactivated, whenever `$. . . $` occurs in the comment.

The style `opal2x.sty` introduces the symbol `\-` as start symbol for program comments (see below).

The following commands can be used to create these (different kinds of) comments.

- `\%` starts a T_EX comment⁶ (that includes the end of the line and thus can be used to write several input lines for one output line).
- `\-` starts an *aligned program comment* (that reaches to the end of the line). The printed comment symbol is ‘--’.
- `\UaCmnt` starts an unaligned program comment. (This command is usually not used itself, but equivated to special symbols, such as `--` in `opal2x.sty`.)
Unfortunately, the unaligned comments don’t work as nicely as aligned comments.⁷ Above all, it may happen that spaces don’t show (see “trouble shooting”).
- `--` starts an *unaligned* program comment in `opal` environments.

Customizing the appearance of program comments is possible throught the following command.

- `\renewcommand{\Comment}[1]{...}` defines the appearance of comments; The default is `\textit{\#1}`.

⁶This is necessary, because the symbol `%` is needed as a normal character in most programming languages.

⁷This has technical reasons related to internal constraints of the scanner.

Line numbers

There are two ways for obtaining line numbers. Either one sets the option ‘[n]’ for the `program` or `opal` environment; then all lines are consecutively numbered. Or one uses the following command in order to achieve selective numbering.

- `\lnr` is written at the end of those lines that shall be numbered. (*Note:* The `\lnr` command has to appear *before* the comment.)

The `\lnr` command has no effect, when the option [n] is given.

Customizing of the appearance of line numbers is possible by the two commands

- `\renewcommand{\Number}[1]{...}` defines the appearance of the line numbers. The default is `\texttt{\footnotesize#1}`.
- `\NumberWidth{...}` defines the width of the line numbers. The default is `2em`.

3.5 Spaces

Spaces are ignored. (More precisely: They obey the rules of the T_EX mathematics mode.) If you want a *hard space*, you can write ‘~’. That is, `f~1~y` produces `f 1 y`.

For convenience, there is one *exception* to this rule: If two identifiers are separated by spaces, then a hard space is generated between them. For example, `f x y` produces `f x y`.

4 Fonts (for Identifiers, Keywords, etc.)

By default, *identifiers* and *digits* are written in `\mathtt` or `\texttt` font (depending on their context), and *keywords* are written using small capitals.

This can be changed by redefining the commands `\KeyWord`, `\IdentifierFont` and `\DigitFont` (preferred) or (not so recommendable) by redefining `\Identifier` and `\Digit` (all defined in `scanner.sty`). For example:

- `\IdentifierFont{it}` makes identifiers italic.
- `\DigitFont{rm}` makes digits roman.
- `\renewcommand{\KeyWord}[1]{...}` redefines the appearance of keywords. The default is `\textsc{\lowercase{#1}}`.

In texts, an identifier `index` may be obtained by writing `\Identifier{index}` and similarly a keyword `FUN` by writing `\KeyWord{FUN}` or, shorter, `\‘FUN’`. But an alternative possibility is to use embedded programs, that is `\(FUN\)` and `\(index\)`.

Note: `\‘foo’` does *not* work in environments, where `\‘` is used as a special macro (for instance in tabular environments). Then you should use `\KeyWord{foo}`.

4.1 Lisp-like identifiers

Normally, a LISP-like identifier such as `query-replace` cannot be written in the `opal` and `program` environments, because the outcome would be `query – replace`. To cope with this notation (if needed) there are two commands:

- `\EnableLisp`

After this command has been given, identifiers such as `query-replace` are written in that form. (That is, a ‘-’ is *not* considered as a ‘minus’, if it occurs between two letters.)

- `\DisableLisp` (default)

After this command has been given, `query-replace` is written as `query – replace` again.

5 Miscellaneous

A number of further features and commands are also part of the style.

- `\Language{...}` is used to write all language names in a uniform font (default is `\textsc`). Example: `\Language{Haskell}` yields `HASKELL`. (Note: This command uses `xspace.sty`.)
- `\DefLanguage{...}` introduces a command for a language name. For example, `\DefLanguage{Haskell}` allows us to write afterwards `\Haskell` in order to obtain `HASKELL`.
- `\Opal` is predefined and yields the name `OPAL`. (Note that the `\xspace` command is used here; hence, the command can be followed by a space.)

6 Special Symbols

In order to make the input text for programs (and mathematics) more readable, a number of special commands are introduced by the package `symbols.sty`.

The user can easily define new symbols himself.

6.1 Predefined symbols

A major feature of our styles is that they allow readable input for many special symbols by providing ‘fancy’ \TeX commands. Example: In order to obtain \iff one merely has to write `<==>`.

Table 9 gives the list of the ‘fancy’ symbols that are introduced by the style `symbols.sty`.

Some characters — such as % or # — that have special meanings in T_EX have to be set to normal in the **program** environment, since they are standard characters in programming languages. This is e.g. done for the symbols in Table 7 by the **opal** environment.

6.2 Defining your own symbols

The tokens assembled by the scanner are usually treated as identifiers, digits, or graphemes. However, there is also a way to turn them into *fancy commands*, that is, commands that cannot normally be written in T_EX or L^AT_EX. This is done by the following commands:

- **\SetCommand**
defines its first argument as new “fancy” command and takes the second argument as its meaning. Example: `\SetCommand{FUN}{\Keyword{fun}}` makes FUN into a command that produces `\Keyword{fun}`.
- **\SetSymbol**
defines its first argument as new “fancy” command and takes the second argument as its meaning. (By contrast to `\SetCommand` this command takes math mode into consideration.) Example: `\SetSymbol{<=>}{\Leftrightarrow}` makes `<=>` into a command that generates “ \Leftrightarrow ”.
- **\SetAmsSymbol**
is analogous to `\SetSymbol`, but has a third argument, which is used instead of the second one, when `amssymb` is not loaded. Example:

```
\SetAmsSymbol{>->}{\rightarrowtail}{%
  \mathbin{\raise0.18ex\hbox{$\scriptstyle>$}}%
  \hskip-0.4em-\hskip-0.75em\raise0.18ex\hbox{$\scriptstyle>$}}}
```

Using `>->` creates either “ \rightarrow ” or “ \Rightarrow ”, depending on the availability of `amssymb`.

- **\UnSetSymbol**
annihilates a fancy symbol. Example: `\UnSetSymbol{<=>}`.
- **\SetMacro**
introduces the definition of a macro into the T_EX “variable” `\Special@Macros`. This variable is called in the **program** environment in order to make all these macros available. Example: `\SetMacro{\vee}{\vee}` defines `\vee` to stand for `\vee` in the **program** environment.

If you have set the option `[suppress]`, then you can define all the symbols in Table 9 by issuing the command

- **\SetAllSymbols**

6.3 Sub/superscripts

Sub- and superscripts in mathematical formulas are somewhat problematic, because the two symbols \wedge and $_$ are printable characters in many programming languages. Therefore, they are converted into normal symbols in the `opal` environment (but *not* in the `program` environment).

However, we do not want to lose the ability to use these symbols also for sub- and supersetting. Here are the rules:

- When \wedge is followed by a $\{\dots\}$ -group, then it acts as `\sp{\dots}`.
- When \wedge is followed by a digit, then it acts as `\sp`.
- In all other cases the character \wedge is printed.

The rules for $_$ are analogous. Table 6 shows examples for the effects of sub- and supersetting.

7 How the Scanner Recognizes Symbols

The style `scanner.sty` essentially implements a *scanner* for analyzing (program) texts. It thus is a prerequisite for providing

- readable notations for graphemes (such as `"<==>"` for obtaining \Longleftrightarrow),
- nice appearance for identifiers in math mode (such as *buffer* instead of `buffer`),
- more flexible notations for keywords (such as `FUN` instead of `\FUN`).

The style provides commands that can be used by other styles to scan certain text areas. These text areas are usually programs, but one can also scan math formulas (see below). The commands scan the designated text area and assemble essentially three classes of tokens:

- *Identifiers*, that is, sequences of letters (e.g. `foo`).
- *Digits*.
- *Graphemes*, that is, sequences of “other” characters (e.g. `[+]`).

Note: The characters ‘(’, ‘)’, and ‘,’ are *not* considered as “other” characters and therefore cannot be used as parts of fancy symbols.

For identifiers and graphemes the principle of *longest match* applies. This is no problem, since spaces can be used as separators without problems.

In addition, the scanner recognizes:

- End of line.
- \TeX comments (i.e. \%).
- Program comments i.e. \-).

7.1 Using the scanner in math mode

The scanner is usually used by other styles such as `opal2x.sty` to analyze program texts. However, it is also possible to use it in math mode, that is, within \dots . In order to activate the scanner in math mode, the command

`\ActivateMathScan`

has to be issued. (It is equivalent to `\everymath{\ScanMath}\EnableMathScan`.)

Afterwards, the two commands

`\DisableMathScan`

`\EnableMathScan`

can be used arbitrarily often to disable and reenale the scanner for math mode.

8 Trouble shooting

The styles appear to be working acceptably well. There are a few typical problems that may arise.

- A wide program fragment may cause *large gaps* in the surrounding program. This is due to the fact that everything is organized as nested tabular environments.

Solution: The commands `\0` and `\Shrink` can be used to create zero width.

- A *line number* appears on the successive line.

Solution: The `\lnr` has to come *before* the comment.

- In *unaligned comments* the *spaces* may disappear, or there may be undesired spaces.

Solution 1: Missing spaces have to be enforced by ‘~’.

Solution 2: Unwanted spaces are trickier. Suppose you obtain something like $f(x)$. The only way to get rid of the spaces is to write your own little command

`\newcommand{\short}[1]{\hbox to 0.6em{\#1}}`

and then write `\short{f}(\short{x})`.

- The principle of longest match sometimes generates unexpected output. For example, `IMPORT Foo[+]` doesn’t work as expected, since it generates `IMPORT Foo␣␣` instead of `IMPORT Foo[+]`.

Solution: The desired output is obtained by adding a space: `IMPORT Foo[+]`.

- Problems with \wedge and $_$: If you try to define commands like `\SetSymbol{\wedge|\wedge}{...}` or `\SetSymbol{_|_}{...}`, you have to turn the \wedge or the $_$ into normal characters first. One solution is `\catcode'\wedge=12\SetSymbol{\wedge|\wedge}{...}\catcode'\wedge=7` and `\catcode'_ =12\SetSymbol{_|_}{...}\catcode'_ =8`.

A better way is to do the change of category only locally within a group. However, this would make the new command $\wedge|\wedge$ also only locally known. Hence, the solution is slightly more complex:

```
{ \catcode'\wedge=12
  \global\def\MyNewSymbol{\SetSymbol{\wedge|\wedge}{...}}
}\MyNewSymbol
```

- `\‘...’` does *not* work in environments, where `\‘` is used as a special macro (for instance in tabular environments).
Solution: Use `\Keyword{foo}`.
- It has not yet been tested, whether the fancy commands and symbols can be used with parameters.

The scanner appears to harmonize with most \TeX and \LaTeX features (and to be reasonably efficient⁸). However, there are known problems:

- \TeX constructs such as `\hbox to 12pt{...}` do not work.
Reason: In order to properly cope with macro parameters and with sub- or superscripts, the scanner has to enclose many tokens into parentheses `{...}`. This is, however, forbidden for things like “12pt”.
Solution: Define an auxiliary command `\def\mybox{\hbox to 12 pt}` outside of the program and use `\mybox{...}` inside.
- An `\end{...}` command cannot be used inside programs.
Reason: The scanner looks for `\end` in order to stop at `\end{program}`
Solution: Same as above.
- If you write `$. . $` inside a program, the first `$` switches the scanner and math mode *off*! The second `$` switches math mode on again, but not the scanner (unless you have said `\ActivateMathScan`). At the following line the scanner is switched on again.
If `\ActivateMathScan` is set, then the situation `$. . $` simply switches *off* scanning *between* the two `$` symbols.

⁸The scanner works on a line-by-line basis.

Appendix: Tables of Commands

Command	effect
<code>\Keyword{...}</code>	keyword (default: <code>\textsc</code>)
<code>\‘...’</code>	keyword (default: <code>\textsc</code>)
<code>\Identifier{...}</code>	identifier (default: <code>\mathtt</code>)
<code>\Grapheme{...}</code>	grapheme (default: <code>\texttt</code>)
<code>\Digit{...}</code>	digit (default: taken as is)
<code>\Comment {...}</code>	comment (default: <code>\textit</code>)
<code>\IdentifierFont{...}</code>	sets the font for identifiers (default <code>\tt</code>)
<code>\DigitFont{...}</code>	sets the font for digits (default <code>\tt</code>)
<code>\Language{...}</code>	language name (e.g. <code>\Language{Haskell}</code>)
<code>\DefLanguage{...}</code>	define command for a language name
<code>\Opal</code>	the language name OPAL
<code>\EnableLisp</code>	allow Lisp-like identifiers (e.g. <code>query-replace</code>)
<code>\DisableLisp</code>	disallow Lisp-like identifiers (default)
<code>\ActivateMathScan</code>	make the special symbols available for math mode
<code>\EnableMathScan</code>	use the special symbols in <code>...\$</code>
<code>\DisableMathScan</code>	don’t use the special symbols in <code>...\$</code>
<code>\SetCommand</code>	define “fancy” command
<code>\SetSymbol</code>	define “fancy” command (for math symbols)
<code>\SetAmsSymbol</code>	define “fancy” command (for ams symbols)
<code>\UnSetSymbol</code>	undefine “fancy” command
<code>\SetMacro</code>	define “fancy” command for use in <code>\program</code> environment

Table 1: Miscellaneous commands (mostly `scanner.sty`)

Command	effect
<code>\begin{program}[pcnf]</code> <code>\end{program}</code> <code>\begin{opal}[pcnf]</code> <code>\end{opal}</code> <code>\ProgramPrelude</code> <code>\OpalPrelude</code>	beginning of program environment end of program environment beginning of opal environment end of opal environment executed at beginning of program environment executed at beginning of opal environment
<code>\ProgramSkip{...}</code> <code>\ProgramWidth{...}</code> <code>\ProgramLeftIndent{...}</code> <code>\ProgramRightIndent{...}</code> <code>\ProgramInnerIndent{...}</code> <code>\EmptyLineSkip{...}</code>	set vertical distance from surrounding paragraphs set width of programs set left indentation of programs set right indentation of programs set indentation for main keywords set vertical space for empty lines
<code>\begin{embeddedProgram}</code> <code>\end{embeddedProgram}</code> <code>\(...\)</code> <code>\EmbeddedProgramPrelude</code>	begin embeddedProgram environment end of embeddedProgram environment embedded program environment executed at beginning of embedded programs
<code>\[</code> <code>\&</code> <code>\]</code> <code>\Break</code> <code>\Rule</code> <code>-----</code> <code>\O\[...]</code> <code>\Shrink{...}</code>	open tabular environment separation of columns close tabular environment enable page break here horizontal rule horizontal rule set width of following tabular environment to zero set width of following fragment to zero
<code>\-</code> <code>--</code> <code>\UaCmnt</code> <code>\%</code> <code>\Comment</code>	aligned program comment (till end of line) unaligned program comment (till end of line) unaligned program comment (till end of line) TeX comment (including end of line) layout of comments
<code>\lnr</code> <code>\Number</code> <code>\NumberWidth{...}</code>	number this line layout of line numbers width for line numbers

Table 2: Layout commands (**basicprog.sty**)

Layout command	just the keyword	defined by ...
EXTERNAL	\EXTERNAL	\TopKeyWord
IMPLEMENTATION	\IMPLEMENTATION	\TopKeyWord
INTERFACE	\INTERFACE	\TopKeyWord
INTERNAL	\INTERNAL	\TopKeyWord
MODULE	\MODULE	\TopKeyWord
SIGNATURE	\SIGNATURE	\TopKeyWord
SPECIFICATION	\SPECIFICATION	\TopKeyWord
STRUCTURE	\STRUCTURE	\TopKeyWord
THEORY	\THEORY	\TopKeyWord
ASSERT	\ASSERT	\MainKeyWord
AXM	\AXM	\MainKeyWord
DATA	\DATA	\MainKeyWord
DEF	\DEF	\MainKeyWord
FUN	\FUN	\MainKeyWord
IMPORT	\IMPORT	\MainKeyWord
INHERIT	\INHERIT	\MainKeyWord
LAW	\LAW	\MainKeyWord
PRED	\PRED	\MainKeyWord
REALIZES	\REALIZES	\MainKeyWord
REQUIRE	\REQUIRE	\MainKeyWord
SORT	\SORT	\MainKeyWord
SPC	\SPC	\MainKeyWord
SPEC	\SPEC	\MainKeyWord
THM	\THM	\MainKeyWord
TYPE	\TYPE	\MainKeyWord
COMPLETELY	\COMPLETELY	\FollowKeyWord[1em]
ONLY	\ONLY	\FollowKeyWord
LET	\LET	\BeginKeyWord
IF	\IF	\IfLikeKeyWord
OTHERWISE	\OTHERWISE	\IfLikeKeyWord[1em]
THEN	\THEN	\ThenLikeKeyWord
PRE	\PRE	\ThenLikeKeyWord
POST	\POST	\ThenLikeKeyWord

Table 3: Predefined keywords in opal2x.sty [part 1]

ELSE	\ELSE	\ElseLikeKeyWord
IN	\IN	\ElseLikeKeyWord
FI	\FI	\EndKeyWord
ALL	\ALL	\PlainKeyWord
AND	\AND	\PlainKeyWord
ANDIF	\ANDIF	\PlainKeyWord
ANY	\ANY	\PlainKeyWord
AS	\AS	\PlainKeyWord
DERIVE	\DERIVE	\PlainKeyWord
DFD	\DFD	\PlainKeyWord
DISCRIMINATORS	\DISCRIMINATORS	\PlainKeyWord
EX	\EX	\PlainKeyWord
FIX	\FIX	\PlainKeyWord
IMPLIES	\IMPLIES	\PlainKeyWord
INJECTIONS	\INJECTIONS	\PlainKeyWord
LEFTASSOC	\LEFTASSOC	\PlainKeyWord
NOT	\NOT	\PlainKeyWord
OR	\OR	\PlainKeyWord
ORIF	\ORIF	\PlainKeyWord
PRIORITY	\PRIORITY	\PlainKeyWord
PROPERTIES	\PROPERTIES	\PlainKeyWord
RIGHTASSOC	\RIGHTASSOC	\PlainKeyWord
SELECTORS	\SELECTORS	\PlainKeyWord
THE	\THE	\PlainKeyWord
UNIQ	\UNIQ	\PlainKeyWord
WHERE	\WHERE	\PlainKeyWord

Table 4: Predefined keywords in `opal2x.sty` [part 2]

Command	effect
\TopKeyWord[...]{...}	define topmost keyword (e.g. STRUCTURE)
\MainKeyWord[...]{...}	define main keyword (e.g. FUN)
\FollowKeyWord[...]{...}	define follow-up keyword (e.g. ONLY)
\BeginKeyWord[...]{...}	define begin keyword (e.g. LET)
\IfLikeKeyWord[...]{...}	define if-like keyword (e.g. IF)
\ThenLikeKeyWord[...]{...}	define then-like keyword (e.g. THEN)
\ElseLikeKeyWord[...]{...}	define else-like keyword (e.g. ELSE)
\EndKeyWord{...}	define end keyword (e.g. FI)
\PlainKeyWord[...]{...}	define plain keyword (e.g. AS)
\Keyword{...}	keyword
\‘...’	keyword

Table 5: Commands for defining new keywords (`basicprog.sty`)

<i>look</i>	<i>type</i>	<i>look</i>	<i>type</i>
a^n	$a^{\{n\}}$	a_n	$a_{\{n\}}$
a^3	a^3	a_1	a_1
nat^{Nat}	nat^{Nat}	$(- + -)$	$(- + -)$

Table 6: Examples for super- and subsetting (`scanner.sty`)

#	^	_	&	@	%
---	---	---	---	---	---

Table 7: Normalized symbols (`opal2x.sty`)

<code>\Ampersand</code>	$\&$	<code>\Compose</code>	\circ	<code>\Box</code>	\square
<code>\At</code>	$\@$	<code>\Concat</code>	⋈	<code>\Diamond</code>	
<code>\Backslash</code>	\backslash	<code>\Disjoint</code>	$\not\cap$	<code>\BoxBar</code>	\boxplus
<code>\Dollar</code>	$\$$	<code>\SymDiff</code>	$\dot{-}$	<code>\BoxMinus</code>	\boxminus
<code>\DoubleQuote</code>	''	<code>\SemOpen</code>	\llbracket	<code>\BoxPlus</code>	\boxplus
<code>\Hashmark</code>	$\#$	<code>\SemClose</code>	\rrbracket	<code>\BoxTimes</code>	\boxtimes
<code>\Percent</code>	$\%$			<code>\BoxDot</code>	\boxdot

Table 8: Auxiliary commands for certain symbols

<i>look</i>	<i>type</i>	<i>-ams</i>	<i>look</i>	<i>type</i>	<i>-ams</i>	<i>look</i>	<i>type</i>	<i>-ams</i>
\leq	<code>=<</code>		\Rightarrow	<code>=></code>		\rightarrow	<code>-></code>	
\geq	<code>>=</code>		\Leftarrow	<code><=</code>		\leftarrow	<code><-</code>	
\ll	<code><<</code>		\Leftrightarrow	<code><=></code>		\leftrightarrow	<code><-></code>	
\gg	<code>>></code>		\Longrightarrow	<code>==></code>		\longrightarrow	<code>--></code>	
\leqslant	<code><.</code>		\Longleftarrow	<code><==</code>		\longleftarrow	<code><--</code>	
\gtrless	<code>>.</code>		\Leftrightarrow	<code><==></code>		\longleftrightarrow	<code><--></code>	
$\dot{\leq}$	<code>.></code>		\wedge	<code>/_</code>	(*)	\rightrightarrows	<code>>-></code>	\rightrightarrows
$\dot{=}$	<code>=.</code>		\vee	<code>_/</code>	(*)	\leftrightsquigarrow	<code><-<</code>	\leftrightsquigarrow
\nless	<code>/<</code>	\nless	\neg	<code>-;</code>		\rightarrowtail	<code>->></code>	\rightarrowtail
\ngtr	<code>/></code>	\ngtr	\forall	<code>\All</code>	(*)	\leftarrowtail	<code><<-</code>	\leftarrowtail
\nlessgtr	<code>/=<</code>	\nlessgtr	\forall	<code>\A</code>	(*)	\uparrow	<code> ^</code>	
\ngtrless	<code>/>=</code>	\ngtrless	\exists	<code>\Ex</code>	(*)	\mapsto	<code> -></code>	
\neq	<code>/=</code>		\exists	<code>\E</code>	(*)	\longmapsto	<code> --></code>	
\equiv	<code>==</code>		\vdash	<code> -</code>		$\dot{-}$	<code>-.</code>	
\ncong	<code>=/=</code>		\models	<code> =</code>		\div	<code>-:-</code>	
\parallel	<code> </code>		\square	<code>[]</code>	\square	\times	<code>**</code>	
\llbracket	<code> [</code>		\diamond	<code><></code>		\times	<code>><</code>	
\rrbracket	<code>] </code>		\triangleleft	<code>< </code>		\pm	<code>+-</code>	
\langle	<code>\<</code>	(*)	\triangleright	<code> ></code>		\mp	<code>++</code>	
\rangle	<code>\></code>	(*)	\boxdot	<code>[.]</code>	\boxdot	$::$	<code>::</code>	
$/$	<code>/</code>		\boxplus	<code>[]</code>		\circ	<code>\o</code>	(*)
$:$	<code>:</code>		\boxminus	<code>[-]</code>	\boxminus	λ	<code>\l</code>	(*)
\dots	<code>...</code>		\boxtimes	<code>[+]</code>	\boxtimes	\mathbb{B}	<code>BB</code>	\mathbb{B}
$"$	<code>"</code>		\boxdot	<code>[*]</code>	\boxdot	\mathbb{N}	<code>NN</code>	\mathbb{N}
$'$	<code>'</code>		\sqcup	<code> _ </code>		\mathbb{Z}	<code>ZZ</code>	\mathbb{Z}
\wedge	<code>^</code>	(*)	\perp	<code>- _</code>		\mathbb{Q}	<code>QQ</code>	\mathbb{Q}
$-$	<code>-</code>	(*)	\cup	<code>\U</code>	(*)	\mathbb{R}	<code>RR</code>	\mathbb{R}
\sqcup	<code>~</code>	(*)	\cap	<code>\I</code>	(*)	\mathbb{V}	<code>VV</code>	\mathbb{V}

Table 9: Special symbols (`symbols.sty`)

The “look” columns show the printed output, the “type” columns show, what you have to type, and the “no-ams” columns show the alternative outputs, when `amssymb` is not loaded; an (*) in this column means that the symbol is *not* introduced (and thus not changeable) by `\SetSymbol`.