S2P

Eetex executable changes

category: Eetex Documentation

version: 2000.12.20

date: December 20, 2000

author: Taco Hoekwater

1 Introduction

This article gives an introduction to eetex. Eetex is an extension to pdfeTEX 2.1 that defines a collection of new primitives. Most of these deal with list data structures, but some other things are added as well.

1.1 Notes on using eetex

Eetex writes it's format files with extension `.eefm` (`.eef` on real-mode MSDOS) to distinguish itself from other executables that use the same TEX source for their format files.

Starting eetex and generating formats for eetex is a lot like using ε-TEX in extended mode. For example,

```
eetex -ini *s2pfmt
```

creates a new format file for `s2pfmt.tex` with the ε-TEX extensions and the eetex extensions both enabled. Don't forget the `*` in the command line, or you will end up with a halfway solution.

2 New primitives for list manipulations

All following primitives deal with lists. Lists behave a lot like normal TEX macros, but they have an internal substructure that can most easily be thought of as specifying separate token list items in a list, separated by one or more tokens that are handled specially.

The primitives listed below are really quite primitive. Higher-level macros should be written to make real use of the new functionality. Depending on the design of these macros, lists can work either as arrays or as lists or as queues or as unique sets.

Without going into very much detail, here are some of the problems that can benefit from the addition of lists to TEX's repertoire of basic data structures:

- Parsing input data.
  Some TEX macro packages read a lot of plain ASCII data that has to be split into separate tokens for processing. An example would be a plotting package, another example would be a macro package like ConTEXt's `supp-ver`, that interprets verbatim text.
  The macros that do this usually process the argument one character at a time using a brute force approach. In most of these cases, the same work can be done a lot easier and faster using lists.
- Parsing key=value pairs.
  There are commands available to get just a portion of an item. This comes in handy when macros are used to parse things like

  ```
  \includegraphics[height=6cm]{figure.eps}
  ```

- Creating cross-reference lists. Labels are usually required to be unique withing a certain scope. Lists make it fairly simple to create an application wherein the scope of the uniqueness (document, chapter, section) can be changed.
  Lists also make it easier to write macros that differentiate between different types of labels (sections, formulas, tables).
- Economizing the hash table.
  TEX's hash table usually has a limited size (even in 'dynamic' versions like web2c, there still is an upper limit). But there usually is a lot of memory for token lists available. Therefore, it makes sense to store commands as token lists.
  Every list occupies just one hash entry, regardless of it's size. Yet, a list can be used to save the value of a macro in it's items.
- Database publishing.
  Lists can be used for publication of for instance production or bibliographical databases, because lists allows you to use more than 50.000 cross-refs within on document easily using the tricks from the previous two items.
- Exporting information from within a group.
  Lists make it possible to save values that are computed within a level of TEX grouping without having to resort to `\global` definitions.
- Maintaining information stacks.

2.1 Tracking what is going on

2.2 \tracinglists

Just like all the other tracing commands, this primitive displays information about what the executable is doing. Currently, this is somewhat more like debugging information than something a user might be interested in. Usable values are 1 and 2.

Related to this new primitive, setting `\tracingstats` to 3 or 4 results in a large amount of memory allocation information.

2.3 Splitting arguments into items

2.4 \listsep

This is a new internal token register, whose contents is used both as item separator when the user specifies a list and as filler between items when an expansion of a list takes place. See below for an example of the usage of `\listsep`.

The logic by which the separation happens is a perhaps little strange, but the current solution turned out to be the most desirable behaviour. It goes like this:

- The first token from the expansion of `\listsep` is used as separator token in list specifications. This token (on its own) is used to decide where items end and a new one starts.
- However, all subsequent tokens from `\listsep` are removed from the input as well, provided that they appear directly after the item separator and in the correct relative order.
- If `\listsep`'s current value is empty, the input will be split up into separate tokens, each item consisting of precisely one token from the input.
- Initex initializes `\listsep` to the equivalent of `\listsep={,}`.

The trick with subsequent tokens makes it possible to say for example `\listsep{, }`, making certain that there are no items in the resulting list that start off with a space (as might be the case for `\listsep{,}`).

2.5 How to define a list

There are many ways to define a list or change the definition of an already existing list. The following new primitives work directly on lists.

```
\listdef <csname> [{item text}|<listcs>]
\appdef   "
\predef   "
\insdef   "
\elistdef "
\eappdef  "
\epredef  "
\einsdef  "
```

The four primitives that start with `"e"` expand the `item text`, the others don't. This difference is anologous to the difference between `\def` and `\edef`.

All eight primitives are assignments, just like the 'normal' `\def`. It is much easier to give an example of how to use these commands than it is to try to explain the formal logic, so here is an example of the four different types of definitions:

```
\listsep{, }
\listdef \mylist {noot,mies}
\appdef  \mylist {wim}
\listdef \first  {aap,}
\predef  \mylist \first
\insdef  \mylist {noot,zus}
\listsep{ }
\message {\mylist}
\bye
```

The terminal output of this example is:

```
zus aap  noot mies wim
```

with two spaces between `aap` and `noot`.

Wat happens on the preceding lines is the following:

1. Initializes `\listsep`
2. This line defines the csname `\mylist` to be a 2-item list consisting of the items `noot` and `mies`.
3. This appends the item `wim`. The list `\mylist` now has 3 items.
4. This defines `\first` as another 2-item list. The second item of this list is empty.
5. Prepends that new list to `\mylist`. (`\mylist` now is `"aap"` `""` `"noot"` `"mies"` `"wim"`)
6. Requests insertion of the items `noot` and `zus`. `noot` already exists in the list, so that one is ignored and `zus` is added.
7. Changes `\listsep` for the subsequent expansion.
8. Expands `\mylist`. Notice that you get 2 spaces around the empty item.

All of these primitives adhere to standard TEX grouping, and they all understand the `\global` prefix. The control sequence name that becomes defined is always considered to be `\long` and never considered to be `\outer`.

An completely empty `item text` does nothing if it is used together with the insertion or addition primitives, but `\listdef \mylist {}` *does* change `\mylist` into a csname that expands into a list (of zero items). Empty lists as well as empty items are legal (both have their uses).

Lists expand into a token list that is a concatenation of the items' contents, with the items separated by the current expansion of `\listsep`. This expansion happens without the need for the user to do anything; lists are 'callable' just like macros even if their interal structure is quite different.

But lists expand into the internal representation of a number if TEX is looking for an integer (for primitives like `\ifcase`, `\number`, counter assignments, etc.) The returned number is the number of items in the list. This gives you a simple way to measure the length of a list.

2.6 \newlist<csname> <number>

Creates a list with `<number>` amount of items. This is useful for array specifications, and for extending or shortening an already existing list. If you are extending an already existing list or if you are creating an entiry new one, all new items in it will be empty.

Already existing items keep their value. If you are shortening an existing list, the items that are cut off are irretrievably lost though. Subsequently extending the list will *not* give them back.

2.7 Mapping macros to lists

2.8 \scanlist<listcs> <token>

Explicitly expands the list that is pointed to by `<listcs>`. But instead of inserting the current meaning of `\listsep`, it inserts the `<token>` between every item and before the very first item of the list, and it adds braces around the separate items. The idea is (see below for an explanation of `\quitlist`):

```
\def\noot{noot}
\def \test#1{\def \tempa{#1}
      \ifx \tempa \noot
        \message{done}%
        \quitlist 1
      \else
     \message{#1}%
      \fi }
\scanlist \mylist \test
```

If `\mylist` consists of the three items `aap`, `noot` and `mies`, the `\scanlist` expands into

```
\test{aap}\test{noot}\test{mies}
```

2.9 \quitlist<number>

Quits from the `<number>`-ed input level above the current one that is a token list which is the result of expanding the `\scanlist` primitive. This sounds complicated, but it simply means that

```
\quitlist 1
```

could be used in the `\test` macro above to escape out of the list's expansion once the condition was met (so that `\test{mies}` was never expanded). On long lists, this can safe a lot of processing time. In nested definitions, numbers higher than one might also be useful.

The command `\quitlist 0` is a special case: it kills the current token list, regardless of its type. This is likely to be the expansion of a macro, and it means that macros can actually quit themselves (like `exit` and `return` do in other programming languages).

2.10 Finding out if this is a list

2.11 \iflist<csname>

Returns true if the <csname> represents a list.

Besides this, \ifx returns true if it's two csnames are two lists which have the same number of items *and* whose expansions fully agree. Note that the comparison is expansion-based, such that in the following example:

```
\listdef\mylist{no,ta}
\listdef\mylisttwo{n,ota}
\listsep{}
\ifx\mylist\mylisttwo
```

the \ifx evaluates to true.

\ifx returns false in all other cases, including the comparison between a one-item list and a macro that has precisely the same expansion.

\ifcase, \ifnum and \number return the number of items (which would be 2 in this case).

2.12 Searching for (part of) an item

```
\ifhasitem <listcs> [{item}|<listcs>]
\ifsubitem <listcs> [{subitem}|<listcs>]
\ifsublist <listcs> [{items}|<listcs>]
\ifsubset  <listcs> [{items}|<listcs>]
```

These are four new \if tests.

\ifhasitem tests for the existance of one item. You can specify more than one item in the items part if you want, but those are never looked at.

\ifsubitem tests for an item that starts with subitem. This is especially useful for key=value pairs.

\ifsublist tests for items appearing in the specified relative order. Intervening items are allowed, but the relative order must be maintained.

\ifsubset tests for all the items appearing in any order at all. There is currently no way to test whether items appear more than once in the list.

```
\itemnumber <listcs> [{item}|<listcs>]
\subitemnumber <listcs> [{subitem}|<listcs>]
```

These are two new expandable primitives that return TEX internal `number`s. Here it is also possible to specify extra items if you want to, but they are ignored completely. If the requested (sub)-item does not exist, these commands return zero.

The above six new primitives have an extra sideeffect: when the tests are succesfull (either the `\if`s are true or for `\...number` there is indeed such an item), the requested item's info is saved in two global variables that can be queried by the user. These are:

```
\lastitemnumber   % a counter
\lastitemdata     % a \long macro
```

If the test is unsuccessful, `\lastitemnumber` will be 0 and `\lastitemdata` will be empty.

If the test is successful, `\lastitemnumber` will be the itemnumber of the requested (sub)item and `\lastitemdata` will be the text of that item.

If the request was for a subitem, `\lastitemdata` contains *only* the trailing contents of the item, with one optional layer of containing braces stripped.

The main advantage of this side-effect is that it allows you to replace the construction

```
\ifsubitem \mylist {clip}
 \getitem \mylist \subitemnumber{clip} to \tempa
 \EA\def\EA\keyval\EA{\EA\stripeq\tempa=}\fi
```

with this code, which is both faster and a lot cleaner to look at:

```
\ifsubitem \mylist {clip=}
 \EA\def\EA\keyval\EA{\lastitemdata}%
\fi
```

Note: for `\ifsublist` and `\ifsubset`, the values will be the info of the last specified item.

Second note: `\lastitemnumber` might be zero even within the true branch (this is the result you get from checking for the existence of an empty item).

One last note: `\lastitemdata` is a `\long` macro after the first use of one of these six primitives, but for initex it is initialized to a weird typeless primitive. The contents or both csnames do not survive dumping and undumping formats, so they can not be used in `\everyjob`.

2.13 Manipulating items

```
\getitem <listcs> <number> to <csname>
\setitem "
\delitem "
\insitem "
```

Four new primitives to play with separate items.

`\getitem` defines `<csname>` to be the meaning of the relevant item of the list. `\setitem` works the other way around. `\delitem` is like `\getitem` but it also destroys the item of the list (the list actually becomes shorter), `\insitem` is `\delitem` inverted (the list gets longer).

Negative values for `<number>` count from the tail forward, such that -1 means the last item, and 1 the first item. These primitives are quite flexible, but `<csname>` has to be a macro. Using other primitives like `\message` will give you an error message.

Argument specifications (both # marks and delimited text) for macros are saved into the list as well, so that it is possible to do this:

```
\def\tempa#1#2{\message{(#1, #2)}}
\setitem \mylist 1 to \tempa
\getitem \mylist 1 to \tempb
\tempb {A}{B}
```

Eetex executable changes

3 Other primitives and functionality

The primitives below have nothing to do with lists, but are added because we thought they might be useful.

3.1 \eeTeXversion

The first of these is for maintenance reasons only. \eeTeXversion is a read-only register that gives you the release number of the version of eetex that you are using. Currently, it returns the value 2.

3.2 Toks manipulation

The commands \apptoks and \pretoks allow you to append or prepend tokens to a token register. Here are four simple examples:

```
\apptoks \everyjob ={\message{This is eetex}}
\pretoks \output   ={\message{Output called}}
\apptoks \toks5    \toks2
\pretoks \mytoks   \toks0
```

3.3 \sgmlmode

This is a read-write register with default value 0. Setting this register to a non-zero value changes the way TEX reads control sequence names. With \sgmlmode=1, a csname ends *only* at the next space (ascii 32).

Suppose you want to parse the following input:

```
<p indent=none>First paragraph</p>
<p>Second paragraph</p>
<s3 align=center>A subsubsection
```

The p's are simple to do in current TEX, using a definition like:

```
\def\p#1>{....}
\catcode ‘\< = 0
```

where in the second paragraph macro the argument #1 is empty.

But the s3 is a little trickier. If there are s3's, it's a safe bet that there are also s1 and s2's. The logical thing in normal TEX would be to define \s in such a way that it looks ahead to see what the next character is and then do an \ifcase based on the result. This is a little cumbersome, but quite straightforward.

Unfortunately, there are also <par-s> constructions. Therefore, you also need to define \par to do such a lookahead. Besides the fact that \par has a primitive meaning, it also has to do a rather large switch matching following tokens to predefined ‘element’ names, etc.; and this is why `\sgmlmode` was invented.

Watch out for the fact that once you get into `\sgmlmode`, spaces at the end of control sequences become required:

```
\sgmlmode=1
....
\sgmlmode= 0
```

creates a total of 2 tokens for the 3rd line: `"\sgmlmode="`, which will probably result in an ‘undefined csname’ error, and the command `"0"` that will be typeset!

3.4 New dimension specifiers

Eetex recognizes two new types of dimensions: `px` and `%`.

A `px` corresponds to a pixel, using a resolution of 96 dots per inch to calculate the conversion factor: `96px` equals `1in`.

The `%` is introduced because certain kinds of input use it to signify a percentage of a default or previous value. Setting up `%` in a meaningful turned out ot be quite tricky, so the current implementation maps one on one to `sp`: `100%` equals `100sp`. Of course, `%` only works if the `\catcode` of `%` is 12.

4 Compiling eetex

You need at least the following:

- The sources for web2c 7.3.1, like those found in the teTeX source distribution
- The source for a standard texmf tree, like the one found in the teTeX source distribution
- The eetex changes, these are in `eetex.tar.gz`
- The GNU C compiler, assembler and make tools.
- Some kind of lexer (like GNU flex).

4.1 Step 1

Extract the web2c sources and texmf tree somewhere.

4.2 Step 2

Change into the upper level `web2c` directory in the source tree, and extract the eetex.tar.gz from there.

4.3 Step 3

Run `./configure` (for teTEX, change to the directory `texk` above this one first).

4.4 Step 4

If needed, go back to the `web2c` directory

4.5 Step 5

Run `make pdftex`.

4.6 Step 6

Run `make eetex`.
This will create the two needed files: `eetex` and `eetex.pool`