The syntax* package

Mark Wooding

17 May 1996

Contents

*The syntax package is currently at version 1.07, dated 17 May 1996.

1 User guide

1.1 Introduction

The syntax package provides a number of commands and environments which extend LaTeX and allow you to typeset good expositions of syntax.

The package provides several different types of features: probably not all of these will be required by every document which needs the package:

- A system of abbreviated forms for typesetting syntactic items.
- An environment for typesetting BNF-type grammars
- A collection of environments for building syntax diagrams.

The package also includes some other features which, while not necessarily syntax-related, will probably come in handy for similar types of document:

- An abbreviated notation for verbatim text, similar to the shortvrb package.
- A slightly different underscore character, which works as expected in text and maths modes.

1.2 The abbreviated verbatim notation

In documents describing programming languages and libraries, it can become tedious to type `\verb|...|` every time. Like Frank Mittelbach's shortvrb package, syntax provides a way of setting up single-character abbreviations. The only real difference between the two is that the declarations provided by syntax obey LaTeX's normal scoping rules.

`\shortverb` You can set up a character as a 'verbatim shorthand' character using the `\shortverb` command. This takes a single argument, which should be a single-character control sequence containing the character you want to use. So, for example, the command

```
\shortverb{\|}
```

would set up the '|' character to act as a verbatim delimiter. While a `\shortverb` declaration is in force, any text surrounded by (in this case) vertical bar characters will be typeset as if using the normal `\verb` command.

`shortverb` Since LaTeX allows any declaration to be used as an environment, you can use a shortverb environment to delimit the text over which your character is active:

```
Some text...
\begin{shortverb}{\|}
...
\end{shortverb}
```

`\unverb` If you want to disable a `\shortverb` character without ending the scope of other declarations, you can use the `\unverb` command, passing it a character as a control sequence, in the same way as above.

The default TeX/LaTeX underscore character is rather too short for use in identifiers. For example:

Old-style underscores

Typing long underscore-filled names, like big_function_name, is normally tedious. The normal positioning of the underscore is wrong, too.

```
Typing long underscore-filled
names, like big\_function\_name,
is normally tedious.  The normal
positioning of the underscore
is wrong, too.
```

The syntax package redefines the `\_` command to draw a more attractive underscore character. It also allows you to use the `_` character directly to produce an underscore outside of maths mode: `_` behaves as a subscript character as usual inside maths mode.

New syntax underscores

You can use underscore-filled names, like big_function_name, simply and naturally. Of course, subscripts still work normally in maths mode, e.g., x_i.

```
You can use underscore-filled
names, like big_function_name,
simply and naturally.  Of
course, subscripts still work
normally in maths mode, e.g.,
$x_i$.
```

1.3 Typesetting syntactic items

The syntax package provides some simple commands for typesetting syntactic items.

`\synt` Typing `\synt{`⟨*text*⟩`}` typesets ⟨*text*⟩ as a 'non-terminal', in italics and surrounded by angle brackets. If you use `\synt` a lot, you can use the incantation

```
\def\<#1>{\synt{#1}}
```

to allow you to type `\<`⟨*text*⟩`>` as an alternative to `\synt{`⟨*text*⟩`}`.

`\lit` You can also display literal text, which the reader should type directly, using the `\lit` command.

Use of `\lit`

Type '`ls`' to display a list of files.

```
Type \lit{ls} to display a
list of files.
```

Note that the literal text appears in quotes. To suppress the quotes, use the '`*`' variant.

The `\lit` command produces slightly better output than `\verb` for running text, since the spaces are somewhat narrower. However, `\verb` allows you to type arbitrary characters, which are treated literally, whereas you must use commands such as `\{` to use special characters within the argument to `\lit`. Of course, you can use `\lit` anywhere in the document: `\verb` mustn't be used inside a command argument.

1.4 Abbreviated forms for syntactic items

It would be very tedious to require the use of commands like `\synt` when building syntax descriptions like BNF grammars. It would also make your LaTeX source

hard to read. Therefore, syntax provides some abbreviated forms which make typesetting syntax quicker and easier.

Since the abbreviated forms use several characters which you may want to use in normal text, they aren't enabled by default. They only work with special commands and environments provided by the syntax package.

The abbreviated forms are shown in the table below:

Input	Output
`<some text>`	⟨*some text*⟩
`‘some text’`	‘some text’
`"some text"`	some text

Within one of these abbreviated forms, text is treated more-or-less verbatim:

- Any `$`, `%`, `^`, `&`, `{`, `}`, `~` or `#` characters are treated literally: their normal special meanings are ignored.
- Other special characters, with the exception of `\`, are also treated literally: this includes any characters made special by `\shortverb`.

However, the `\` character retains its meaning. Since the brace characters are not recognised, most commands can't be used within abbreviated forms. However, you can use special commands to type some of the remaining special characters:

Command	Result
`\\`	A ‘\’ character
`\>`	A ‘>’ character
`\’`	A ‘’’ character
`\"`	A ‘"’ character
`\␣`	A ‘␣’ character (not a space)

Note that `\\`, `\>`, `\"` and `\␣` are only useful in a `\tt` font, i.e., inside `‘...’` and `"..."` forms, since the characters don't exist in normal fonts. The `\>`, `\"` and `\’` commands are only provided so you can use these characters within `<...>`, `"..."` and `‘...’` forms respectively: in the other forms, there is no need to use the special command.

In addition, when the above abbreviations are enabled, the character `|` is set to typeset a | symbol, which is conventionally used to separate alternatives in syntax descriptions.

`\syntax` Normally, these abbreviated forms are enabled only within special environments, such as grammar and syntdiag. To use them in running text, use the `\syntax` command. The abbreviations are made active within the argument of the `\syntax` command.[1] Note that you cannot use the `\syntax` command within the argument of another command.

`\synshorts` `synshorts` You can also enable the syntax shortcuts using the `\synshorts` declaration or the synshorts environment. This enables the syntax shortcuts until the scope of the declaration ends.

`\synshortsoff` If syntax shortcuts are enabled, you can disable them using the `\synshortsoff`

[1] The argument of the `\syntax` command may contain commands such as `\verb`, which are normally not allowed within arguments.

declaration.

1.5 The grammar environment

grammar

For typesetting formal grammars, for example, of programming languages, the syntax package provides a grammar environment. Within this environment, the abbreviated forms described above are enabled.

Within the environment, separate production rules should be separated by blank lines. You can use the normal \\ command to perform line-breaking of a production rule. Note that a production rule must begin with a nonterminal name enclosed in angle brackets (< ... >), followed by whitespace, then some kind of production operator (usually '::=') and then some more whitespace. You can control how this text is actually typeset, however.

\[[\]]

You can use syntax diagrams (see below) instead of a straight piece of BNF by enclosing it in a \[[... \]] pair. Note that you can't mix syntax diagrams and BNF in a production rule, and you will get something which looks very strange if you try.

\alt

In addition, a command \alt is provided for splitting long production rules over several lines: the \alt command starts a new line and places a | character slightly in the left margin. This is useful when a symbol has many alternative productions.

The grammar environment

⟨*statement*⟩ ::= ⟨*ident*⟩ '=' ⟨*expr*⟩
| 'for' ⟨*ident*⟩ '=' ⟨*expr*⟩ 'to' ⟨*expr*⟩ 'do' ⟨*statement*⟩
| '{' ⟨*stat-list*⟩ '}'
| ⟨*empty*⟩

⟨*stat-list*⟩ ::= ⟨*statement*⟩ ';' ⟨*stat-list*⟩ | ⟨*statement*⟩

```
\begin{grammar}
<statement> ::= <ident> ‘=’ <expr>
  \alt ‘for’ <ident> ‘=’ <expr> ‘to’ <expr> ‘do’ <statement>
  \alt ‘{’ <stat-list> ‘}’
  \alt <empty>

<stat-list> ::= <statement> ‘;’ <stat-list> | <statement>
\end{grammar}
```

You can modify the appearance of grammars using three length parameters:

\grammarparsep is the amount of space inserted between production rules. It is a rubber length whose default value is 8 pt, with 1 pt of stretch and shrink.

\grammarindent is the amount by which the right hand side of a production rule is indented from the left margin. It is a rigid length. Its default value is 2 em.

\grammarlabel

You can also control how the 'label' is typeset by redefining the \grammarlabel

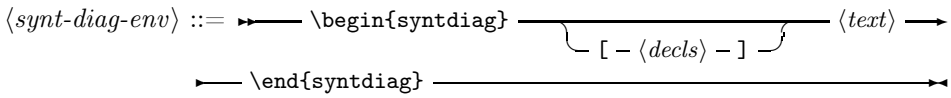
command. The command is given two arguments: the name of the nonterminal (which was enclosed in angle brackets), and the 'production operator'. The command is expected to produce the label. By default, it typesets the nonterminal name using `\synt` and the operator at opposite ends of the label, separated by an `\hfill`.

1.6 Syntax diagrams

A full formal BNF grammar can be somewhat overwhelming for less technical readers. Documents aimed at such readers tend to display grammatical structures as *syntax diagrams*.

`syntdiag` A syntax diagram is always enclosed in a syntdiag environment. You should think of the environment as enclosing a new sort of LaTeX mode: trying to type normal text into a syntax diagram will result in very ugly output. LaTeX ignores spaces and return characters while in syntax diagram mode.

The syntax of the environment is very simple:

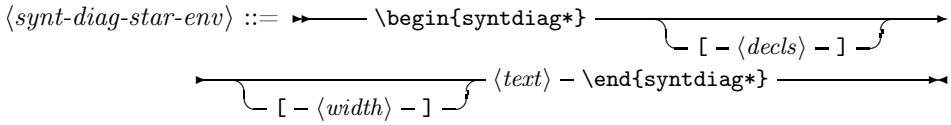


The ⟨*decls*⟩ contain any declarations you want to insert, to control the environment. The parameters to tweak are described below.

Within a syntax diagram, you can include syntactic items using the abbreviated forms described elsewhere. The output from these forms is modified slightly in syntax diagram mode so that the diagram looks right.

I probably ought to point out now that the syntax diagram typesetting commands produce beautiful-looking diagrams with all the rules and curves accurately positioned. Some device drivers don't position these objects correctly in their output. I've had particular trouble with `dvips`. I'll say it again: it's not my fault!

`syntdiag*` The syntdiag environment only works in paragraph mode, and it acts rather like a paragraph, splitting over several lines when appropriate. If you just want to typeset a snippet of a syntax diagram, you can use the starred environment syntdiag∗.

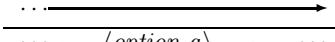
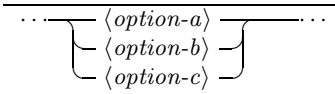
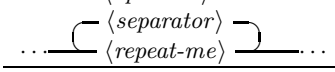


When typesetting little demos like this, it's not normal to fully adorn the syntax diagram with the full double arrows ('⟨ ⋯ ⟩'). The two declarations `\left{`⟨*arrow*⟩`}` and `\right{`⟨*arrow*⟩`}` allow you to choose the arrows on each side of the syntax diagram snippet. The possible values of ⟨*arrow*⟩ are shown in the table-ette below:

»-	⟨	>-	⟨	->	⟨
-><	⟩	...	⋯	-	

These declarations should be used only in the optional argument to the syntdiag∗ command. The second optional argument to the environment, if specified, fixes the width of the syntax diagram snippet; if you omit this argument, the diagram is made just wide enough to fit everything in.

Example of syntdiag*

Construction	Meaning
	Start of syntax diagram
	End of syntax diagram
	Continued on next line
	Continued from previous line
	Alternatives: choose any one
	One or more items, with separators

```
\newcommand{\bs}[2]{%
  \begin{minipage}{1.6in}%
  \begin{syntdiag*}[\left{#1}\right{#2}][1.6in]%
}
\newcommand{\es}{\end{syntdiag*}\end{minipage}}

\begin{center}
\begin{tabular}{cl}                                          \\ \hline
\bf Construction    & \bf Meaning                            \\ \hline
\bs {>>-} {...} \es & Start of syntax diagram                \\
\bs {...} {-><} \es & End of syntax diagram                  \\
\bs {>-}  {...} \es & Continued on next line                 \\
\bs {...} {->}  \es & Continued from previous line           \\ \hline
\bs {...} {...}
  \begin{stack} <option-a> \\ <option-b> \\ <option-c> \end{stack}
\es                 & Alternatives: choose any one           \\
\bs {...} {...}
  \begin{rep} <repeat-me> \\ <separator> \end{rep}
\es                 & One or more items, with separators \\ \hline
\end{tabular}
\end{center}
```

\tok You can also include text using the \tok command. The argument of this command is typeset in LaTeX's LR mode and inserted into the diagram. Syntax abbreviations are allowed within the argument, so you can, for example, include textual descriptions like

```
\tok{any <char> except `"'}
```

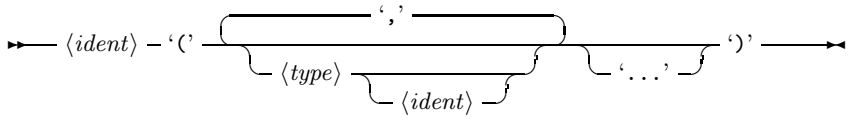
stack Within a syntax diagram, a choice between several different items is shown by stacking the alternatives vertically. In LaTeX, this is done by enclosing the items in a stack environment. Each individual item is separated by \\ commands, as in the array and tabular environments. Each row may contain any syntax diagram material, including \tok commands and other stack environments.

Note if you end a stack environment with a \\ command, a blank row is added to the bottom of the stack, indicating that none of the items need be specified.

rep Text which can be repeated is enclosed in a rep environment: the text is displayed with a backwards pointing arrow drawn over it, showing that it may be repeated. Optionally, you can specify text to be displayed in the arrow, separating it from the main text with a \\ command.

Note that items on the backwards arrow of a rep construction should be displayed *backwards*. You must put the individual items in reverse order when building this part of your diagrams. syntax will correctly reverse the arrows on rep structures, but apart from this, you must cope on your own. You are recommended to keep these parts of your diagrams as simple as possible to avoid confusing readers.

A syntax diagram




```
\begin{syntdiag}
<ident> `('
  \begin{rep} \begin{stack} \\
    <type> \begin{stack} \\ <ident> \end{stack}
  \end{stack} \\ `,' \end{rep}
\begin{stack} \\ `...' \end{stack} `)'
\end{syntdiag}
```

1.6.1 Line breaking in syntax diagrams

Syntax diagrams are automatically broken over lines and across pages. Lines are only broken between items on the outermost level of the diagram: i.e., not within stack or rep environments.

You can force a line break at a particular place by using the \\ command as usual. This supports all the usual LaTeX features: a '*' variant which prohibits page breaking, and an optional argument specifying the extra vertical space between lines.

1.6.2 Customising syntax diagrams

There are two basic styles of syntax diagrams supported:

square Lines in the syntax diagram join at squared-off corners. This appears to be the standard way of displaying syntax diagrams in IBM manuals, and most other documents I've seen.

rounded Lines curve around corners. Also, no arrows are drawn around repeating loops: the curving of the lines provides this information instead. This style is used in various texts on Pascal, and appears to be more popular in academic circles.

You can specify the style you want to use for syntax diagrams by giving the style name as an option on the `\usepackage` command. For example, to force rounded edges to be used, you could say

```
\usepackage[rounded]{syntax}
```

`\sdsize` `\sdlengths` The syntdiag environment takes an option argument, which should contain declarations which are obeyed while the environment is set up. The default value of this argument is '`\sdsize\sdlengths`'. The `\sdsize` command sets the default type size for the environment: this is normally `\small`. `\sdlengths` sets the values of the length parameters used by the environment based on the current text size. These parameters are described below.

For example, if you wanted to reduce the type size of the diagrams still further, you could use the command

```
\begin{syntdiag}[\tiny\sdlengths]
```

The following length parameters may be altered:

`\sdstartspace` The length of the rule between the arrows which begin each line of the syntax diagram and the first item on the line. Note that most objects have some space on either side of them as well. This is a rubber length. Its default value is 1 em, although it can shrink by up to 10 pt.

`\sdendspace` The length of the rule between the last item on a line and the arrow at the very end. Note that the final line also has extra rubber space on the end. This is a rubber length. Its default value is 1 em, although it will shrink by up to 10 pt.

`\sdmidskip` The length of the rule on either side of a large construction (either a stack or a rep). It is a rubber length. Its default value is $\frac{1}{2}$ em, with a very small amount of infinite stretch.

`\sdtokskip` The length of the rule on either side of a `\tok` item or syntax abbreviation. It is a rubber length. Its default value is $\frac{1}{4}$ em, with a very small amount of inifnite stretch.

`\sdfinalskip` The length of the rule which finishes the last line of a syntax diagram. It is a rubber length. Its default value is $\frac{1}{2}$ em, with 10000 fil of stretch, which will left-align the items on the line.[2]

[2] This is a little TEXnical. The idea is that if a stray 1 fil of stretch is added to the end of the line, it won't be noticed. However, the alignment of the text on the line can still be modified using `\sd@rule\hfill`, if you're feeling brave.

`\sdrulewidth` Half the width of the rules used in the diagram. It is a rigid length. Its default value is 0.2 pt.

`\sdcirclediam` The diameter of the circle from which the quadrants used in rounded-style diagrams are taken. This must be a multiple of 4 pt, or else the lines on the diagram won't match up.

In addition, you should call `\sdsetstrut` passing it the total height (height + depth) of a normal line of text at the current size. Normally, the value of `\baselineskip` will be appropriate.

You can also alter the appearance of stacks and reps by using their optional positioning arguments. By default, stacks descend below the main line of the diagram, and reps extend above it. Specifying an optional argument of `[b]` for either environment reverses this, putting stacks above and reps below the line.

1.7 Changing the presentation styles

You can change the way in which the syntax items are typeset by altering some simple commands (using `\renewcommand`). Each item (nonterminals, as typeset by `\synt`, and quoted and unquoted terminals, as typeset by `\lit` and `\lit*`) has two style commands associated with it, as shown in the table below.

Syntax item	Left command	Right command
Nonterminals	`\syntleft`	`\syntright`
Quoted terminals	`\litleft`	`\litright`
Unquoted terminals	`\ulitleft`	`\ulitright`

It's not too hard to see how this works. For example, if you look at the implementation for `\syntleft` and `\syntright` in the implementation section, you'll notice that they're defined like this:

```
\newcommand{\syntleft}{$\langle$\normalfont\itshape}
\newcommand{\syntright}{$\rangle$}
```

I think this is fairly simple, if you understand things like font changing.

Note that changing these style commands alters the appearance of all syntax objects of the appropriate types, as created by the `\synt` and `\lit` commands, in grammar environments, and in syntax diagrams.

2 Change history

Version 1.07

- Fixed problem with underscore hacking in a tabbing environment.

Version 1.06

- Added style hooks for syntax items.
- Improved colour handling in syntax diagrams, thanks to the `\doafter` package.

- Fixed some nasty bugs in the grammar environment which confused other lists and ruined the spacing. The grammar handling is now much tidier in general.

Version 1.05

- Fixed 'the bug' in the syntax diagram typesetting. It now breaks lines almost psychically, and doesn't break in the wrong places.
- Almost rewrote the grammar environment. It now does lots of the list handling itself, to allow more versatile typesetting of the left hand sides. There's lots of evil in there now.
- Added some more configurability. In particular, two new settings have been added to control grammar environments, and a neat way of adding new syntax diagram structures has been introduced.

Version 1.04

- Changed the vertical positioning of the rules, to make all the text line up properly. While the old version was elegant and simple, it had the drawback of looking nasty.
- Allow line breaks at underscores, but don't if there's another one afterwards. Also, prevent losing following space if underscore is written to a file.

Version 1.02

- Added support for rounded corners in syntax diagrams.
- Changed lots of `\hskip` commands to `\kern`s, to prevent possible line breaks.

Version 1.01

- Allowed disabling of underscore active character, to avoid messing up filenames.
- Added `\grammarparsep` and `\grammarindent` length parameters to control the appearance of grammars.

3 Implementation of syntax

1 ⟨*package⟩

3.1 Options handling

We define all the options we know about, and then see what's been put on the usepackage line.

The options we provide currently are as follows:

rounded draws neatly rounded edges on the diagram.

square draws squared-off edges on the diagram. This is the default.

nounderscore disables the undescore active character, The _ command still produces the nice version created here.

```
\DeclareOption{rounded}{\sd@roundtrue}
\DeclareOption{square}{\sd@roundfalse}
\DeclareOption{nounderscore}{\@uscorefalse}
```

Now process the options:

```
\newif\ifsd@round
\newif\if@uscore\@uscoretrue
\ExecuteOptions{square}
\ProcessOptions
```

3.2 Special character handling

A lot of the syntax package requires the use special active characters. These must be added to two lists: \dospecials, which is used by \verb and friends, and \@sanitize, which is used by \index. The two macros here, \addspecial and \remspecial, provide these registration facilities.

Two similar macros are found in Frank Mittelbach's doc package: these have the disadvantage of global operation. My macros here are based on Frank's, which in turn appear to be based on Donald Knuth's list handling code presented in Appendix D of *The TEXbook*.

Both these macros take a single argument: a single-character control sequence containing the special character to be added to or removed from the lists.

\addspecial This is reasonably straightforward. We remove the sequence from the lists, in case it's already there, and add it in in the obvious way. This requires a little bit of fun with \expandafter.

```
\def\addspecial#1{%
  \remspecial{#1}%
  \expandafter\def\expandafter\dospecials\expandafter{\dospecials\do#1}%
  \expandafter\def\expandafter\@santize\expandafter{%
    \@sanitize\@makeother#1}%
}
```

\remspecial This is the difficult bit. Since \dospecials and \@sanitize have the form of list macros, we can redefine \do and \@makeother to do the job for us. We must be careful to put the old meaning of \@makeother back. The current implementation assumes it knows what \@makeother does.

```
\def\remspecial#1{%
  \def\do##1{\ifnum`#1=`##1 \else\noexpand\do\noexpand##1\fi}%
  \edef\dospecials{\dospecials}%
  \def\@makeother##1{\ifnum`#1=`##1 \else%
    \noexpand\@makeother\noexpand##1\fi}%
  \edef\@sanitize{\@sanitize}%
  \def\@makeother##1{\catcode`##112}%
}
```

3.3 Underscore handling

When typing a lot of identifiers, it can be irksome to have to escape all '_' characters in the manuscript. We make the underscore character active, so that it typesets an underscore in horizontal mode, and does its usual job as a subscript operator in maths mode. Underscore must already be in the special character lists, because of its use as a subscript character, so this doesn't cause us a problem.

`\underscore` The `\underscore` macro typesets an underline character, using a horizontal rule. This is positioned slightly below the baseline, and is also slightly wider than the default TEX underscore. This code is based on a similar implementation found in the lgrind package.

```
\def\underscore{%
  \leavevmode%
  \kern.06em%
  \vbox{%
    \hrule\@width.6em\@depth.4ex\@height-.34ex%
  }%
  \ifdim\fontdimen\@ne\font=\z@%
    \kern.06em%
  \fi%
}
```

`\@uscore` This macro is called by the '_' active character to sort out what to do.

If this is maths mode, we use the `\sb` macro, which is already defined to do subscripting. Otherwise, we call `\textunderscore`, which picks the nicest underscore it can find.

There's some extra cunningness here, because I'd like to be able to hyphenate after underscores usually, but not when there's another one following. And then, because tabbing redefines `\_`, there's some more yukkiness to handle that: the usual `\@tabacckludge` mechanism doesn't cope with this particular case.

```
\let\usc@builtindischyphen\-
\def\@uscore.{%
  \ifmmode%
    \expandafter\@firstoftwo%
  \else%
    \expandafter\@secondoftwo%
  \fi%
  \sb%
  {\textunderscore\@ifnextchar_{}{\usc@builtindischyphen}}%
}
```

Now we set up the active character. Note the `\protect`, which makes underscores work reasonably well in moving arguments. Note also the way we end with a some funny stuff to prevent spaces being lost if this is written to a file.

```
\if@uscore
  \AtBeginDocument{%
    \catcode`\_\active%
    \begingroup%
    \lccode`\~`\_%
    \lowercase{\endgroup\def~{\protect\@uscore.}}%
  }
\fi
```

Finally, we redefine the _ macro to use our own \underscore, because it's prettier. Actually, we don't: we just redefine the \?\textunderscore command (funny name, isn't it?).

```
\expandafter\let\csname?\string\textunderscore\endcsname\underscore
```

3.4 Abbreviated verbatim notation

In similar style to the doc package, we allow the user to set up characters which delimit verbatim text. Unlike doc, we make such changes local to the current group. This is performed through the \shortverb and \unverb commands.

The implementations of these commands are based upon the \MakeShortVerb and \DeleteShortVerb commands of the doc package, although these versions have effect local to the current grouping level. This prevents their redefinition of \dospecials from interfering with the grammar shortcuts, which require local changes only.

The command \shortverb takes a single argument: a single-character control sequence defining which character to make into the verbatim text delimiter. We store the old meaning of the active character in a control sequence called \mn@\⟨*char*⟩. Note that this control sequence contains a backslash character, which is a little odd. We also define a command \cc@\⟨*char*⟩ which will return everything to normal. This is used by the \unverb command.

\shortverb Here we build the control sequences we need to make everything work nicely. The active character is defined via \lowercase, using the ~ character: this is already made active by TEX.

The actual code requires lots of fiddling with \expandafter and friends.

```
\def\shortverb#1{%
```

First, we check to see if the command \cc@\⟨*char*⟩ has been defined.

```
  \@ifundefined{cc@\string#1}{%
```

If it hasn't been defined, we add the character to the specials list.

```
    \addspecial#1%
```

Now we set our character to be the lowercase version of ~, which allows us to use it, even though we don't know what it is.

```
    \begingroup%
    \lccode`\~`#1%
```

Finally, we reach the tricky bit. All of this is lowercased, so any occurrences of ~ are replaced by the user's special character.

```
    \lowercase{%
      \endgroup%
```

We remember the current meaning of the character, in case it has one. We have to use \csname to build the rather strange name we use for this.

```
      \expandafter\let\csname mn@\string#1\endcsname~%
```

Now we build \cc@\⟨*char*⟩. This is done with \edef, since more of this needs to be expanded now than not. In this way, the actual macros we create end up being very short.

```
      \expandafter\edef\csname cc@\string#1\endcsname{%
```

First, add a command to restore the character's old catcode.

```
61        \catcode`\noexpand#1\the\catcode`#1%
```

Now we restore the character's old meaning, using the version we saved earlier.

```
62        \let\noexpand~\expandafter\noexpand%
63          \csname mn@\string#1\endcsname%
```

Now we remove the character from the specials lists.

```
64        \noexpand\remspecial\noexpand#1%
```

Finally, we delete this macro, so that `\unverb` will generate a warning if the character is `\unverb`ed again.

```
65        \let\csname cc@\string#1\endcsname\relax%
66      }%
```

All of that's over now. We set up the new definition of the character, in terms of `\verb`, and make the character active. The nasty `\syn@ttspace` is there to make the spacing come out right. It's all right really. Honest.

```
67      \def~{\verb~\syn@ttspace}%
68    }%
69    \catcode`#1\active%
```

If our magic control sequence already existed, we can assume that the character is already a verbatim delimiter, and raise a warning.

```
70  }{%
71    \PackageWarning{syntax}{Character `\expandafter\@gobble\string#1'
72                            is already a verbatim\MessageBreak
73                            delimiter}%
74  }%
75 }
```

`\unverb` This is actually terribly easy: we just use the `\cc@\`⟨*char*⟩ command we definied earlier, after making sure that it's been defined.

```
76 \def\unverb#1{%
77   \@ifundefined{cc@\string#1}{%
78     \PackageWarning{syntax}{Character `\expandafter\@gobble\string#1'
79                             is not a verbatim\MessageBreak
80                             delimiter}%
81   }{%
82     \csname cc@\string#1\endcsname%
83   }%
84 }
```

3.5 Style hooks for syntax forms

To allow the appearance of syntax things to be configured, we provide some redefinable bits.

The three types of objects (nonterminal symbols, and quoted and unquoted terminals) each have two macros associated with them: one which does the 'left' bit of the typesetting, and one which does the 'right' bit. The items are typeset as LR boxes. I'll be extra good while defining these hooks, so that it's obvious what's going on; macho TEX hacker things resume after this section.

\syntleft \syntright

I can't see why anyone would want to change the typesetting of nonterminals, although I'll provide the hooks for symmetry's sake.

```
\newcommand{\syntleft}{$\langle$\normalfont\itshape}
\newcommand{\syntright}{$\rangle$}
```

\ulitleft \ulitright \litleft \litright

Now we can define the left and right parts of quoted and unquoted terminals. US readers may want to put double quotes around the quoted terminals, for example.

```
\newcommand{\ulitleft}{\normalfont\ttfamily\syn@ttspace\frenchspacing}
\newcommand{\ulitright}{}
\newcommand{\litleft}{`\bgroup\ulitleft}
\newcommand{\litright}{\ulitright\egroup'}
```

3.6 Simple syntax typesetting

In general text, we allow access to our typesetting conventions through standard LaTeX commands.

\synt

The `\synt` macro typesets its argument as a syntactic quantity. It puts the text of the argument in italics, and sets angle brackets around it. Breaking of a `\synt` object across lines is forbidden.

```
\def\synt#1{\mbox{\syntleft{#1\/}\syntright}}
```

\lit

The `\lit` macro typesets its argument as literal text, to be typed in. Normally, this means setting the text in `\tt` font, and putting quotes around it, although the quotes can be suppressed by using the $*$-variant.

The `\syn@ttspace` macro sets up the spacing for the text nicely: `\tt` spaces tend to be a little wide.

```
\def\lit{\@ifstar{\lit@i\ulitleft\ulitright}{\lit@i\litleft\litright}}
\def\lit@i#1#2#3{\mbox{#1{#3\/}#2}}
```

\syn@ttspace

This sets up the `\spaceskip` value for `\tt` text.

```
\def\syn@ttspace@{\spaceskip.35em\@plus.2em\@minus.15em\relax}
```

However, this isn't always the right thing to do.

```
\def\ttthinspace{\let\syn@ttspace\syn@ttspace@}
\def\ttthickspace{\let\syn@ttspace\@empty}
```

I know what I like thoough.

```
\ttthinspace
```

3.6.1 The shortcuts

The easy part is over now. The next job is to set up the 'grammar shortcuts' which allow easy changing of styles.

We support four shortcuts:

- `'literal text'` typesets 'literal text'
- `<non-terminal>` typesets ⟨*non-terminal*⟩
- `"unquoted text"` typesets "unquoted text"

- | typesets a | character

These are all implemented through active characters, which are enabled using the `\syntaxShortcuts` macro, described below.

`\readupto` `\readupto{`⟨*char*⟩`}{`⟨*decls*⟩`}{`⟨*command*⟩`}` will read all characters up until the next occurrence of ⟨*char*⟩. Normally, all special characters will be deactivated. However, you can reactivate some characters, using the ⟨*decls*⟩ argument, which is processed before the text is read.

The code is borrowed fairly obviously from the LaTeX 2ε source for the `\verb` command.

```
\def\readupto#1#2#3{%
  \bgroup%
  \verb@eol@error%
  \let\do\@makeother\dospecials%
  #2%
  \catcode`#1\active%
  \lccode`\~`#1%
  \gdef\verb@balance@group{\verb@egroup%
     \@latex@error{\noexpand\verb illegal in command argument}\@ehc}%
  \def\@vhook{\verb@egroup#3}%
  \aftergroup\verb@balance@group%
  \lowercase{\let~\@vhook}%
}
```

`\syn@assist` The `\syn@assist` macro is used for defining three of the shortcuts. It is called as

`\syn@assist{`⟨*left-decls*⟩`}{`⟨*actives*⟩`}{`⟨*delimeter*⟩`}`
`{`⟨*right-decls*⟩`}{`⟨*end-cmd*⟩`}`

It creates an hbox, sets up the escape sequences for quoting our magic characters, and then typesets a box containing

⟨*left-decls*⟩`{`⟨*delimited-text*⟩`\/}`⟨*right-decls*⟩

The ⟨*left-decls*⟩ and ⟨*right-decls*⟩ can be `\relax` if they're not required.

The ⟨*actives*⟩ argument is passed to `\readupto`, to allow some special characters through. By default, we re-enable \, and make '␣' typeset some space glue, rather than a space character. A macro '\␣' is defined to actually print a space character, which yield '␣' in the '`\tt`' font.

Finally, it defines a `\ch` command, which, given a single-character control sequence as its argument, typesets the character. This is useful, since `‘` has been made active when we set up these calls, so the direct `\char‘\`⟨*char*⟩ doesn't work.

```
\def\syn@assist#1#2#3#4#5{%
```

First, we start the box, and open a group. We use `\mbox` because it does all the messing with `\leavevmode` which is needed.

```
  \mbox\bgroup%
```

Next job is to set up the escape sequences.

```
  \chardef\\`\\%
  \chardef\>`\>%
  \chardef\'`\'%
  \chardef\"`\"%
  \chardef\ `\ %
```

Now to define \ch. This is done the obvious way.

```
  \def\ch##1{\char`##1}%
```

For active characters, we do some fiddling with \lccodes.

```
  \def\act##1{%
    \catcode`##1\active%
    \begingroup%
    \lccode`\~`##1%
    \lowercase{\endgroup\def~}%
  }%
```

Finally, we do the real work of setting the text. We use \readupto to actually find the text we want.

```
  #1%
  \begingroup%
  \readupto#3{%
    \catcode`\\0%
    \catcode`\ 10%
    #2%
  }{%
    \/\endgroup#4\egroup#5%
  }%
}
```

\syn@shorts This macro actually defines the expansions for the active characters. We have to do this separately because ` must be active when we use it in the \def, but we can't do that and use \catcode at the same time. The arguments are commands to do before and after the actual command. These are passed up from \syntaxShortcuts.

All of the characters use \syn@assist in the obvious way except for |, which drops into maths mode instead.

Note that when changing the catcodes, we must save ` until last.

```
\begingroup
\catcode`\<\active
\catcode`\|\active
\catcode`\"\active
\catcode`\`\active
%
\gdef\syn@shorts#1#2{%
```

The '<' character must typeset its argument in italics. We make '_' do the same as the '_' command.

```
  \def<{%
    #1%
    \syn@assist%
      \syntleft%
      {\act_{\@uscore.}}%
      >%
      \syntright%
      {#2}%
  }%
```

The ‘‘’ and ‘"’ characters should print its argument in \tt font. We change the ‘\tt’ space glue to provide nicer spacing on the line.

```
\def‘{%
  #1%
  \syn@assist%
    \litleft%
    \relax%
    ’%
    \litright%
    {#2}%
}%
\def"{%
  #1%
  \syn@assist%
    \ulitleft%
    \relax%
    "%
    \ulitright%
    {#2}%
}%
```

Finally, the ‘|’ character is typeset by using the mysterious \textbar command.

```
\def|{\textbar}%
```

We’re finished here now.

```
}
%
\endgroup
```

\syntaxShortcuts This is a user-level command which enables the use of our shortcuts in the current group. It uses \addspecial, defined below, to register the active characters, sets up their definitions and activates them.

The two arguments are commands to be performed before and after the handling of the abbreviation. In this way, you can further process the output.

This command is not intended to be used directly by users: it should be used by other macros and packages which wish to take advantage of the facilities offered by this package. We provide a \synshorts declaration (which may be used as an environment, of course) which is more ‘user palatable’.

```
\def\syntaxShortcuts#1#2{%
  \syn@shorts{#1}{#2}%
  \addspecial\‘%
  \addspecial\<%
  \addspecial\|%
  \addspecial\"%
  \catcode‘\|\active%
  \catcode‘\<\active%
  \catcode‘\"\active%
  \catcode‘\‘\active%
}
%
\def\synshorts{\syntaxShortcuts\relax\relax}
```

`\synshortsoff` This macro can be useful occasionally: it disables the syntax shortcuts, so you can type normal text for a while.

```
\def\synshortsoff{%
  \catcode`\|12%
  \catcode`\<12%
  \catcode`\"12%
  \catcode`\`12%
}
```

`\syntax` The `\syntax` macro typesets its argument, allowing the use of our shortcuts within the argument.

Actually, we go to some trouble to ensure that the argument to `\syntax` *isn't* a real argument so we can change catcodes as we go. We use the `\let\@let@token=` trick from PLAIN TEX to do this.

```
\def\syntax#{\bgroup\syntaxShortcuts\relax\relax\let\@let@token}
```

`grammar` The grammar environment is the final object we have to define. It allows typesetting of beautiful BNF grammars.

First, we define the length parameters we need:

```
\newskip\grammarparsep
  \grammarparsep8\p@\@plus\p@\@minus\p@
\newdimen\grammarindent
  \grammarindent2em
```

Now define the default label typesetting. This macro is designed to be replaced by a user, so we'll be extra-well-behaved and use genuine LATEX commands. Well, almost ...

```
\newcommand{\grammarlabel}[2]{%
  \synt{#1} \hfill#2%
}
```

Now for a bit of hacking to make the item stuff work properly. This gets done for every new paragraph that's started without an `\item` command.

First, store the left hand side of the production in a box. Then I'll end the paragraph, and insert some nasty glue to take up all the space, so no-one will ever notice that there was a paragraph break there. The strut just makes sure that I know exactly how high the line is.

```
\def\gr@implitem<#1> #2 {%
  \sbox\z@{\hskip\labelsep\grammarlabel{#1}{#2}}%
  \strut\@@par%
  \vskip-\parskip%
  \vskip-\baselineskip%
```

The `\item` command will notice that I've inserted these funny glues and try to remove them: I'll stymie its efforts by inserting an invisible rule. Then I'll insert the label using `\item` in the normal way.

```
  \hrule\@height\z@\@depth\z@\relax%
  \item[\unhbox\z@]%
```

Just before I go, I'll make '<' back into an active character.

```
  \catcode`\<\active%
}
```

Now for the environment proper. Deep down, it's a list environment, with some nasty tricks to stop anyone from noticing.

The first job is to set up the list from the parameters I'm given.

```
\newenvironment{grammar}{%
  \list{}{%
    \labelwidth\grammarindent%
    \leftmargin\grammarindent%
    \advance\grammarindent\labelsep
    \itemindent\z@%
    \listparindent\z@%
    \parsep\grammarparsep%
  }%
```

We have major problems in `\raggedright` layouts, which try to use `\par` to start new lines. We go back to normal `\\` newlines to try and bodge our way around these problems.

```
  \let\\\@normalcr
```

Now to enable the shortcuts.

```
  \syntaxShortcuts\relax\relax%
```

Now a little bit of magic. The `\alt` macro moves us to a new line, and typesets a vertical bar in the margin. This allows typesetting of multiline alternative productions in a pretty way.

```
  \def\alt{\\\llap{\textbar\quad}}%
```

Now for another bit of magic. We set up some `\par` cleverness to spot the start of each production rule and format it in some cunning and user-defined way.

```
  \def\gr@setpar{%
    \def\par{%
      \parshape\@ne\@totalleftmargin\linewidth%
      \@@par%
      \catcode`\<12%
      \everypar{%
        \everypar{}%
        \catcode`\<\active%
        \gr@implitem%
      }%
    }%
  }%
  \gr@setpar%
  \par%
```

Now set up the `\[[` and `\]]` commands to do the right thing. We have to check the next character to see if it's correct, otherwise we'll open a maths display as usual.

```
  \let\gr@leftsq\[%
  \let\gr@rightsq\]%
  \def\gr@endsyntdiag]{\end{syntdiag}\gr@setpar\par}%
  \def\[{\@ifnextchar[{\begin{syntdiag}\@gobble}\gr@leftsq}%
  \def\]{\@ifnextchar]\gr@endsyntdiag\gr@rightsq}%
```

Well, that's it for this side of the environment.

```
}{%
```

Closing the environment is a simple matter of tidying away the list.

```
  \@newlistfalse%
  \everypar{}%
  \endlist%
}
```

3.7 Syntax diagrams

Now we come to the final and most complicated part of the package.

Syntax diagrams are drawn using arrow characters from LaTeX's line font, used in the picture environment, and rules. The horizontal rules of the diagram are drawn along the baselines of the lines in which they are placed. The text items in the diagram are placed in boxes and lowered below the main baseline. Struts are added throughout to keep the vertical spacing consistent.

The vertical structures (stacks and loops) are all implemented with TeX's primitive `\halign` command.

3.7.1 User-configurable parameters

First, we allocate the ⟨*dimen*⟩ and ⟨*skip*⟩ arguments needed. Fixed lengths, as the LaTeXbook calls them, are allocated as ⟨*dimen*⟩s, to take some of the load off of all the ⟨*skip*⟩ registers.

```
\newskip\sdstartspace
\newskip\sdendspace
\newskip\sdmidskip
\newskip\sdtokskip
\newskip\sdfinalskip
\newdimen\sdrulewidth
\newdimen\sdcirclediam
\newdimen\sdindent
```

We need some TeX ⟨*dimen*⟩s for our own purposes, to get everything in the right places. We use labels for the 'temporary' TeX parameters which we use, to avoid wasting registers.

```
\dimendef\sd@lower\z@
\dimendef\sd@upper\tw@
\dimendef\sd@mid4
\dimendef\sd@topcirc6
\dimendef\sd@botcirc8
```

`\sd@setsize` When the text size for syntax diagrams changes, it's necessary to work out the height for various rules in the diagram.

```
\def\sd@setsize{%
  \sd@mid\ht\strutbox%
  \advance\sd@mid-\dp\strutbox%
  \sd@mid.5\sd@mid%
  \sd@upper\sdrulewidth%
    \advance\sd@upper\sd@mid%
  \sd@lower\sdrulewidth%
    \advance\sd@lower-\sd@mid%
  \sd@topcirc-.5\sdcirclediam%
    \advance\sd@topcirc\sd@mid%
```

```
  \sd@botcirc-.5\sdcirclediam%
    \advance\sd@botcirc-\sd@mid%
}
```

`\sdsize` You can set the default type size used by syntax diagrams by redefining the `\sdsize` command, using the `\renewcommand` command.

By default, syntax diagrams are set slightly smaller than the main body text.[3]

```
\newcommand{\sdsize}{%
  \small%
}
```

`\sdlengths` Finally, the default length parameters are set in the `\sdlengths` command. You can redefine the command using `\renewcommand`.

We set up the length parameters here.

```
\newcommand{\sdlengths}{%
  \setlength{\sdstartspace}{1em minus 10pt}%
  \setlength{\sdendspace}{1em minus 10pt}%
  \setlength{\sdmidskip}{0.5em plus 0.0001fil}%
  \setlength{\sdtokskip}{0.25em plus 0.0001fil}%
  \setlength{\sdfinalskip}{0.5em plus 10000fil}%
  \setlength{\sdrulewidth}{0.2pt}%
  \setlength{\sdcirclediam}{8pt}%
  \setlength{\sdindent}{0pt}%
}
```

3.7.2 Other declarations

We define four switches. The table shows what they're used for.

Switch	Meaning
`\ifsd@base`	We are at 'base level' in the diagram: i.e., not in any other sorts of constructions. This is used to decide whether to allow line breaking.
`\ifsd@top`	The current loop construct is being typeset with the loop arrow above the baseline.
`\ifsd@toplayer`	We are typesetting the top layer of a stack. This is used to ensure that the vertical rules on either side are typeset at the right height.
`\ifsd@backwards`	We're typesetting backwards, because we're in the middle of a loop arrow. the only difference this makes is that any subloops have the arrow on the side.

Table 1: Syntax diagram switches

[3] I've used pure LaTeX commands for this and the `\sdlengths` macro, to try and illustrate how these values might be changed by a user. The rest of the code is almost obfuscted in its use of raw TeX features, in an attempt to dissuade more naïve users from fiddling with it. I suppose this is what you get when you let assembler hackers loose with something like LaTeX.

```
\newif\ifsd@base
\newif\ifsd@top
\newif\ifsd@toplayer
\newif\ifsd@backwards
```

`\sd@err` We output our errors through this macro, which saves a little typing.

```
\def\sd@err{\PackageError{syntax}}
```

3.7.3 Arrow-drawing

We need to draw some arrows. LaTeX tries to make this as awkward as possible, so we have to start moving the arrows around in boxes quite a lot.

The left and right pointing arrows are fairly simple: we just add some horizontal spacing to prevent the width of the arrow looking odd.

```
\def\sd@arrow{%
  \ht\tw@\z@%
  \dp\tw@\z@%
  \raise\sd@mid\box\tw@%
  \egroup%
}
\def\sd@rightarr{%
  \bgroup%
  \setbox\tw@\hbox{\kern-6\p@\@linefnt\char'55}%
  \sd@arrow%
}
\def\sd@leftarr{%
  \bgroup%
  \raise\sd@mid\hbox{\@linefnt\char'33\kern-6\p@}%
  \sd@arrow%
}
```

The up arrow is very strange. We need to bring the arrow down to base level, and smash its height.

```
\def\sd@uparr{%
  \bgroup%
  \setbox\tw@\hb@xt@\z@{\kern-\sdrulewidth\@linefnt\char'66\hss}%
  \setbox\tw@\hbox{\lower10\p@\box\tw@}%
  \sd@arrow%
}
```

The down arrow is similar, although it's already at the right height. Thus, we can just smash the box.

```
\def\sd@downarr{%
  \bgroup%
  \setbox\tw@\hb@xt@\z@{\kern-\sdrulewidth\@linefnt\char'77\hss}%
  \sd@arrow%
}
```

3.7.4 Drawing curves

If the user has selected curved edges, we use the LaTeX features provided to obtain the curves. These are drawn slightly oddly to make it easier to fit them into the diagram.

Some explanation about the LaTeX circle font is probably called for before we go any further. The font consists of sets of four quadrants of a particular size (and some other characters, which aren't important at the moment). Each collection of quadrants fit together to form a perfect circle of a given diameter. The individual quadrant characters have strange bounding boxes, as described in the files *lcircle.mf* and *ltpict.dtx*, and also in Appendix D of *The TeXbook*. Our job here is to make these quadrants useful in the context of drawing syntax diagrams.

`\sd@circ` First, we define `\sd@circ`, which performs the common parts of the four routines. Since the characters in the circle font are grouped together, we can pick out a particular corner piece by specifying its index into the group for the required size. The `\sd@circ` routine will pick out the required character, given this index as an argument, and put it in box 2, after fiddling with the sizes a little:

- We clear the width to zero. The individual routines then add a kern of the correct amount, so that the quadrant appears in the right place.
- The piece is lowered by half the rule width. This positions the top and bottom pieces of the circle to be half way over the baseline, which is the correct position for the rest of the diagram.

Finally, we make sure we're in horizontal mode: horrific results occur if this is not the case. I'm sure I don't need to explain this any more graphically.

```
316 \def\sd@circ#1{%
317   \@getcirc\sdcirclediam%
318   \advance\@tempcnta#1%
319   \setbox\tw@\hbox{\lower\sdrulewidth%
320     \hbox{\@circlefnt\char\@tempcnta}}%
321   \wd\tw@\z@%
322   \leavevmode%
323 }
```

`\sd@tlcirc` `\sd@trcirc` `\sd@blcirc` `\sd@brcirc` These are the macros which actually draw quadrants of circles. They all call `\sd@circ`, passing an appropriate index, and then fiddle with the box sizes and apply kerning specific to the quadrant positioning.

The exact requirements for positioning are as follows:

- The horizontal parts of the arcs must lie along the baseline (i.e., half the line must be above the baseline, and half must be below). This is consistent with the horizontal rules used in the diagram.
- The vertical parts must overlap vertical rules on either side, so that a `\vrule\sd@`*xx*`circ` makes the arc appear to be a real curve in the line. The requirements are actually somewhat inconsistent; for example, the stack environment uses curves *before* the `\vrule`s. Special requirements like this are handled as special cases later.
- The height and width of the arc are at least roughly correct.

```
324 \def\sd@tlcirc{{%
325   \sd@circ3%
326   \ht\tw@\sdrulewidth%
327   \dp\tw@.5\sdcirclediam%
```

```
  \kern-\tw@\sdrulewidth%
  \raise\sd@mid\box\tw@%
  \kern.5\sdcirclediam%
}}
\def\sd@trcirc{{%
  \sd@circ0%
  \ht\tw@\sdrulewidth%
  \dp\tw@.5\sdcirclediam%
  \kern.5\sdcirclediam%
  \raise\sd@mid\box\tw@%
}}
\def\sd@blcirc{{%
  \sd@circ2%
  \ht\tw@.5\sdcirclediam%
  \dp\tw@\sdrulewidth%
  \kern-\tw@\sdrulewidth%
  \raise\sd@mid\box\tw@%
  \kern.5\sdcirclediam%
}}
\def\sd@brcirc{{%
  \sd@circ1%
  \ht\tw@.5\sdcirclediam%
  \dp\tw@\sdrulewidth%
  \kern.5\sdcirclediam%
  \raise\sd@mid\box\tw@%
}}
```

`\sd@llc` `\sd@rlc`

In the `rep` environment, we need to be able to draw arcs with horizontal lines running through them. The two macros here do the job nicely. `\sd@llc` (which is short for left overlapping circle) is analogous to `\llap`: it puts its argument in a box of zero width, sticking out to the left. However, it also draws a rule along the baseline. This is important, as it prevents text from overprinting the arc. `\sd@rlc` is very similar, just the other way around.

```
\def\sd@llc#1{%
  \hb@xt@.5\sdcirclediam{%
    \sd@rule\hskip.5\sdcirclediam%
    \hss%
    #1%
  }%
}
\def\sd@rlc#1{%
  \hb@xt@.5\sdcirclediam{%
    #1%
    \hss%
    \sd@rule\hskip.5\sdcirclediam%
  }%
}
```

3.7.5 Drawing rules

It's important to draw the rules *along* the baseline, rather than above it: hence, the depth of the rule must be equal to the height.

\sd@rule We use rule leaders instead of glue through most of the syntax diagrams. The command \sd@rule⟨*skip*⟩ draws a rule of the correct dimensions, which has the behaviour of an \hskip⟨*skip*⟩.

```
\def\sd@rule{\leaders\hrule\@height\sd@upper\@depth\sd@lower}
```

\sd@gap The gap between elements is added using this macro. It will allow a line break if we're at the top level of the diagram, using a rather strange discretionary.

This is called as \sd@gap{⟨*skip-register*⟩}.

```
\def\sd@gap#1{%
```

First, we see if we're at the top level. Within constructs, we avoid the overhead of a \discretionary. We put half of the width of the skip on each side of the discretionary break.

```
  \ifsd@base%
    \skip@#1%
      \divide\skip\z@\tw@%
    \nobreak\sd@rule\hskip\skip@%
    \discretionary{%
      \sd@qarrow{->}%
    }{%
      \hbox{%
        \sd@qarrow{>-}%
        \sd@rule\hskip\sdstartspace%
        \sd@rule\hskip3.5\p@%
      }%
    }{%
    }%
    \nobreak\sd@rule\hskip\skip@%
```

If we're not at the base level, we just put in a rule of the correct width.

```
  \else%
    \sd@rule\hskip#1%
  \fi%
}
```

3.7.6 The syntdiag environment

All syntax diagrams are contained within a syntdiag environment.

syntdiag The only argument is a collection of declarations, which by default is

```
\sdsize\sdlengths
```

However, if the optional argument is not specified, TEX reads the first character of the environment, which may not be catcoded correctly. We set up the catcodes first, using the \syntaxShortcuts command, and then read the argument. We don't use \newcommand, because that would involve creating yet *another* macro. Time to fiddle with \@ifnextchar ...

```
\def\syntdiag{%
  \syntaxShortcuts\sd@tok@i\sd@tok@ii%
  \@ifnextchar[\syntdiag@i{\syntdiag@i[]}%
}
```

Now we actually do the job we're meant to.

```
\def\syntdiag@i[#1]{%
```

The first thing to do is execute the user's declarations. We then set up things for the font size.

```
  \sdsize\sdlengths%
  #1%
  \sd@setsize%
```

Next, we start a list, to change the text layout.

```
  \list{}{%
    \leftmargin\sdindent%
    \rightmargin\leftmargin%
    \labelsep\z@%
    \labelwidth\z@%
  }%
  \item[]%
```

We reconfigure the paragraph format quite a lot now. We clear `\parfillskip` to avoid any justification at the end of the paragraph. We also turn off paragraph indentation.

```
  \parfillskip\z@%
  \noindent%
```

Next, we add in the arrows on the beginning of the line, and a bit of glue.

```
  \sd@qarrow{>>-}%
  \nobreak\sd@rule\hskip\sdstartspace%
```

This is the base level of the diagram, so we enable line breaking.

```
  \sd@basetrue%
```

Since the objects being broken are rather large, we enable sloppy line breaking. We also try to avoid page breaks in mid-diagram, by upping the `\interlinepenalty`.

```
  \sloppy%
  \interlinepenalty100%
  \hyphenpenalty0%
```

We handle all the spacing within the environment, so we make TEX ignore spaces and newlines.

```
  \catcode`\ 9%
  \catcode`\^^M9%
```

We now have to change the behaviour of \\ to line-break syntax diagrams.

```
  \let\\\sd@newline%
  \ignorespaces%
}
```

When we end the diagram, we just have to add in the final fillskip, and double arrow.

```
\def\endsyntdiag{%
  \unskip%
  \nobreak\sd@rule\hskip\sdmidskip%
```

```
  \sd@rule\hskip\sdfinalskip%
  \sd@qarrow{->><}%
  \endlist%
}
```

syntdiag* The starred form of syntdiag typesets a syntax diagram in LR-mode; this is useful if you're describing parts of syntax diagrams, for example.

This is in fact really easy. The first bit which checks for an optional argument is almost identical to the non-∗ version.

```
\@namedef{syntdiag*}{%
  \syntaxShortcuts\sd@tok@i\sd@tok@ii%
  \@ifnextchar[\syntdiag@s@i{\syntdiag@s@i[]}%
}
```

Handle another optional argument giving the width of the box to fill.

```
\def\syntdiag@s@i[#1]{%
  \@ifnextchar[{\syntdiag@s@ii{#1}}{\syntdiag@s@iii{#1}{\hbox}}%
}
\def\syntdiag@s@ii#1[#2]{\syntdiag@s@iii{#1}{\hb@xt@#2}}
```

Now to actually start the display. This is mostly simple. Just to make sure about the LR-ness of the typesetting, I'll put everything in an hbox.

```
\def\syntdiag@s@iii#1#2{%
  \leavevmode%
  #2\bgroup%
```

Now configure the typesetting according to the user's wishes.

```
  \let\@@left\left%
  \let\@@right\right%
  \def\left##1{\def\sd@startarr{##1}}%
  \def\right##1{\def\sd@endarr{##1}}%
  \left{>-}\right{->}%
  \sdsize\sdlengths%
  #1%
  \sd@setsize%
  \let\left\@@left%
  \let\right\@@right%
```

Put in the initial double-arrow.

```
  \sd@qarrow\sd@startarr%
  \sd@rule\hskip\sdmidskip%
```

We're in horizontal mode, so don't bother with linebreaking.

```
  \sd@basefalse%
```

Finally, disable spaces and things.

```
  \catcode`\ 9%
  \catcode`\^^M9%
  \ignorespaces%
}
```

Ending the environment is very similar.

```
\@namedef{endsyntdiag*}{%
  \unskip%
```

```
  \sd@rule\hskip\sdmidskip%
  \sd@rule\hskip\sdfinalskip%
  \sd@qarrow\sd@endarr%
  \egroup%
}
```

`\sd@qarrow` This typesets the various left and right arrows required in syntax diagrams. The argument is one of '»-', '->', '>-' or '-><'.

```
\def\sd@qarrow#1{%
  \begingroup%
  \lccode`\~=`\<\lowercase{\def~{<}}%
  \hbox{\csname sd@arr@#1\endcsname}%
  \endgroup%
}
\@namedef{sd@arr@>>-}{\sd@rightarr\kern-.5\p@\sd@rightarr\kern-\p@}
\@namedef{sd@arr@>-}{\sd@rightarr\kern-\p@}
\@namedef{sd@arr@->}{\sd@rightarr}
\@namedef{sd@arr@-><}{\sd@rightarr\kern-\p@\sd@leftarr}
\@namedef{sd@arr@...}{$\cdots$}
\@namedef{sd@arr@-}{}
```

`\sd@newline` The line breaking within a syntax diagram is controlled by the `\sd@newline` command, to which `\\` is assigned.

We support all the standard LaTeX features here. The line breaking involves adding a fill skip and arrow, moving to the next line, adding an arrow and a rule, and continuing.

```
\def\sd@newline{\@ifstar{\vadjust{\penalty\@M}\sd@nl@i}\sd@nl@i}
\def\sd@nl@i{\@ifnextchar[\sd@nl@ii\sd@nl@iii}
\def\sd@nl@ii[#1]{\vspace{#1}\sd@nl@iii}
\def\sd@nl@iii{%
  \nobreak\sd@rule\hskip\sdmidskip%
  \sd@rule\hskip\sdfinalskip%
  \kern-3\p@%
  \sd@rightarr%
  \newline%
  \sd@rightarr%
  \nobreak\sd@rule\hskip\sdstartspace%
  \sd@rule\hskip3.5\p@%
}
```

3.7.7 Putting things in the right place

Syntax diagrams have fairly stiff requirements on the positioning of text relative to the diagram's rules. To help people (and me) to write extensions to the syntax diagram typesetting which automatically put things in the right place, I provide some simple macros.

`sdbox` By placing some text in the sdbox environment, it will be read into a box and then output at the correct height for the syntax diagram. Note that stuff in the box is set in horizontal (LR) mode, so you'll have to use a minipage if you want formatted text. The macro also supplies rules on either side of the box, with a length given in the environment's argument.

Macro writers are given explicit permission to use this environment through the `\sdbox` and `\endsdbox` commands if this makes life easier.

The calculation in the `\endsdbox` macro works out how to centre the box vertically over the baseline. If the box's height is h, and its depth is d, then its centre-line is $(h + d)/2$ from the bottom of the box. Since the baseline is already d from the bottom, we need to lower the box by $(h + d)/2 - d$, or $h/2 - d/2$.

```
\def\sdbox#1{%
  \@tempskipa#1\relax%
  \sd@gap\@tempskipa%
  \setbox\z@\hbox\bgroup%
    \begingroup%
    \catcode`\ 10%
    \catcode`\^^M5%
    \synshortsoff%
}
\def\endsdbox{%
    \endgroup%
  \egroup%
  \@tempdima\ht\z@%
  \advance\@tempdima-\dp\z@%
  \advance\@tempdima-\tw@\sd@mid%
  \lower.5\@tempdima\box\z@%
  \sd@gap\@tempskipa%
}
```

3.7.8 Typesetting syntactic items

Using the hooks built into the syntax abbreviations above, we typeset the text into a box, and write it out, centred over the baseline. A strut helps to keep the actual text baselines level for short pieces of text.

`\sd@tok@i` The preamble for a syntax abbreviation. We start a box, and set the space and return characters to work again. A strut is added to the box to ensure correct vertical spacing for normal text.

```
\def\sd@tok@i{%
  \sdbox\sdtokskip%
  \strut%
  \space%
}
```

`\sd@tok@ii`

```
\def\sd@tok@ii{%
  \space%
  \endsdbox%
}
```

3.7.9 Inserting other pieces of text

Arbitrary text may be put into a syntax diagram through the use of the `\tok` macro. Its 'argument' is typeset in the same way as a syntactic item (centred over the baseline). The implementation goes to some effort to ensure that the text is not actually an argument, to allow category codes to change while the text is being typeset.

\tok We start a box, and make space and return do their normal jobs. We use \aftergroup to regain control once the box is finished. \doafter is used to get control after the group finishes.

```
\def\tok#{%
  \sdbox\sdtokskip%
  \strut%
  \enspace%
  \syntaxShortcuts\relax\relax%
  \doafter\sd@tok%
}
```

The \sd@tok macro is similar to \sd@tok@ii above.

```
\def\sd@tok{%
  \enspace%
  \endsdbox%
}
```

3.7.10 The stack environment

The stack environment is used to present alternatives in a syntax diagram. The alternatives are separated by \\ commands.

\stack The optional positioning argument is handled using LaTeX's \newcommand mechanism.

```
\newcommand\stack[1][t]{%
```

First, we add some horizontal space.

```
  \sd@gap\sdmidskip%
```

We're within a complex construction, so we need to clear the \ifsd@base flag.

```
  \begingroup\sd@basefalse%
```

The top and bottom rows of the stack are different to the others, since the vertical rules mustn't extend all the way up the side of the item. The bottom row is handled separately by \endstack below. The top row must be handled via a flag, \ifsd@toplayer.

Initially, the flag must be set true.

```
  \sd@toplayertrue%
```

We set the \\ command to separate the items in the \halign.

```
  \let\\\sd@stackcr%
```

The actual structure must be set in vertical mode, so we must place it in a box. The position argument determines whether this must be a \vbox or a \vtop. We also insert a bit of rounding if the options say we must.

```
  \if#1t%
    \let\@tempa\vtop%
    \sd@toptrue%
    \ifsd@round\llap{\sd@trcirc\kern\tw@\sdrulewidth}\fi%
  \else\if#1b%
    \let\@tempa\vbox%
    \sd@topfalse%
    \ifsd@round\llap{\sd@brcirc\kern\tw@\sdrulewidth}\fi%
```

```
  \else%
    \sd@err{Bad position argument passed to stack}%
           {The positioning argument must be one of `t' or `b'.  I%
            have^^Jassumed you meant to type `t'.}%
    \let\@tempa\vtop%
  \fi\fi%
```

Now we start the box, which we will complete at the end of the environment.

```
  \@tempa\bgroup%
```

We must remove any extra space between rows of the table, since the rules will not join up correctly. We can use `\offinterlineskip` safely, since each individual row contains a strut.

```
  \offinterlineskip%
```

Now we can start the alignment. We actually use Plain TEX's `\ialign` macro, which also clears `\tabskip` for us.

```
  \ialign\bgroup%
```

The preamble is trivial, since we must do all of the work ourselves

```
    ##\cr%
```

We can now start putting the text into a box ready for typesetting later. The strut makes the vertical spacing correct.

```
  \setbox\z@\hbox\bgroup%
    \strut%
}
```

`\endstack` The first part of this is similar to the `\sd@stackcr` macro below, except that the vertical rules are different. We don't support rounded edges on single-row stacks, although this isn't a great loss to humanity.

```
\def\endstack{%
  \egroup%
  \ifsd@toplayer%
    \sd@dostack\sd@upper\sd@lower\relax\relax%
  \else%
    \ifsd@round%
      \ifsd@top%
        \sd@dostack{\ht\z@}\sd@botcirc\sd@blcirc\sd@brcirc%
      \else%
        \sd@dostack{\ht\z@}\sd@botcirc\relax\relax%
      \fi%
    \else%
      \sd@dostack{\ht\z@}\sd@lower\relax\relax%
    \fi%
  \fi%
```

We now close the `\halign` and the vbox we created.

```
  \egroup%
  \egroup%
```

Deal with any rounding we started off.

```
  \ifsd@round%
    \ifsd@top
```

```
      \rlap{\kern\tw@\sdrulewidth\sd@tlcirc}%
    \else%
      \rlap{\kern\tw@\sdrulewidth\sd@blcirc}%
    \fi%
  \fi%
```

Finally, we add some horizontal glue to space the diagram out.

```
  \endgroup\sd@gap\sdmidskip%
}
```

`\sd@stackcr` The `\\` command is set to this macro during a stack environment.

```
\def\sd@stackcr{%
```

The first job is to close the box containing the previous item.

```
  \egroup%
```

Now we typeset the vertical rules differently depending on whether this is the first item in the stack. This looks quite terrifying initially, but it's just an enumeration of the possible cases for the different values of `\ifsd@toplayer`, `\ifsd@top` and `\ifsd@round`, putting in appropriate rules and arcs in the right places.

```
  \ifsd@toplayer%
    \ifsd@round%
      \ifsd@top%
        \sd@dostack\sd@topcirc{\dp\z@}\relax\relax%
      \else%
        \sd@dostack\sd@topcirc{\dp\z@}\sd@tlcirc\sd@trcirc%
      \fi%
    \else%
      \sd@dostack\sd@upper{\dp\z@}\relax\relax%
    \fi%
  \else%
    \ifsd@round%
      \ifsd@top%
        \sd@dostack{\ht\z@}{\dp\z@}\sd@blcirc\sd@brcirc%
      \else%
        \sd@dostack{\ht\z@}{\dp\z@}\sd@tlcirc\sd@trcirc%
      \fi%
    \else%
      \sd@dostack{\ht\z@}{\dp\z@}\relax\relax%
    \fi%
  \fi%
```

The next item won't be the first, so we clear the flag.

```
  \sd@toplayerfalse%
```

Now we have to set up the next cell. We put the text into a box again.

```
  \setbox\z@\hbox\bgroup%
    \strut%
}
```

`\sd@dostack` Actually typesetting the text in a cell is performed here. The macro is called as

`\sd@dostack{`⟨*height*⟩`}{`⟨*depth*⟩`}{`⟨*left-arc*⟩`}{`⟨*right-arc*⟩`}`

where ⟨*height*⟩ and ⟨*depth*⟩ are the height and depth of the vertical rules to put around the item, and ⟨*left-arc*⟩ and ⟨*right-arc*⟩ are commands to draw rounded edges on the left and right hand sides of the item.

The values for the height and depth are quite often going to be the height and depth of box 0. Since we empty box 0 in the course of typesetting the row, we need to cache the sizes on entry.

```
\def\sd@dostack#1#2#3#4{%
  \@tempdima#1%
  \@tempdimb#2%
  \kern-\tw@\sdrulewidth%
  \vrule\@height\@tempdima\@depth\@tempdimb\@width\tw@\sdrulewidth%
  #3%
  \sd@rule\hfill%
  \sd@gap\sdtokskip%
  \unhbox\z@%
  \sd@gap\sdtokskip%
  \sd@rule\hfill%
  #4%
  \vrule\@height\@tempdima\@depth\@tempdimb\@width\tw@\sdrulewidth%
  \kern-\tw@\sdrulewidth%
  \cr%
}
```

3.7.11 The rep environment

The rep environment is used for typesetting loops in the diagram. Again, we use `\halign` for the typesetting. Loops are simpler than stacks, however, since there are always two rows. We store both rows in box registers, and build the loop at the end.

`\rep` Again, we use `\newcommand` to process the optional argument.

```
\newcommand\rep[1][t]{%
```

First, leave a gap on the left side.

```
  \sd@gap\sdmidskip%
```

We're not at base level any more, so disable linebreaking.

```
  \begingroup\sd@basefalse%
```

Remember we're going backwards now.

```
  \ifsd@backwards\sd@backwardsfalse\else\sd@backwardstrue\fi%
```

Define \\ to separate the two parts of the loop.

```
    \let\\\sd@loop%
```

Now check the argument, and use the appropriate type of box. In addition to changing the typesetting, we must remember which way up to typeset the loop, since the end code must always put the first argument on the baseline, with the loop either above or below.

```
  \if#1t%
    \let\@tempa\vbox%
    \sd@toptrue%
  \else\if#1b%
```

```
    \let\@tempa\vtop%
    \sd@topfalse%
  \else%
    \sd@err{Bad position argument passed to loop}%
           {The positioning argument must be `t' or `b'.  I have^^J%
            assumed you meant to type `t'.}%
    \let\@tempa\vbox%
    \sd@toptrue%
  \fi\fi%
```

Now we start the box.

```
  \@tempa\bgroup%
```

The loop is by default empty, apart from a strut. This is put into box 1.

```
  \setbox\tw@\copy\strutbox%
```

Now start typesetting the main text in box 0.

```
  \setbox\z@\hbox\bgroup\strut%
}
```

`\endrep` The final code must first close whatever box was open.

```
\def\endrep{%
  \egroup%
```

Now we typeset the loop, depending on which way up it was meant to be. Again, this terrifying piece of code is a simple list of possibile values of our various flags.

```
  \ifsd@top%
    \ifsd@round%
      \sd@doloop\tw@\z@\relax\relax%
        \sd@tlcirc\sd@trcirc{\sd@rlc\sd@blcirc}{\sd@llc\sd@brcirc}%
    \else%
      \sd@doloop\tw@\z@\relax\sd@downarr\relax\relax\relax\relax%
    \fi%
  \else%
    \ifsd@round%
      \sd@doloop\z@\tw@\relax\relax%
        {\sd@rlc\sd@tlcirc}{\sd@llc\sd@trcirc}\sd@blcirc\sd@brcirc%
    \else%
      \sd@doloop\z@\tw@\sd@uparr\relax\relax\relax\relax\relax%
    \fi%
  \fi%
```

Close the vbox we opened.

```
  \egroup%
```

Finally, we leave a gap before the next structure.

```
  \endgroup\sd@gap\sdmidskip%
}
```

`\sd@loop` This macro handles the \\ command within a loop environment. We close the current box, and start filling in box 1. We also redefine \\ to raise an error when the \\ command is used again.

```
\def\sd@loop{%
  \egroup%
  \def\\{\sd@err{Too many \string\\\space commands in loop}\@ehc}%
  \setbox\tw@\hbox\bgroup\strut%
}
```

`\sd@doloop` This is the macro which actually creates the `\halign` for the loop. It is called with four arguments, as:

> `\sd@doloop{`⟨*top-box*⟩`}{`⟨*bottom-box*⟩`}{`⟨*top-arrow*⟩`}{`⟨*btm-arrow*⟩`}`
> `{`⟨*top-left-arc*⟩`}{`⟨*top-right-arc*⟩`}{`⟨*bottom-left-arc*⟩`}{`⟨*btm-right-arc*⟩`}`

The two ⟨*box*⟩ arguments give the numbers of boxes to extract in the top and bottom rows of the alignment. The ⟨*arrow*⟩ arguments specify characters to typeset at the end of the top and bottom rows for arrows. The various ⟨*arc*⟩ arguments are commands which typeset arcs around the various parts of the items.

We calculate the height and depth of the two boxes, and store them in ⟨*dimen*⟩ registers, because the boxes are emptied before the right-hand rules are typeset.

Actually, the two rows of the alignment are typeset in a different macro: we just pass the correct information on.

```
\def\sd@doloop#1#2#3#4#5#6#7#8{%
  \@tempdima\dp#1\relax%
  \@tempdimb\ht#2\relax%
  \offinterlineskip%
  \ialign{%
    ##\cr%
    \ifsd@round%
      \sd@doloop@i#1#3\sd@topcirc\@tempdima{#5}{#6}%
      \sd@doloop@i#2#4\@tempdimb\sd@botcirc{#7}{#8}%
    \else%
      \sd@doloop@i#1#3\sd@upper\@tempdima{#5}{#6}%
      \sd@doloop@i#2#4\@tempdimb\sd@lower{#7}{#8}%
    \fi%
  }%
}
```

`\sd@doloop@i` Here we do the actual job of typesetting the rows of a loop alignment. The four arguments are:

> `\sd@doloop@i{`⟨*box*⟩`}{`⟨*arrow*⟩`}{`⟨*rule-height*⟩`}{`⟨*rule-depth*⟩`}`
> `{`⟨*left-arc*⟩`}{`⟨*right-arc*⟩`}`

The arrow position is determined by the `\ifsd@backwards` flag. The rest is fairly simple.

```
\def\sd@doloop@i#1#2#3#4#5#6{%
  \ifsd@backwards#2\fi%
  \kern-\tw@\sdrulewidth%
  \vrule\@height#3\@depth#4\@width\tw@\sdrulewidth%
  #5%
  \sd@rule\hfill%
  \sd@gap\sdtokskip%
  \unhbox#1%
  \sd@gap\sdtokskip%
```

```
688   \sd@rule\hfill%
689   #6%
690   \vrule\@height#3\@depth#4\@width\tw@\sdrulewidth%
691   \ifsd@backwards\else#2\fi%
692   \kern-\tw@\sdrulewidth%
693   \cr%
694 }
```

3.8 The end

Phew! That's all of it completed. I hope this collection of commands and environments is of some help to someone.

695 ⟨/package⟩

Mark Wooding, 17 May 1996

Appendix

A The GNU General Public Licence

The following is the text of the GNU General Public Licence, under the terms of which this software is distrubuted.

GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

A.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

A.2 Terms and conditions for copying, distribution and modification

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such

modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

(a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

(b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

(c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

(a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

(b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of

Sections 1 and 2 above on a medium customarily used for software interchange; or,

(c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as

a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. **Because the Program is licensed free of charge, there is no warranty for the Program, to the extent permitted by applicable law.**

except when otherwise stated in writing the copyright holders and/or other parties provide the program "as is" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the Program is with you. Should the Program prove defective, you assume the cost of all necessary servicing, repair or correction.

12. **In no event unless required by applicable law or agreed to in writing will any copyright holder, or any other party who may modify and/or redistribute the program as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use the program (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the Program to operate with any other programs), even if such holder or other party has been advised of the possibility of such damages.**

END OF TERMS AND CONDITIONS

A.3 Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>
Copyright (C) 19yy  <name of author>

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.
```

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than ‘show w’ and ‘show c’; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
‘Gnomovision’ (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Index

Numbers written in italic refer to the page where the corresponding entry is described, the ones underlined to the code line of the definition, the rest to the code lines where the entry is used.

Symbols

T

U

V