The readarray Package

Routines for inputting formatted array data and recalling it on an element-by-element basis.
Currently supports 2-D and 3-D array structures

Steven B. Segletes
steven.b.segletes.civ@mail.mil

May 9, 2013
v1.2

1 Description and Commands

The readarray package allows for the inputting of data arrays (numeric, string, or even formatted) in either file form or `\def` form, such that the elements of multiple arrays can be specified and later recalled in an orderly fashion, on a cell-by-cell basis. Routines have been developed to support the storage and recall of both 2-D and 3-D arrays.

The commands included in this package help the user input data, define it in terms of array elements and recall those elements at will. Those commands are:
`\readdef`{*filename*}{*token*}
`\showrecord`[*error*]{*record number*}
`\copyrecords`{*identifier*}
`\readArrayij`{*token*}{*identifier*}{*columns*}
`\readArrayijk`{*token*}{*identifier*}{*rows*}{*columns*}
`\Arrayij`[*error*]{*identifier*}{*row*}{*column*}
`\Arrayijk`[*error*]{*identifier*}{*plane*}{*row*}{*column*}
`\arrayij`{*identifier*}{*row*}{*column*}
`\arrayijk`{*identifier*}{*plane*}{*row*}{*column*}

Several strings of fixed name are defined through the use of this package which are accessible to the user:
`\nrows`
`\ncols`
`\nrecords`
`\record`*index*

In addition to the strings of fixed name, there are various strings created whose name is a function of the user-specified data, such as
\identifier`CELLS`
\identifier`PLANES`
\identifier`ROWS`

\identifier`COLS`
where *identifier* is the alphabetic-character string by which you have designated a particular array. They will be discussed in relation to the commands that create these strings.

Support routines which are generally not required directly by the user for the specification and recall of data arrays, but which are useful to this package and in a variety of other circumstances include the following:
`\getargsC{`*token* or *string*`}`
`\arg`*index*
`\narg`
`\showargs[`*number*`]`
`\def\converttilde{`T or F`}`

[This Space Intentionally Left Blank]

2 Data Structure

The first requirement is to lay out a format for the data interface to this package. The readarray package is set up to digest space-separated data. The format for the data organization is as follows, for 2-D arrays:

$$\begin{array}{lllll}
A_{11} & A_{12} & A_{13} & \ldots & A_{1(\text{columns})} \\
A_{21} & A_{22} & \ldots & & \\
\vdots & & & & \\
A_{(\text{rows})1} & A_{(\text{rows})2} & A_{(\text{rows})3} & \ldots & A_{(\text{rows})(\text{columns})}
\end{array}$$

and for 3-D arrays:

$$\begin{array}{lllll}
A_{111} & A_{112} & A_{113} & \ldots & A_{11(\text{columns})} \\
A_{121} & A_{122} & \ldots & & \\
\vdots & & & & \\
A_{1(\text{rows})1} & A_{1(\text{rows})2} & A_{1(\text{rows})3} & \ldots & A_{1(\text{rows})(\text{columns})} \\
 & & & & \\
A_{211} & A_{212} & A_{213} & \ldots & A_{21(\text{columns})} \\
A_{221} & A_{222} & \ldots & & \\
\vdots & & & & \\
A_{2(\text{rows})1} & A_{2(\text{rows})2} & A_{2(\text{rows})3} & \ldots & A_{2(\text{rows})(\text{columns})} \\
 & & & & \\
\vdots & & & & \\
 & & & & \\
A_{(\text{planes})11} & A_{(\text{planes})12} & A_{(\text{planes})13} & \ldots & A_{(\text{planes})1(\text{columns})} \\
A_{(\text{planes})21} & A_{(\text{planes})22} & \ldots & & \\
\vdots & & & & \\
A_{(\text{planes})(\text{rows})1} & A_{(\text{planes})(\text{rows})2} & A_{(\text{planes})(\text{rows})3} & \ldots & A_{(\text{planes})(\text{rows})(\text{columns})}
\end{array}$$

3 Getting Data into Array Structures

One can provide data to be digested by this package in one of two ways: either through an external file, or by way of "cut and paste" into a `\def`. If one chooses `\readdef` the external file approach, the command `\readdef` is the command which can achieve this result. The command takes two arguments. The first is the file in which the data is stored, while the second is the token into which the data will be placed, for example

```
\readdef{data.txt}{\dataA}
```

In this case, the contents of the file `data.txt` will be placed into the token `\dataA`. At this point, the data is still not digested into a data "array," but merely stuffed into a `\def` (a `\protected@edef` actually). Thus, there is no *requirement* that carriage returns be part of the input file after each row of data, nor that blank lines exist between planes of data (if the data is 3-D). *However*, there is a reason to do so, nonetheless. In particular, for datafiles that are organized in the preferred fashion, for example:

```
A111 A112 A113 A114
A121 A122 A123 A124
A131 A132 A133 A134
A211 A212 A213 A214
A221 A222 A223 A224
A231 A232 A233 A234
```

`\ncols` `\nrows` `\nrecords` a `\readdef` will cause the following strings to be set: `\ncols` and `\nrows`, in this case to values of `4` and `3`, respectively. Such data could prove useful if the array size is not known in advance. When `\readdef` is invoked, a string `\nrecords` will also be set to the number of file records processed by the `\readdef` command.

In lieu of `\readdef`, a generally less preferred, but viable way to make the data available is to cut and paste into a `\def`. However, because a blank line is not permitted as part of the `\def`, a filler symbol (`%` or `\relax`) must be used in its place, if it is desired to visually separate planes of data, as shown in the `\def` example at the top of the following page. Note that the `%` is also required at the end of the line containing `\def`, in order to guarantee that, in this case, `A111` is the first element of data (and not a linebreak). However, unlike `\readdef`, this definition will set the values of neither `\ncols` nor `\nrows`.

```
\def{\dataA}{%
A111 A112 A113 A114
A121 A122 A123 A124
A131 A132 A133 A134
%
A211 A212 A213 A214
A221 A222 A223 A224
A231 A232 A233 A234
}
```

Once the data to be placed into an array is available by way of either `\readdef` or `\def`, the command to digest the data into an array is either `\readArrayij` (or `\copyrecords`), in the case of 2-D data, or `\readArrayijk`, for 3-D data.

3.1 Creating 2-D Arrays

`\readArrayij` In the case of `\readArrayij`, the command takes three arguments. The first is the token into which the data had previously been stuffed. The second is an alphabetic-string identifier for the array, which can be one or more characters in length. Finally, the last argument is the number of columns in the array. If the data had been read by way of `\readdef`, the string `\ncols` may be used to signify this value.

3.2 Creating Pseudo-1-D Arrays

While the readarray package has no explicit provisions for 1-D arrays, one could use the 2-D `\readArrayij` command, with the third argument set to a value of unity {1}, instead of `\ncols`. In this way, each space-separated word of the input file will be set to a new data row. While this approach is useful for sticking each "word" of input data into its own single-column data row, the command `\copyrecords` can be used to stick the individual "file records" from the most recent `\readdef` into a 2-D array of **single-column width**. The `\copyrecords` command takes as its argument an alphabetic string identifier to associate with the array of data. Its use accomplishes two things: 1) it allows records (rather than words) of a file to be accessed using the general `\Arrayij` nomenclature to be discussed shortly; and 2) it saves the most recently read file data into its own data structure, so that it is not overwritten by a subsequent invocation of `\readdef`.

`\copyrecords`

3.3 Creating 3-D Arrays

`\readArrayijk` For the 3-D case, `\readArrayijk` takes an additional argument, in comparison to `\readArrayij`. The first two arguments are identical to `\readArrayij`; namely, the token containing the data and an identifier for the array. The third argument is the number of rows in the array, while the fourth argument is the number of columns. Likewise, if `\readdef` had been used on a properly formed input file, both `\nrows` and `\ncols` may be used to supply the third and fourth arguments.

While it may be easily envisioned that the array data is numerical, this need not be the case. The data may be text, and even formatted text. Furthermore, one may introduce space characters into the data of individual cells through the use of hardspaces (~), since normal white space would otherwise be interpreted as a data separator. Thus, given the following definitions and array initialization,

```
\def\I#1{\textit{#1}}
\def\dataC{%
\I{am}  \I{are} have~\I{been} have~\I{been}
\I{are} \I{are} have~\I{been} have~\I{been}
\I{is}  \I{are} has~\I{been}  have~\I{been}
%
\I{was}  \I{were} had~\I{been} had~\I{been}
\I{were} \I{were} had~\I{been} had~\I{been}
\I{was}  \I{were} had~\I{been} had~\I{been}
%
will~\I{be} will~\I{be} will~have~\I{been} will~have~\I{been}
will~\I{be} will~\I{be} will~have~\I{been} will~have~\I{been}
will~\I{be} will~\I{be} will~have~\I{been} will~have~\I{been}
}
\readArrayijk{\dataC}{tobeConjugation}{3}{4}
```

multi-word sequences will be placed into the individual array elements of an array identified as "tobeConjugation," with the appropriate italic emphases applied to the words.

If, perchance, a row is only partially defined by `\readArrayij` or a plane is only partially defined by `\readArrayijk`, the partial data is discarded.

3.4 Treatment of Hardspaces

For either the 2-D or 3-D `\readArray` commands, the interpretation of hardspaces (`~`) as data is specifically designed to allow multi-word data entries. However, one may choose to turn this feature off by setting the flag

```
\def\converttilde{T}
```

which will have the effect of converting hardspaces to regular space tokens.

4 Recalling Data from Array Structures

\identifier`CELLS`
\identifier`PLANES`
\identifier`ROWS`
\identifier`COLS`

While one has specified the number of columns and/or rows associated with the `\readArray...` initialization, those numbers may not yet be known to the user, if the values employed came from the `\readdef` initializations of `\ncols` and `\nrows`. Therefore, the `\readArray...` commands also set the following strings: \identifier`CELLS`, \identifier`PLANES`\identifier`ROWS`, and \identifier`COLS`, where *identifier* is the array identifier string that was supplied to the `\readarray...` command. Note that it is the case, for 3-D arrays, that

\identifier`CELLS` = \identifier`PLANES` × \identifier`ROWS` × \identifier`COLS`

For the "tobeConjugation" example of the prior section, 36=3×3×4. Likewise, for 2-D arrays

\identifier`CELLS` = \identifier`ROWS` × \identifier`COLS`

`\Arrayij` `\Arrayijk` To retrieve the data from the array, one need employ either the `\Arrayij` or `\Arrayijk` commands, depending on whether the array is 2-D or 3-D. The first mandatory argument to either of these commands is the array identifer. The remaining arguments to these commands are simply the row and column, in the case of the 2-D `\Arrayij`, or else the plane, row, and column, in the case of the 3-D `\Arrayijk`.

Thus, in the case of the earlier example involving conjugation of the verb *to be*, the second-person future-perfect tense of the verb is given by
`\Arrayijk{tobeConjugation}{3}{2}{4}`,
which yields "will have *been*."

For pseudo-1-D arrays, either created with `\copyrecords` or else using the `\readarray` command with the third argument set to unity, access to these arrays is achieved by way of `\Arrayij`, setting the column argument to unity.

`\record`*index* There are also strings defined, one for each record that was read from the file, with the name `\record`*index*, where *index* is the record number expressed in roman numerals. Thus in the example from section 3, `\recordii` would contain the string "`A121 A122 A123 A124`". However, such syntax provides no bounds checking.

`\showrecord` An alternate (perhaps preferred) way to access one of the file records read during the most recent `\readdef`, in a way which provides bounds checking, is to use the `\showrecord` command. For the section 3 example, the alternative to executing `\recordii` would be to use `\showrecord{2}`. The optional argument to this command provides an alternative error message if the record requested is out of bounds. The default error messages for negative or excessive record numbers are, respectively:

Only positive showrecords permitted

RECORD=10 exceeds records read

Unlike the `\Arrayij` and `\Arrayijk` commands already discussed, the record data from a given file read will only be available until the next invocation of `\readdef`. For this reason, the command `\copyrecords` was introduced to convert file-record data into an array data structure.

While the user is developing his application involving the readarray package, there may accidentally arise the unintended circumstance where an array element is asked for which falls outside the array bounds. Like `\showrecord`, the `\Arrayij` and `\Arrayijk` commands also employ bounds checking. These commands check for four error conditions that, by default, produce the following error messages:

```
PLANE=9 exceeds limit for tobeConjugation
ROW=9 exceeds limit for tobeConjugation
COL=9 exceeds limit for tobeConjugation
```

which are written in lieu of a valid array element datum. While such messages help in debugging, the user may desire an error tailored to the application. The optional argument to both `\Arrayij` and `\Arrayijk` replaces the default error messages and is to be printed if any error condition arises. It could be a blank `[]`, a black box `[\rule{1ex}{1ex}]`, or anything else that makes sense for the application.

5 Alternate Accessing Commands if `\edef` is Required

`\arrayij` `\arrayijk`

The`\Arrayij` and `\Arrayijk` commands may, of course, be placed as arguments of `\def` commands. They, cannot, however, be placed into an `\edef`. If the user has need to place the array-element content into an `\edef` (and assuming that the actual array-element content is suitable for an `\edef`), the user should employ the `\arrayij` and `\arrayijk` commands, as alternatives. These commands do not perform any bounds checking (thus eliminating the need for an optional argument). Their advantage, however, is that they may be freely placed into an `\edef`.

6 Support Routines

`\getargsC` `\arg`*index* `\narg` `\showargs`

The engine for the readarray package is the `\getargsC` command, based on the `\getargs` command found in the stringstrings package. This command has been herein rewritten for speed. The `\getargsC` command takes a string or token as its argument and separates each of the space-separated words of it into individual strings, named with roman numerals as `\argi`, `\argii`, *etc.* The total number of arguments that are separated is given by the string `\narg`. This command can be useful for a variety of applications outside of readarray. While generally used only for diagnostic purposes, the command `\showargs` is used print out all the arguments recently digested by an invocation of `\getargsC`, separated by small black blocks. The optional argument to `\showargs` is the number of individual arguments to place on a single line of output, before issuing a linefeed. Thus, the command

```
\showargs[4]
```

yields the following result:

narg=37: ▮*am*▮*are*▮have *been*▮have *been*▮
▮*are*▮*are*▮have *been*▮have *been*▮
▮*is*▮*are*▮has *been*▮have *been*▮
▮*was*▮*were*▮had *been*▮had *been*▮
▮*were*▮*were*▮had *been*▮had *been*▮
▮*was*▮*were*▮had *been*▮had *been*▮
▮will *be*▮will *be*▮will have *been*▮will have *been*▮
▮will *be*▮will *be*▮will have *been*▮will have *been*▮
▮will *be*▮will *be*▮will have *been*▮will have *been*▮
■

Note that the 37^{th} argument is non-printing and represents the residual linefeed left over fromt he `\def` of `\dataC`. A similar residual argument is also left by the `\readdef` command. It can be eliminated if the last record of the `\readdef` input file (or the last record of the `\def` command) ends with a `%` symbol, so as to discard the final linefeed prior to the end-of-file (or `}`).

`\converttilde` As mentioned earlier, the flag `converttilde`, by default "false," can be set to `\def\converttilde{T}` so as to tell `\getargsC` to treat hardspaces as ordinary spaces.

7 Acknowledgements

The author would like to thank Dr. David Carlisle for his assistance in helping the author rewrite the `\getargs` command, originally found in the stringstrings package. To distinguish the two versions, and in deference to him, it is herein named `\getargsC`.

8 Code Listing

```
\ProvidesPackage{readarray}
[2013/05/09 v1.2
Routines for inputting array data and recalling it on an
element-by-element basis.  Currently supports 2-D and 3-D array]
%
% This work may be distributed and/or modified under the
% conditions of the LaTeX Project Public License, either version 1.3
% of this license or (at your option) any later version.
% The latest version of this license is in
%   http://www.latex-project.org/lppl.txt
% and version 1.3c or later is part of all distributions of LaTeX
% version 2005/12/01 or later.
%
% This work has the LPPL maintenance status ‘maintained’.
%
% The Current Maintainer of this work is Steven B. Segletes.
%
% Revisions:
% v1.01 Documentation revision
% v1.1  Added \csname record\roman{@row}\endcsname to \readdef
% v1.2  -Corrected the [truncated] LPPL license info
%       -Added \arrayij and \arrayijk, which can be put into \edef
%       -Used \romannumeral in preference to \roman{}, when possible,
%        to avoid unnecessary use of counters.
\usepackage{ifthen}
\usepackage{ifnextok}
%
\newcounter{@index}
\newcounter{@plane}
\newcounter{@row}
\newcounter{@col}
\newcounter{use@args}
\newcounter{@record}
\def\the@rule{\rule{.8ex}{1.6ex}}%
%
\newcommand\readArrayijk[4]{%
  \getargsC{#1}%
  \setcounter{@plane}{\numexpr(\narg/#4/#3)}%
  \setcounter{use@args}{\numexpr\arabic{@plane}*#3*#4}%
  \ifthenelse{\arabic{use@args} > \narg}{%
    \addtocounter{@plane}{-1}%
    \setcounter{use@args}{\numexpr\arabic{@plane}*#3*#4}%
  }{}%
  \expandafter\edef\csname#2PLANES\endcsname{\arabic{@plane}}%
  \expandafter\edef\csname#2ROWS\endcsname{#3}%
  \expandafter\edef\csname#2COLS\endcsname{#4}%
  \expandafter\edef\csname#2CELLS\endcsname{\arabic{use@args}}%
  \setcounter{@index}{0}%
  \setcounter{@plane}{1}%
  \setcounter{@row}{1}%
  \setcounter{@col}{0}%
  \whiledo{\value{@index} < \value{use@args}}{%
    \addtocounter{@index}{1}%
    \addtocounter{@col}{1}%
    \ifthenelse{\value{@col} > #4}%
      {\addtocounter{@row}{1}%
       \addtocounter{@col}{-#4}}%
      {}%
    \ifthenelse{\value{@row} > #3}%
      {\addtocounter{@plane}{1}%
       \addtocounter{@row}{-#3}}%
      {}%
    \expandafter\protected@edef%
         \csname#2X\roman{@plane}X\roman{@row}X\roman{@col}\endcsname%
      {\expandafter\csname arg\roman{@index}\endcsname}%
  }%
}
%
\newcommand\readArrayij[3]{%
  \getargsC{#1}%
  \setcounter{@row}{\numexpr(\narg/#3)}%
  \setcounter{use@args}{\numexpr\arabic{@row}*#3}%
  \ifthenelse{\arabic{use@args} > \narg}{%
    \addtocounter{@row}{-1}%
    \setcounter{use@args}{\numexpr\arabic{@row}*#3}%
  }{}%
  \expandafter\edef\csname#2PLANES\endcsname{1}%
  \expandafter\edef\csname#2ROWS\endcsname{\arabic{@row}}%
  \expandafter\edef\csname#2COLS\endcsname{#3}%
  \expandafter\edef\csname#2CELLS\endcsname{\arabic{use@args}}%
  \setcounter{@index}{0}%
  \setcounter{@row}{1}%
  \setcounter{@col}{0}%
  \whiledo{\value{@index} < \value{use@args}}{%
    \addtocounter{@index}{1}%
    \addtocounter{@col}{1}%
    \ifthenelse{\value{@col} > #3}%
      {\addtocounter{@row}{1}%
       \addtocounter{@col}{-#3}}%
      {}%
    \expandafter\protected@edef%
                    \csname#2X\roman{@row}X\roman{@col}\endcsname%
      {\expandafter\csname arg\roman{@index}\endcsname}%
  }%
}
%
\def\nonposrecordmessage{{\tiny Only positive showrecords permitted}}
\def\recordmessage#1{{\tiny RECORD=#1 exceeds records read}}
%
\def\nonposmessage{{\tiny Only positive array indices permitted}}
\def\planemessage#1#2{{\tiny PLANE=#2 exceeds limit for #1}}
\def\rowmessage#1#2{{\tiny ROW=#2 exceeds limit for #1}}
\def\colmessage#1#2{{\tiny COL=#2 exceeds limit for #1}}
%
\newcommand\arrayijk[4]{%
  \csname#1X\romannumeral#2X\romannumeral#3X\romannumeral#4\endcsname%
}
%
\newcommand\Arrayijk[5][$]{%
  \ifthenelse{\value{@plane}<1 \OR \value{@row}<1 \OR \value{@col}<1}{%
    \if$#1\nonposmessage\else#1\fi}{%
    \ifthenelse{#3 > \csname#2PLANES\endcsname}{%
      \if$#1\planemessage{#2}{#3}\else#1\fi}{%
      \ifthenelse{#4 > \csname#2ROWS\endcsname}{%
        \if$#1\rowmessage{#2}{#4}\else#1\fi}{%
        \ifthenelse{#5 > \csname#2COLS\endcsname}{%
          \if$#1\colmessage{#2}{#5}\else#1\fi}{%
          \csname#2X\romannumeral#3X\romannumeral#4X\romannumeral#5\endcsname}%
        }%
      }%
  }%
}
%
\newcommand\arrayij[3]{%
  \csname#1X\romannumeral#2X\romannumeral#3\endcsname%
}
%
\newcommand\Arrayij[4][$]{%
  \ifthenelse{\value{@row}<1 \OR \value{@col}<1}{%
    \if$#1\nonposmessage\else#1\fi}{%
    \ifthenelse{#3 > \csname#2ROWS\endcsname}{%
      \if$#1\rowmessage{#2}{#3}\else#1\fi}{%
      \ifthenelse{#4 > \csname#2COLS\endcsname}{%
        \if$#1\colmessage{#2}{#4}\else#1\fi}{%
        \csname#2X\romannumeral#3X\romannumeral#4\endcsname}%
      }%
  }%
}
%
\newcommand\copyrecords[1]{%
  \setcounter{@record}{0}%
  \whiledo{\value{@record} < \nrecords}{%
    \addtocounter{@record}{1}%
    \expandafter\protected@edef\csname#1X\roman{@record}Xi\endcsname{%
      \csname record\roman{@record}\endcsname}%
  }%
  \expandafter\edef\csname#1PLANES\endcsname{1}%
  \expandafter\edef\csname#1ROWS\endcsname{\nrecords}%
  \expandafter\edef\csname#1COLS\endcsname{1}%
  \expandafter\edef\csname#1CELLS\endcsname{\nrecords}%
}
%
\newcommand\showrecord[2][$]{%
  \ifthenelse{#2<1}{%
    \if$#1\nonposrecordmessage\else#1\fi}{%
    \ifthenelse{#2 > \nrecords}{%
      \if$#1\recordmessage{#2}\else#1\fi}{%
        \csname record\romannumeral#2\endcsname}%
  }%
}
%
\newcommand\readdef[2]{%
\def\first@row{T}%
\def\first@plane{T}%
\catcode\endlinechar=10\relax%
\def#2{}%
\setcounter{@record}{0}%
\newread\file%
\openin\file=#1%
\loop\unless\ifeof\file%
    \read\file to\fileline % Reads a line of the file into \fileline%
    \addtocounter{@record}{1}%
    \protected@edef#2{#2\fileline}%
    \if T\first@row\getargsC{#2}\setcounter{@col}{\numexpr(\narg-1)}%
      \edef\ncols{\arabic{@col}}\def\first@row{F}\setcounter{@row}{1}%
    \else%
      \if T\first@plane\getargsC{\fileline}\ifthenelse{\equal{\narg}{1}}{%
        \edef\nrows{\arabic{@row}}\def\first@plane{F}}{%
        \addtocounter{@row}{1}}%
      \fi%
    \fi%
    \expandafter\protected@edef\csname record\roman{@record}\endcsname{%
      \fileline}%
\repeat%
\edef\nrecords{\arabic{@record}}%
\closein\file%
\catcode\endlinechar=5\relax%
}
%
\newcommand\showargs[1][0]{%
  narg=\narg:~%
  \the@rule%
  \setcounter{arg@index}{0}%
  \setcounter{break@count}{0}%
  \whiledo{\value{arg@index} < \narg}{%
    \addtocounter{arg@index}{1}%
    \addtocounter{break@count}{1}%
    \csname arg\roman{arg@index}\endcsname%
    \the@rule%
    \ifthenelse{\equal{#1}{\value{break@count}}}{%
      \ifthenelse{\equal{\value{arg@index}}{\narg}}{}{%
        \newline\the@rule\addtocounter{break@count}{-#1}}%
      }%
    {}%
  }%
  \setcounter{arg@index}{0}%
}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% The \getargsC macro mimics the behavior of the \getargs macro
% of the stringstrings package, but runs faster, and can handle
% arbitrary tokens.  For the development of \getargsC, significant
% assistance was provided by David Carlisle, for which the author is
% most appreciative.
% http://tex.stackexchange.com/questions/101604/
%      parsing-strings-containing-diacritical-marks-macros
%
\def\string@end{$\SaveHardspace}
\def\converttilde{F}
\newcounter{arg@index}
\newcounter{break@count}
\let\SaveHardspace~%%%
%
\def\getargsC#1{%
  \if T\converttilde\def~{ }\else\catcode`~=12\fi
  \protected@edef\the@string{#1}%
  \setcounter{arg@index}{0}%
  \lowercase{\expandafter\parse@Block\the@string} \string@end
  \let~\SaveHardspace%
  \catcode`~=13
}
%
\def\parse@Block#1 {%
  \stepcounter{arg@index}%
  \@namedef{arg\roman{arg@index}}{#1}%
  \futurelet\tmp\parse@Block@}
%
\def\parse@Block@{%
\ifx\tmp\string@end\edef\narg{\thearg@index}\expandafter\@gobble
\else\expandafter\parse@Block\fi}
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
\endinput
```