The algxpar package[*]

Jander Moreira moreira.jander@gmail.com

June 26, 2023

Abstract

The algxpar package is an extension of the algorithmicx[1]/algpseudocode package to handle multi-line text with proper indentation and provide a number of other improvements.

Contents

[*]This document corresponds to algxpar v0.99, dated 2023/06/26. This text was last revised October 16, 2023.

[1]https://ctan.org/pkg/algorithmicx.

Description: LZW Compression using a table with all known sequences of bytes.
Input: A flow of bytes
Output: A flow of bits with the compressed representation of the input bytes

```
 1: Initialize a table with all bytes                ▷ each position of the table has a single byte
 2: Initilize sequence with the first byte in the input flow
 3: while there are bytes in the input do            ▷ wait until all bytes are processed
 4:     Get a single byte from input and store it in byte
 5:     if the concatention of sequence and byte is in the table then
 6:         Set sequence to $sequence + byte$        ▷ concatenate without producing any output
 7:     else
 8:         Output the code for                      ▷ i.e., the binary representation of its position in the
              sequence                                 table
 9:         Add the concatention of sequence and byte to the      ▷ the table learns a longer
              table                                                 sequence
10:         Set sequence to byte                     ▷ starts a new sequence with the remaining byte
11:     end if
12: end while
13: Output the code for sequence                     ▷ the remaining sequence of bits
```

1 Introduction

I teach algorithms and programming and have adopted the algorithmicx package (algpseudocode) for writing my algorithms as it provides clear and easy to read pseudocodes with minimal effort to get a visually pleasing code.

The process of teaching algorithms requires a slightly different use of pseudocode than that normally presented in scientific articles, in which the solutions are presented in a more formal and synthetic way. Students work on more abstract algorithms often preceding the actual knowledge of a programming language, and thus the logic of the solution is more relevant than the variables themselves. Likewise, the use of the development strategy by successive refinements also requires a less programmatic and more verbose code. Thus, when discussing the reasoning for solving a problem, it is common to use sentences such as "*accumulate current expenses in the total sum of costs*", because "$s \leftarrow s + c$" is, in this case, too synthetic and necessarily involves knowing how variables work in programs.

The consequence of more verbose pseudocode leads, however, to longer sentences that often span two or more lines. As pseudocodes, by nature, value visual organization, with regard to control structures and indentations, it became necessary to develop a package that supports the use of commands and comments that could be easily displayed when more than one line was needed.

The algorithmx and algpseudocode packages do not natively support multi-line statements. This package therefore extends several macros to handle multiple lines correctly. Some new commands and a number of features have also been added.

2 Package usage and options

This package depends on the following packages:

algorithmicx	(`https://ctan.org/pkg/algorithmicx`)
algpseudocode	(`https://ctan.org/pkg/algorithmicx`)
amssymb	(`https://ctan.org/pkg/amsfonts`)
fancyvrb	(`https://ctan.org/pkg/fancyvrb`)
pgfmath	(`https://ctan.org/pkg/pgf`)
pgfopts	(`https://ctan.org/pkg/pgf`)
ragged2e	(`https://ctan.org/pkg/ragged2e`)
tcolorbox	(`https://www.ctan.org/pkg/tcolorbox`)
varwidth	(`https://www.ctan.org/pkg/varwidth`)
xcolor	(`https://www.ctan.org/pkg/xcolor`)

To use the package, simply request its use in the preamble of the document.

`[`⟨*package options list*⟩`]`

Currently, the list of package options includes the following.

⟨*language name*⟩

By default, algorithm keywords are developed in English. The English language keyword set is always loaded. When available, other sets of keywords in other languages can be used simply by specifying the language names. The last language in the list is automatically set as the document's default language.
Currently supported languages:

- `english` (default language, always loaded)
- `brazilian` Brazilian Portuguese

`language = `⟨*language name*⟩

This option chooses the set of keywords corresponding to ⟨*language name*⟩ as the default for the document. This option is available as a general option (see `language`).
This option is useful when other languages are loaded.

`noend`

The `noend` suppresses the line that indicates the end of a block, keeping the indentation.
See more information in `end` and `noend` options.

3 Writting pseudocode

Algorithms, following the functionality of the algorithmicx package, are written within the environment. The possibility of using a number to determine how the lines will be numbered is maintained as in the original version.

An algorithm is composed of instructions and control structures such as conditionals and loops. And also, some documentation and comments.

```
**Description**: Calculation of the factorial of a natural number
**Input**: $n \in \mathbb{N}$
**Output**: $n!$

    read $n$
    $factorial \leftarrow 1$                              ▷ $0! = 1! = 1$
    for $k \leftarrow 2$ to $n$ do                        ▷ *from 2 up*
        $factorial \leftarrow factorial \times k$         ▷ $(k-1)! \times k$
    end for
    write $factorial$
```

3.1 A preamble on comments

This is the Euclid's algorithm as provided in the algorithmicx package documentation[2].

```
1: procedure EUCLID$(a, b)$                       ▷ The g.c.d. of a and b
2:     $r \leftarrow a \bmod b$
3:     while $r \neq 0$ do                        ▷ We have the answer if r is 0
4:         $a \leftarrow b$
5:         $b \leftarrow r$
6:         $r \leftarrow a \bmod b$
7:     end while
8:     return $b$                                 ▷ The gcd is b
9: end procedure
```

Comments are added *in loco* with the macro, which makes them appear along the right margin. The algxpar package embeded comments as part of the commands themselves in order to add multi-line support.

Until algxpar v0.95, they could be added as an optional parameter before the text, in the style of most LaTeX macros.

```
1: procedure EUCLID$(a, b)$
2:     $r \leftarrow a \bmod b$
3:     while $r \neq 0$ do                        ▷ We have the answer if r is 0
4:         $a \leftarrow b$
5:         $b \leftarrow r$
6:         $r \leftarrow a \bmod b$
7:     end while
8:     return $b$                                 ▷ The gcd is b
9: end procedure
```

Using the comment before the text always bothered me somewhat, as it seemed more natural to put it after. Thus, as of v0.99, the comment can be placed after the text (as the second parameter of the macro), certainly making writing algorithms more

[2]A label was supressed here.

user-friendly. To maintain backward compatibility, the use of comments before text is still supported, although it is discouraged.

In addition to this change, the use of comments in the new format has been extended to most pseudocode macros, such as for example.

```
procedure EUCLID($a, b$)                              ▷ The g.c.d. of a and b
    $r \leftarrow a \bmod b$
    while $r \neq 0$ do                               ▷ We have the answer if r is 0
        $a \leftarrow b$
        $b \leftarrow r$
        $r \leftarrow a \bmod b$
    end while                                         ▷ end loop
    return $b$                                        ▷ The gcd is b
end procedure
```

Using still produces the expected result, although it may break automatic tracking of longer lines.

Throughout this documentation, former style comments are denoted as ⟨*comment**⟩, while the new format uses ⟨*comment*⟩.

See more about comments in section 3.8.

3.2 A preamble on options

As of version 0.99, a list of options can be added to each command, changing some algorithm presentation settings. These settings are optional and must be entered using angle brackets at the end of the command.

```
IF $a > b$ THEN                     ▷ check conditions
    WHILE $a > 0$ DO
        PROCESS($a$)                ▷ process current data
    END WHILE
END IF
```

There is a lot of additional information about options and how they can be used. See discussion and full list in section 4.

3.3 Statements

The macros and defined in algorithmicx can still be used for single statements and have the same general behaviour.

For automatic handling of comments and multi-line text, the macro is available, which should be used instead of .

[⟨*comment**⟩]{⟨*text*⟩}[⟨*comment*⟩]<⟨*options*⟩>

The macro corresponds to an statement that can extrapolate a single line. The continuation of each line is indented from the baseline and this indentation is based on the value indicated in the `statement indent` option.

Any ⟨*options*⟩ specified uniquely affect this macro.

As an example, observe lines 8 and 9 of the LZW compression algorithm on page 19.

3.4 Flow Control Blocks

Flow control is essentially based on conditionals and loop.

3.4.1 The if block

This block is the standard *if* block.

```
read $v$
if $v < 0$ then                                        ▷ is it negative?
    $v \leftarrow -v$                                  ▷ make it positive
end if
```

[⟨*comment**⟩]{⟨*text*⟩}[⟨*comment*⟩]<⟨*options*⟩>

shows ⟨*text*⟩ (the condition) and must be closed with an , creating a block of nested commands.
Any of the ⟨*options*⟩ specified in this macro will affect this command and all items in the inner block, propagating up to and including the closing macro.

[⟨*comment*⟩]<⟨*options*⟩>

closes its respective .
Any ⟨*options*⟩ specified uniquely affect this macro.

[⟨*comment*⟩]<⟨*options*⟩>

This macro defines the **else** part of the statement.
Any of the ⟨*options*⟩ specified in this macro will affect this command and all items in the inner block, propagating up to and including the closing macro.

[⟨*comment**⟩]{⟨*text*⟩}[⟨*comment*⟩]<⟨*options*⟩>

defines the chaining. The argument ⟨*text*⟩ is the new condition.
Any of the ⟨*options*⟩ specified in this macro will affect this command and all items in the inner block, propagating up to and including the closing macro.

3.4.2 The switch block

```
Get option
switch option
    case 1 do                                          ▷ inserts new record
        INSERT(record)
    end case
    case 2 do                                          ▷ deletes a record
        DELETE(key)
    end case
    otherwise
        Print “invalid option”
    end otherwise
end switch
```

[⟨*comment**⟩]{⟨*expression*⟩}[⟨*comment*⟩]<⟨*options*⟩>

The is closed by a matching .
Any of the ⟨*options*⟩ specified in this macro will affect this command and all items in the inner block, propagating up to and including the closing macro.

[⟨*comment*⟩]<⟨*options*⟩>

This macro closes a block.
Any ⟨*options*⟩ specified uniquely affect this macro.

[⟨*comment**⟩]{⟨*constant-list*⟩}[⟨*comment*⟩]<⟨*options*⟩>

When the result of the **switch** expression matches one of the constants in ⟨*constants-list*⟩, then the **case** is executed. Usually the ⟨*constant-list*⟩ is a single constant, a comma-separated list of constants or some kind of range specification.
Any of the ⟨*options*⟩ specified in this macro will affect this command and all items in the inner block, propagating up to and including the closing macro.

[⟨*comment*⟩]<⟨*options*⟩>

This macro closes a corresponding statement.
Any ⟨*options*⟩ specified uniquely affect this macro.

[⟨*comment*⟩]<⟨*options*⟩>

A **switch** structure can optionally use an **otherwise** clause, which is executed when no previous **case**s had a hit.
Any of the ⟨*options*⟩ specified in this macro will affect this command and all items in the inner block, propagating up to and including the closing macro.

[⟨*comment*⟩]<⟨*options*⟩>

This macro closes a corresponding statement.
Any ⟨*options*⟩ specified uniquely affect this macro.

3.4.3 The for block

The **for** loop uses and is also flavored with two variants: **for each** () and **for all** ().

```
for $i \leftarrow 0$ to $n$ do
    Do something with $i$
end for
for all $item \in C$ do
    Do something with $item$
end for
for each $item$ in queue $Q$ do
    Do something with $item$
end for
```

[⟨*comment**⟩]{⟨*text*⟩}[⟨*comment*⟩]<⟨*options*⟩>

The ⟨*text*⟩ is used to establish the loop scope.
Any of the ⟨*options*⟩ specified in this macro will affect this command and all items in the inner block, propagating up to and including the closing macro.

[⟨*comment*⟩]<⟨*option*⟩>

This macro closes a corresponding , or .
Any ⟨*options*⟩ specified uniquely affect this macro.

[⟨*comment**⟩]{⟨*text*⟩}[⟨*comment*⟩]<⟨*options*⟩>

Same as .

[⟨*comment**⟩]{⟨*text*⟩}[⟨*comment*⟩]<⟨*options*⟩>

Same as .

3.4.4 The while block

is the loop with testing condition at the top.

- -

```
while $n > 0$ do
    Do something
    $n \leftarrow n - 1$
end while
```

[⟨*comment**⟩]{⟨*text*⟩}[⟨*comment*⟩]<⟨*options*⟩>

In ⟨*text*⟩ is the boolean expression that, when FALSE, will end the loop.
Any of the ⟨*options*⟩ specified in this macro will affect this command and all items in the inner block, propagating up to and including the closing macro.

[⟨*comment*⟩]<⟨*options*⟩>

This macro closes a matching block.
Any ⟨*options*⟩ specified uniquely affect this macro.

3.4.5 The repeat-until block

The loop with testing condition at the bottom is the / block.

- -

```
repeat
    Do something
    $n \leftarrow n - 1$
until $n \leq 0$
```

[⟨*comment*⟩]<⟨*options*⟩>

This macro starts the **repeat** loop, which is closed with .
Any of the ⟨*options*⟩ specified in this macro will affect this command and all items in the inner block, propagating up to and including the closing macro.

[⟨*comment**⟩]{⟨*text*⟩}[⟨*comment*⟩]<⟨*options*⟩>

In ⟨*text*⟩ is the boolean expression that, when , will end the loop.
Any ⟨*options*⟩ specified uniquely affect this macro.

3.4.6 The loop block

A generic loop is build with .

```
loop
    Do something
    $n \leftarrow n + 1$
    if $n$ is multiple of 5 then
        continue                                  ▷ restarts loop
    end if
    Do something else
    if $n \leq 0$ then
        break                                     ▷ ends loop
    end if
    Keep working
end loop
```

[⟨*comment*⟩]<⟨*options*⟩>

The generic loop starts with and ends with . Usually the infinite loop is interrupted by and internal or restarted with .
Any of the ⟨*options*⟩ specified in this macro will affect this command and all items in the inner block, propagating up to and including the closing macro.

[⟨*comment*⟩]<⟨*options*⟩>

closes a matching block.
Any ⟨*options*⟩ specified uniquely affect this macro.

3.5 Constants and Identifiers

A few macros for well known constants were defined: (True), (False), and (Nil).

The macro was created to handle "program-like" named identifiers, such as *sum*, *word_counter* and so on.

{⟨*identifier*⟩}

Identifiers are in italics: is *value*. Its designed to work in both text and math modes: is offer_k.

3.6 Assignments and I/O

To support teaching-like, basic pseudocode writing, the macros and are provided.

```
read $v_1, v_2$
$mean \leftarrow \frac{v_1 + v_2}{2}$                                    ▷ calculate
write $mean$
```

The macro can be used for assignments.

{⟨*lvalue*⟩}{⟨*expression*⟩} (deprecated)

This macro expands to .
As the handling of text and math modes should be done and its usage brings no evident advantage, this macro will no longer be supported. It will be kept as is for backward compatibility however.

3.7 Procedures and Functions

Modularization uses or .

```
procedure SAVENODE($node$)                               ▷ saves a $B^+$-tree node to disk
    if $node.is\_modified$ then
        if $node.address == -1$ then
            Set file writting position after file's last byte      ▷ creates a new node on disk
        else
            Set file writting position to $node.address$                 ▷ updates the node
        end if
        Write $node$ to disk
        $node.is\_modified \leftarrow$ FALSE
    end if
end procedure
```

```
function FACTORIAL($n$)                                                        ▷ $n \geq 0$
    if $n \in \{0, 1\}$ then
        return 1                                                           ▷ base case
    else
        return $n \times$ FACTORIAL($n - 1$)                                ▷ recursive case
    end if
end function
```

{⟨*name*⟩}{⟨*argument list*⟩}[⟨*comment*⟩]<⟨*options*⟩>

This macro creates a **procedure** block that must be ended with .
Any of the ⟨*options*⟩ specified in this macro will affect this command and all items in the inner block, propagating up to and including the closing macro.

[⟨*comment*⟩]<⟨*optons*⟩>

This macro closes the block.
Any ⟨*options*⟩ specified uniquely affect this macro.

`{`⟨*name*⟩`}{`⟨*argument list*⟩`}[`⟨*comment*⟩`]<`⟨*options*⟩`>`

This macro creates a **function** block that must be ended with . A is defined.
Any of the ⟨*options*⟩ specified in this macro will affect this command and all items in the inner block, propagating up to and including the closing macro.

`[`⟨*comment*⟩`]<`⟨*optons*⟩`>`

This macro closes the block.
Any ⟨*options*⟩ specified uniquely affect this macro.

For calling a procedure or function, should be used.

`{`⟨*name*⟩`}{`⟨*arguments*⟩`}<`⟨*options*⟩`>`

is used to state a function or procedure call. The module's ⟨*name*⟩ and ⟨*arguments*⟩ are mandatory.
Any ⟨*options*⟩ specified uniquely affect this macro.

3.8 Comments

The macro defined by algorithmicx has the same original behavior and has been redefined to handle styling options.

`{`⟨*text*⟩`}<`⟨*options*⟩`>`

The redesigned version of can be used with , and . When used with , it must be enclosed inside the text braces, but multi-line statements should work differently than expected.
Any ⟨*options*⟩ specified uniquely affect this macro.

```
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    Store the value zero in variable x            ▷ first
    assignment
    Store the value zero in variable x            ▷ first
       assignment
    Store the value zero in variable x    ▷ first assign-
                                            ment
```

`{`⟨*text*⟩`}<`⟨*options*⟩`>`

While pushes text to the end of the line, the macro is "local". In other words, it just puts a comment in place.
Local comments follows regular text and no line changes are checked.
Any ⟨*options*⟩ specified uniquely affect this macro.

```
if $a > 0$  ▷ *special case*
   or
   $a < b$  ▷ *general case*
   then
      Process data  ▷ *may take a while*
end if
```

{⟨*text*⟩}<⟨*options*⟩>

is an alternative to *line comments* which usually extends to the end of the line. This macro defines a comment with a begin and an end. A comment starts with ▷ and ends with ◁.
Any ⟨*options*⟩ specified uniquely affect this macro.

```
if $a > 0$ ▷ *special case* ◁ or $a < b$ ▷ *general case* ◁ then
   Process data  ▷ *may take a while*
end if
```

3.9 Documentation

A series of macros are defined to provide the header documentation for a pseudocode.

Description: Calculation of the factorial of a natural number through successive multiplications
Require: $n \in \mathbb{N}$
Ensure: $f = n!$

⟨*description text*⟩

The is intended to hold the general description of the pseudocode.

⟨*pre-conditions*⟩

The required initial state that the code relies on. These are *pre-conditions*.

⟨*post-conditions*⟩

The final state produced by the code. These are *post-conditions*.

Description: Calculation of the factorial of a natural number through successive multiplications
Input: n (integer)
Output: $n!$ (integer)

⟨*inputs*⟩

This works as an alternative to , presenting **Input**.

⟨*outputs*⟩

This works as an alternative to , presenting **Output**.

4 Customization and Fine Tunning

As of version 0.99 of algxpar, a series of options have been introduced to customize the presentation of algorithms. Colors and fonts that only apply to keywords, for example, can be specified, providing an easier and more convenient way to customize each algorithm.

The macro serves this purpose.

{⟨*options list*⟩}

This macro sets algorithmic settings as specified in the ⟨*options list*⟩, which is key/value comma-separated list.
All settings will be applied to the entire document, starting from the point of the macro call. The scope of a definition made with can be restricted to a part of the document simply by including it in a TEX group.

```
read $k$
if $k < 0$ then
               $k \gets -k$
end if
write $k$
```

If the settings are only applied to a single algorithm and not a group of algorithms in a text section, the easiest way is to include the options in the `algorithmicx` environment.

```
read $k$
if $k < 0$ then
    $k \gets -k$
end if
write $k$
```

Named styles can also be defined using the pgfkeys syntax.

```
// Process $k$
read $k$
if $k < 0$ then
    $k \gets -k$                                                        // back to positive
end if
write $k$
```

Sometimes some settings need to be applied exclusively to one command, for example to highlight a segment of the algorithm.

```
▷ Process $k$
read $k$
if $k < 0$ then
    $k \leftarrow -k$                                        ▷ back to positive
end if
write $k$
```

4.1 Options

This section presents the options that can be specified for the algorithms, either using or the ⟨*options*⟩ parameter of the various macros.

`language` = ⟨*language*⟩ *Default:* `english`

This key is used to choose the keyword language set for the current scope. The language keyword set should already have been loaded through the package options (see section 2).

`noend`

Structured algorithms use blocks for its structures, marking their begin and end. In pseudocode it is common to use a line to finish a block. Using the option `end`, this line is suppressed.
The result is similar to a program written in Python.

`end`

This option reverses the behaviour of `end`, and the closing line of a block presented.

```
for $i \leftarrow 0$ to $N - 1$ do
    for $j \leftarrow$ to $N - 1$ do
        if $m_{ij} < 0$ then
            $m_{ij} \leftarrow 0$
        end if
```

`keywords` = ⟨*list of keywords assignments*⟩

This option allows to change a keyword (or define a new one). See section 4.2 for more information on keywords and translations.

```
whilst TRUE do
    if $t < 0$ {
        Run the Terminate module
    }
end whilst
```

`algorithmic indent` = ⟨*width*⟩ *Default:* `1em`

The algorithmic indent is the amount of horizontal space used for indentation inner commands.
This option actually sets the algorithmicx's .

`comment symbol` = ⟨*symbol*⟩ *Default:*

The default symbol that preceeds the text in comments is (▷), as used by algorithmicx, and can be changed with this key.
The current comment symbol is available with . Do not change this symbol by redefining , as font, shape and color settings will no longer be respected. Always use `comment symbol`.

`comment symbol right` = ⟨*symbol*⟩ *Default:*

This is the symbol that closes a . This symbol is set to ◁ and can be retrieved with the macro. Do not attempt to change the symbol by redefining , as font, shape and color settings will no longer be respected. Always use `comment symbol right`.

4.1.1 Fonts, shapes and sizes

The options ins this section allows setting font family, shape, weight and size for several parts of an algorithm.

Notice that color are handled separately (see section 4.1.2) and using with font options will tend to break the document.

`text font` = ⟨*font, shape and size*⟩ *Default:* –empty–

This setting corresponds to the font family, its shape and size and applies to the ⟨*text*⟩ field in each of the commands.

`comment font` = ⟨*font, shape and size*⟩ *Default:*

This setting corresponds to the font family, its shape and size and applies to all comments.

`keyword font` = ⟨*font, shape and size*⟩ *Default:*

This setting sets the font family, shape, and size, and applies to all keywords, such as **function** or **end**.

`constant font` = ⟨*font, shape and size*⟩ *Default:*

This setting sets the font family, shape, and size, and applies to all constants, such as Nil, True and False.
This setting also applies when is used.

`module font` = ⟨*font, shape and size*⟩ *Default:*

This setting sets the font family, shape, and size, and applies to both procedure and function identifiers, as well as their callings with .

4.1.2 Colors

Colors are defined using the xcolors package.

`text color` = ⟨*color*⟩ *Default:* `.` (dot)

This setting corresponds to the color that applies to the ⟨*text*⟩ field in each of the commands.

`comment color` = ⟨*color*⟩ *Default:* `.!70`

This setting corresponds to the color that applies to all comments.

`keyword color` = ⟨*color*⟩ *Default:* `.` (dot)

This key is used to set the color for all keywords.

`constant color` = ⟨*color*⟩ *Default:* `.` (dot)

This setting corresponds to the color that applies to the defined constant (see section 3.5) and also when macro is used.

`module color` = ⟨*color*⟩ *Default:* `.` (dot)

This color is applied to the identifier used in both and definitions, as well as module calls with . Notice that the arguments use `text color`.

4.1.3 Paragraphs

Multi-line support are internally handled by es.

```
procedure EUCLID($a, b$)                          ▷ The g.c.d. of a and b
    $r \leftarrow a \bmod b$
    while $r \neq 0$ do                           ▷ We have the answer if r is 0
        $a \leftarrow b$
        $b \leftarrow r$
        $r \leftarrow a \bmod b$
    end while
    return $b$                                    ▷ The g.c.d. is b
end procedure
```

The options in this section should be used to set how these paragraphs will be presented.

`text style` = ⟨*style*⟩ *Default:*

This ⟨*style*⟩ is applied to the paragraph box that holds the ⟨*text*⟩ field in all commands.

`comment style` = ⟨*style*⟩ *Default:*

This ⟨*style*⟩ is applied to the paragraph box that holds the ⟨*comment*⟩ field in all algorithmic commands. This setting will not be used with , or .

`comment separator width` = ⟨*width*⟩ *Default:* `1em`

The minimum space between the text box and the . This affects the available space in a line for keywords, text and comment.

`statement indent` = ⟨*width*⟩ *Default:* `1em`

This is the set inside statements.

`comment width` = `auto|nice|`⟨*width*⟩ *Default:* `auto`

There are two ways to balance the lengths of ⟨*text*⟩ and ⟨*comments*⟩ on a line, each providing different visual experiences.
In automatic mode (`auto`), the balance is chosen considering the widths that the actual text and comment have, trying to reduce the total number of lines, given there is not enough space in a single line for the keywords, text , comment and comment symbol. The consequence is that each line with a comment will have its own balance.
The second mode, `nice`, sets a fixed width for the entire algorithm, maintaining consistency across all comments. In that case, longer comments will tend to span a larger number of lines. The "nice value" is hardcoded and sets the comment width to .
Also, a fixed comment width can be specified.

4.2 Languages and translations

A simple mechanism is employed to allow keywords to be translated into other languages.

```
procedimento EUCLID($a, b$)
    $r \leftarrow a \bmod b$
    enquanto $r \neq 0$ faça
        $a \leftarrow b$
        $b \leftarrow r$
        $r \leftarrow a \bmod b$
    fim enquanto
    retorne $b$
fim procedimento
```

Creating a new keyword set uses the macro.

`{`⟨*language name*⟩`}{`⟨*keyword assignments*⟩`}`

This macro sets new values for known keywords as well as new ones. Once created, keywords cannot be deleted.
In case a default keyword is not reset, the English version will be used.
To create a new set, copy the file `algxpar-english.kw.tex` and edit it accordingly.
Note that there is a set of keywords for the lines that close each block. These keys are provided to allow for more versatility in changing how these lines are presented. It is highly recommended that references to other keywords use the Keyworkd macro so that font, color and language changes can be made without any problems.

In translations, these *compound keywords* do not necessarily need to appear (see file `brazilian.kw.tex`, which follows the settings in `algxpar-english.kw.tex`). However, if defined, there will be different versions for each language.

The mechanism behind uses the macro, which is called to adjust the value of a single keyword[3]. To retrieve the value of a given keyword, the macro must be used. It returns the formatted value according to the options currently in use for keywords.

[⟨*language*⟩]{⟨*keyword*⟩}{⟨*value*⟩}

The macro changes a given ⟨*keyword*⟩ to ⟨*value*⟩ if it exists; otherwise a new keyword is created.
If ⟨*language*⟩ is omitted, the language currently in use is changed.
See also the `keywords` option.

[⟨*language*⟩]{⟨*keyword*⟩}

This macro expands to the value of a keyword in a ⟨*language*⟩ using the font, shape, size, and color determined for the keyword set.
If ⟨*language*⟩ is not specified, the current language is used. ⟨*keyword*⟩ is any keyword defined for a language, including custom ones.

Depending on the language, a keyword can take different forms: **if** (English), **wenn** (German) or **se** (Brazilian Portuguese).

4.3 Other features

[⟨*name*⟩]

This macro presents ⟨*name*⟩ using font, shape, size and color defined for constants.

[⟨*name*⟩]

This macro presents ⟨*name*⟩ using font, shape, size and color defined for procedures and functions.

5 To do

This is a *todo* list:

- Add font, shape, size and color settings to a whole algorithm;
- Add font, shape, size and color settings to line numbers;
- Add font, shape, size and color settings to identifiers.

[3]Macros like from the algorithimicx package are no longer used.

6 Examples

6.1 LZW revisited

Description: LZW Compression using a table with all known sequences of bytes.
Input: A flow of bits
Output: A flow of bits with the compressed representation of the input bytes

```
1:  Initialize a table with all bytes              ▷ each position of the table has a single byte
2:  Initilize sequence with the first byte in the input flow
3:  while there are bytes in the input do          ▷ wait until all bytes are processed
4:      Get a single byte from input and store it in byte
5:      if the concatention of sequence and byte is in the table then
6:          Set sequence to sequence + byte        ▷ concatenate without producing any output
7:      else
8:          Output the code for sequence           ▷ i.e., the binary representation of its position in the table
9:          Add the concatention of sequence and
              byte to the table                    ▷ the table learns a longer sequence
10:         Set sequence to byte                   ▷ starts a new sequence with the remaining byte
11:     end if
12: end while
13: Output the code for sequence                   ▷ the remaining sequence of bits
```

6.2 LZW revisited again

Description: LZW Compression using a table with all known sequences of bytes.
Input: A flow of bytes
Output: A flow of bits with the compressed representation of the input bytes

```
1:  Initialize a table with all bytes              ▷ each position of the table has a single byte
2:  Initilize sequence with the first byte in the input flow
3:  while there are bytes in the input do          ▷ wait until all bytes are processed
4:      Get a single byte from input and store it in byte
5:      if the concatention of sequence and byte is in the table then
6:          Set sequence to sequence + byte        ▷ concatenate without producing any output
7:      else
8:          Output the code for sequence           ▷ i.e., the binary representation of its position in the table
9:          Add the concatention of sequence and byte to the
              table                                ▷ the table learns a longer sequence
10:         Set sequence to byte                   ▷ starts a new sequence with the remaining byte
11: Output the code for sequence                   ▷ the remaining sequence of bits
```

Index